HELSINKI UNIVERSITY OF TECHNOLOGY
TEKNILLINEN KORKEAKOULU
Laboratory of Telecommunications Technology

An Implementation of the Internet Call Waiting Service using SIP

Inmaculada Espigares del Pozo

Supervisor: Professor Francisco J. Mora
Instructor: Professor Raimo Kantola

Espoo, December 1999

Helsinki University of Technology **Abstract of the Master's Thesis**

Author:	Inmaculada Espigares del Pozo
Title of the Thesis:	An Implementation of the Internet Call Waiting Service using SIP.
Date:	December 1999 Number of pages: 85
Faculty:	Laboratory of Telecommunications Technology Helsinki University of Technology (HUT)
Supervisor:	Professor Francisco J. Mora
Instructor:	Professor Raimo Kantola M. Sc. Jose M. Costa Requena

Nowadays telephony services can be approached using the traditional switched network or using Internet. Both approaches employ different ways to establish connections, transmit the voice and terminate calls. Multiple new services are being required and implemented.

For real time communications two different protocols are used to establish and release the connection between the two end systems: H.323 and SIP. Both protocols provide a similar set of services but SIP is much simpler because it has less logical components. This project focuses on SIP, the Session Initiation Protocol, but also compares it against the H.323 signaling and control protocol.

We have studied the SIP for the purpose of evaluating it and to make an implementation of a new service, the Internet Call Waiting (ICW). It is a useful solution for the calls that otherwise would be lost when the line is busy and also for rejecting undesirable incoming calls. On the other hand, it is a way of not wasting network resources and contributing to call completion. Thus, pop-up dialogue boxes are presented to make it simpler and easier to the user whose satisfaction is always an important objective for an IP based service.

For service implementation, as the main tool, we have used the XML language. XML is considered one of the best languages for describing complex data relationships. We have also chosen XML because it is easily extended, flexible and because it has a text-based syntax.

The complete project consists of a JAVA program that implements an UAS/UAC running in a PC and also an extension (embedding the XML parser) of the SIP server written in C borrowed from Columbia University to handle the scripts written in XML defining the service required by the users.

In conclusion, we have tried to use the most efficient tools and mechanisms to complete this work as we consider that time and money are resources to take into account when developing the services of the new era.

Keywords: XML, script, JAVA, C, parser, service

Preface

This master thesis has been written at the Laboratory of Telecommunications Technology in Helsinki, Finland. First of all, I would like to express my gratitude to the Helsinki University of Technology and the Polytechnic University of Valencia for giving me the opportunity to complete my studies and meet fantastic new people from different countries.

I would like to thank all my friends and colleagues from the department (specially Kimmo Pitkäniemi, Arja, Mirja and Anita), and from other institutions who have helped me to *survive* in Finland. I also sincerely appreciate Henning Schulzrinne’s support given in the first "SIP Bake off" hosted by Columbia University in April 1999, New York (USA). Thanks to him we received the license to work with the SIP server from Columbia University.

Truly thankful to my instructor, Raimo Kantola for his interest and guidance constantly, and for giving me the opportunity of testing the program in the third "SIP Bake off" hosted by Ericsson in December 1999, Richardson, Texas (USA). I cannot forget to mention my supervisor in Spain, Francisco J. Mora, for giving me the chance to develop my master thesis in Helsinki.

On the other hand, I would like to thank all my family, specially my parents, for staying always by my side. Finally, I would love to thank my *second* instructor, Jose Costa Requena, for his incessant and unconditional support given with such happiness and optimism all the time. Without him, nothing would have been the same.

December, 1999
Helsinki, Finland

Inmaculada Espigares del Pozo

Table of Contents

Abbreviations and Acronyms

ACK	ACKnowledge
API	Application Program Interface
ASCII	American Standard Code for Information Interchange
AVP	Audio Video Profile
CGI	Common Gateway Interface
CLC	C Libraries Collection
CPL	Call Processing Language
CR	Carriage Return
CRLF	Carriage Return and Linefeed
CU	Columbia University
DN	Destination Number
DTD	Document Type Definition
GDBM	GNU DataBase Management
GML	Generalized Markup Language
GNU	GNU’s Not Unix
GSTN	Global Switched Telephone Network
H.323	Internet Video Conferencing standards from ITU
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
HUT	Helsinki University of Technology
IETF	Internet Engineering Task Force
ICW	Internet Call Waiting
IETF	Internet Engineering Task Force
IGMP	Internet Group Management Protocol
IN	Intelligent Networks
IP	Internet Protocols
ISDN	Integrated Services Digital Network
ISO	International Standards Organization
ITU	International Telecommunications Union
JMF	Java Media Framework
MC	Multipoint Controller
MCU	Multipoint Control Unit
MMUSIC	Multiparty Multimedia SessIon Control
Mbone	Internet Multicast Backbone

NTP	Network Time Protocol
OSI	Open Source Initiative
PC	Personal Computer
PERL	Practical Extraction and Report Language
PSTN	Public Switched Telephone Network
QoS	Quality of Service
RFC	Request For Comments
RSVP	Resource ReserVation Protocol
RTCP	Real Time Control Protocol
RTP	Real Time Protocol
RTSP	Real Time Stream Protocol
SAP	Session Announcement Protocol
SCCP	Simple Conference Control Protocol
SCF	Service Control Function
SCP	Service Control Point
SDES	Source DESscription packets
SDP	Session Description Protocol
SGML	Standard Generalized Markup Language
SIP	Session Initiation Protocol
SN	Service Nodes
SS7	Signaling System 7
SP	Space Character
T.120	Internet Multimedia Conferencing standards from ITU
TCL	Tool Command Language
TCP	Transmission Control Protocol
UA	User Agent
UAC	User Agent Client
UAS	User Agent Server
UDP	User Datagram Protocol
UML	Unified Modeling Language
URI	Universal Resources Identifier
URL	Uniform Resource Locator
WWW	World Wide Web
W3C	World Wide Web Consortium
XML	eXtensible Markup Language
XSL	eXtensible Style Language

List of figures

List of tables

Chapter 1

Introduction

Telephony service today is provided for the most part over circuit-switched networks, which are referred to as Public Switched Telephone Networks (PSTN). A new trend that is emerging in recent years is to provide telephony service over IP networks, known as *IP Telephony* or *Voice over IP* [52].

An important driving force behind IP Telephony is cost savings, especially for corporations with large data networks. The high cost of long-distance and international voice calls is the crux of the issue. A significant portion of this cost originates from regulatory taxes imposed on long-distance voice calls. Such surcharges are not applicable to long distance circuits carrying data traffic; thus, for a given bandwidth, making a data call is much less expensive than making a voice call.

There are other very significant motivating factors for carrying voice traffic over data networks as well. A very important benefit of IP Telephony is the integration of voice and data applications, which can result in more effective business processes (e.g. teleconferencing, integrated voice mail and e-mail). Another benefit is the enabling of many new services both for businesses and for customers. The flexibility offered by IP Telephony by moving the intelligence from the network to the end stations, as well as the open nature of IP networks, are factors that enable new services.

In order for IP Telephony to gain mainstream acceptance and ultimately replace the traditional telephony service, two conditions have to be met. First, the quality of the voice communication must be at least at the same level. Secondly, the ease of operation and functionality offered to the end-user must be at least also the same. This condition requires the IP Telephony architecture to provide a signaling infrastructure that offers at least the same capabilities and features as the Signaling System 7 (SS7) architecture in PSTN.

More specifically, the signaling infrastructure must:

- ❑ provide the **functionality** required to set up, manage, and tear down calls and connections;
- ❑ be **scalable** to support a very large number of registered endpoints, and a very large number of simultaneous calls;
- ❑ support **network management** features for policy control, billing, etc.;
- ❑ provide a mechanism to communicate and set up the **Quality of Service** requested by the end points;
- ❑ be **extensible** to help with adding new features easily;
- ❑ support **interoperability** among different vendors' implementations, among different versions of the signaling protocol, and with different signaling protocols.

Nowadays two standards compete for IP Telephony signaling. The older standard is the ITU-T recommendation H.323, which defines a multimedia communications system over packet-switched networks, including IP networks [18]. The other standard, the Session Initiation Protocol (SIP) comes from the IETF MMUSIC working group [30]. For this project we have used SIP for different reasons described at the end of chapter 3 where we compare SIP and H.323 in terms of complexity, modularity, transport protocol neutrality, extensibility and services [19][20].

1.1 Purpose of the Master Thesis

As the title of the project says, our goal was to implement the Internet Call Waiting service. The Session Initiation Protocol (SIP) was used for signaling features and the Call Processing Language (CPL) for describing the service [1][34]. For that purpose we had to gain experience in the SIP signaling technology to take advantage of its characteristics to improve the service implementation. We could have chosen many other different services to be developed and to show how *easy* it is to create new services depending on user requirements using IP Telephony [7].

The main objective is to demonstrate how new services can be defined from end points attending to the end-user wishes. The mechanism briefly is as follows. A user decides that he/she wants a certain service, so he/she sends to the SIP server a REGISTER message, where his/her information and the definition of the service required are included. To write the service definition we have used XML for many reasons presented in chapter 6. When the server receives the REGISTER message, it reads the user preferences from the script included in the message [3]. Next the server stores the user's service information that will be consulted when a new call for the user arrives to the server. Users can define as many services as they desire, storing them in the server and enjoying them.

In this way, services in IP Telephony can be defined like a new concept of services *residing* in their own level independent of the signaling protocol.

1.2 Organization of this Thesis

The rest of this thesis is organized as follows. Chapter 2 gives an overview of the ITU and IETF multimedia conferencing standards [18][27]. Chapter 3 describes the Session Initiation Protocol (SIP) analyzing the structure of the messages (requests and responses) and giving examples to clarify the protocol behavior [1]. At the end of chapter 3 we study the most significant differences between the H.323 protocol and the Session Initiation Protocol [22].

In chapter 4 we introduce the Session Description Protocol (SDP) as an elemental tool for the development of this project [23]. The SDP is used to advertise multimedia conferences and communicate the conference addresses and conference tool-specific information necessary for participation.

Chapter 5 describes the Columbia SIP server behavior. This is the server we have used for testing and where we have included the necessary extensions to create new services. In chapter 6 we present the concept of the Call Processing Language used to define services and we also include an XML overview [34][37].

Chapter 7 could be considered the most interesting one. In this chapter we include all the details about our implementation, the Internet Call Waiting service. To complete this project we present our conclusions and summarize the work in chapter 8.

Chapter 2

Multimedia Conferencing Standards

This chapter provides some background on the current conferencing standards. The first section explains the relevant ITU conferencing standards focusing on the H.323 and T.120 series [18]. The second section examines the IETF conferencing standards [28].

It is difficult to show exactly how the ITU and IETF standards compare directly to each other. This is largely because the ITU standards are monolithic, while the IETF standards are divided by the function they serve. Many of the conferencing standards of the IETF and the ITU are complementary but a few are overlapping.

2.1 ITU Conferencing Standards

The key standards from the International Telecommunication Union (ITU-T) for Internet conferencing are the T.120 and H.323 series of standards [18].

The focus of the T.120 standard is on data protocols for multimedia conferencing. T.120 itself specifies the relation of the standards in the T.120 series, which come from Study Group 8 of the ITU. These standards also provide support for multipoint communications using a centralized mechanism called a Multipoint Conferencing Service (MCS).

The focus of the H.323 standard is on video conferencing on local area networks (LANs) without quality of service (QoS) guarantee, which includes most LANs on the Internet. H.323 specifies the relation of standards in the H.323 series, which comes from Study Group 15 of the ITU (Figure 1).

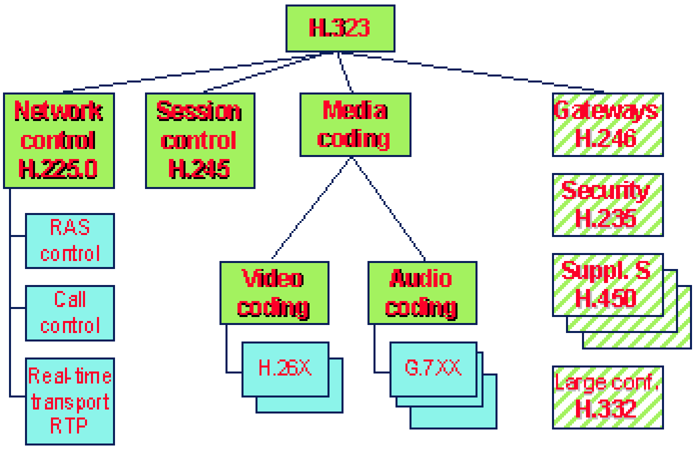


Figure 1. H323 umbrella

H.323 is basically an Internet "patch" of the H.320 series of standards, which specify videoconferencing over the PSTN. The T.120 and H.323 standards are complementary although it appears that there is an overlap between some of the call control functions [21].

2.2 IETF Conferencing standards

The focus of the IETF multimedia conferencing has been primarily on standards to enable the use of multicast for multimedia conferencing [28]. The working group in charge of the higher level standards specifically for multicast conferencing is called the Multiparty Multimedia Session Control (MMUSIC) group [30]. To have a full understanding of standards from MMUSIC, it is necessary to understand the underlying standards that they rely on from other IETF groups. The list of standards includes:

- **Internet Group Management Protocol (IGMP)** provides management functions for multicast routers such as a router joining a multicast tree.

- **Resource Reservation protocol (RSVP)** provides the setup protocol to reserve resources in a router and it can be used to request a guaranteed QoS in an IP network.

- **Real Time Protocol (RTP) and Real Time Control Protocol (RTCP).** RTP has a short header, which is added to UDP packets for carrying real time information such as audio or video. It includes time-stamp information for synchronizing audio and video and a sequence number for detecting lost packets and identifiers. RTCP provides control information such as feedback on the quality of data transmission. RTCP can also send Source Description (SDES) packets with information on members in a multicast group, which can be used to maintain a real-time list of membership in a conference [24].

2.2.1 MMUSIC STANDARDS

The Multiparty MUltimedia SessIon Control (MMUSIC) develops Internet standards track protocols to support Internet teleconferencing sessions [30]. MMUSIC's focus is on supporting the loosely controlled conferences that are pervasive on the MBone today.

To date, MMUSIC has drafted several protocols [29].

- **Session Description Protocol (SDP-**RFC 2327**)** and **Session Announcement Protocol (SAP)**. To distribute session descriptions (announcement of e.g. session's names, types of media, available protocols, session times, etc.) for the purposes of multimedia session initiation [23][26].
- **SAP Security**. To provide security for session announcements.
- **Real-time Stream Protocol (RTSP-**RFC 2326**)**. To control on demand delivery of real-time data [25].
- **Session Initiation Protocol (SIP)**. To initiate sessions and for inviting users to sessions [1].
- **Simple Conference Control Protocol (SCCP)**. To manage tightly controlled sessions [28].

The Session Initiation Protocol will be described and analyzed in detail in Chapter 3, as it is an important element of the study skeleton.

2.3 Conclusion

Providing an integrated Internet telephony service is no small task. It requires signaling protocols, transport protocols, directory protocols, service specification languages, gateway discovery protocols, and a host of other mechanisms. In this chapter, we have provided a general overview of some of the protocols required.

Chapter 3

The Session Initiation Protocol

"The Session Initiation Protocol (SIP) is an application-layer control (signaling) protocol for creating, modifying and terminating multimedia sessions or calls with one or more participants."

In this chapter we focus on the study of the Session Initiation Protocol [1]. Firstly, we give a general idea of the protocol presenting important issues such as its characteristics, its functionality, the structure of the message, and the behavior of the servers. Secondly, the Session Description Protocol (SDP) [23] is introduced and we conclude comparing the SIP and the H.323 protocols [21].

3.1 Main Characteristics of SIP

The Session Initiation Protocol has distinct characteristics including the following.

- It is designed to be independent of the lower-layer transport protocol. This is because it has its own reliability system and can work with UDP as well as with TCP transport protocols.
- It invites both persons and "robots" such as a media storage service.
- It invites parties to both unicast and multicast sessions.
- It can be used to initiate sessions as well as invite members to existing sessions.
- It supports transparently name mapping and redirection services, allowing the implementation of ISDN and Intelligent Network telephone subscriber services that enable personal mobility.
- **Personal mobility** is defined as the ability of end users to originate and

receive calls and access subscribed telecommunication services on any terminal in any location, and the ability of the network to identify end users as they move [6].

- It supports five facets of establishing and terminating multimedia communications:

  User location: determination of the end system to be used for communication.

  User capabilities: determination of the media and media parameters to be used.

  User availability: determination of the willingness of the called party to engage in communications.

  Call setup: establishments of call parameters at both called and calling party.

  Call handling: including transfer and termination of calls.

- It can initiate multiparty calls using a Multipoint Control Unit (MCU) or fully meshed interconnection instead of multicast.
- As we presented in Chapter 2, it is designed as part of the overall IETF multimedia data and control architecture [27].
- It can be used in conjunction with other call setup and signaling protocols [22].
- It does not offer conference control services and does not prescribe how a conference is to be managed, but SIP can be used to introduce conference control protocols.
- It can invite users to sessions with and without resource reservation.
- It is text-based. This allows easy implementation, easy debugging and makes SIP extensible and flexible.

3.2 SIP components

There are two components in a SIP system: a User Agent (UA) and a network server. A UA is an end system that acts on behalf of a user. Usually it consists of two parts, a client (UAC) and a server (UAS), as the user probably is wishing to both be able to call and to be called. The UAC is used to initiate a SIP request while the UAS receives requests and returns responses on behalf of the user.

There are two kinds of network servers, namely, the proxy and the redirect servers.

❑ A SIP proxy server forwards requests to the next server after deciding which one it should be. This next server could be any kind of SIP server and the proxy does not need to know the type of the next server. Before the request has reached the UAS it may have traversed several servers. Those will be traversed in reverse order by the response.

The SIP proxy server can be stateful or stateless. Table 1 shows the differences between the two models. When stateful, a proxy remembers the incoming request, which generated outgoing requests. For that purpose the stateful server creates new processes to attend each new incoming request. Instead, a stateless proxy sequentially processes each new request and forgets all the information once an outgoing request is generated. It has the benefits of less processing and memory requirements in the server.[2]

STATEFUL PROXY SERVER	STATELESS PROXY SERVER
Maintains call context	No call context
Replicates UAS/UAC to process requests/responses	Response is not based on UA replication
Call state and transaction state can be maintained	Provides client anonymity
Forking proxies require state	Restricted gateway access
TCP proxies must be stateful for reliability	High processing capacity
Enhanced services require state for execution	Allows for easier replication that the stateful
Can populate billing information	Can have semi-stateful proxy for ultimate benefits

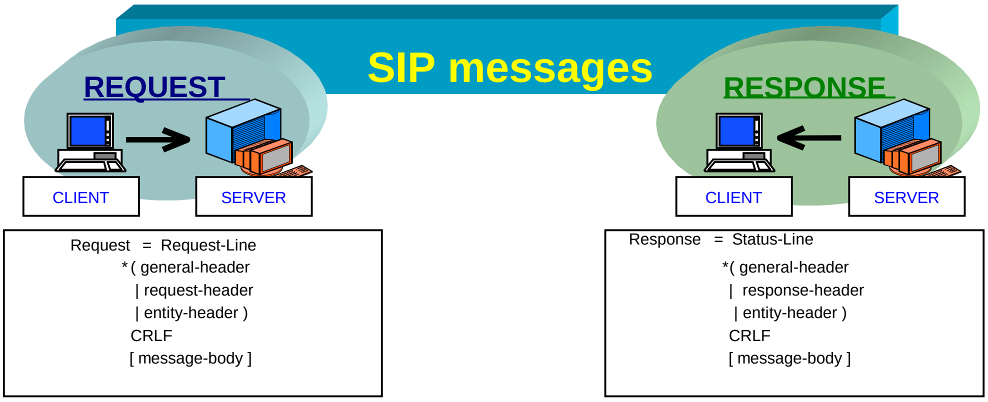
Table 1. Proxy server: stateful, stateless.

- A redirect server does not forward requests to the next server. Instead of that, it sends a redirect response back to the client containing the address of the next server to contact with.

There is also a server that accepts REGISTER requests, which is called the registrar. A registrar is typically co-located with a proxy or redirect server and may also offer location services.

3.3 SIP message

A SIP message is either a REQUEST from a client to a server or a RESPONSE from a server to a client. Figure 2 depicts the structure and headers of the SIP messages.



```
Request   =  Request-Line
          *( general-header
           | request-header
           | entity-header )
          CRLF
          [ message-body ]
```

```
Response   =  Status-Line
           *( general-header
            |  response-header
            | entity-header )
           CRLF
           [ message-body ]
```

Figure 2. SIP message

3.3.1 REQUEST MESSAGE

The request message begins with a *Request-Line* and after that the headers are added. Those headers can be *General-headers*, *Request-headers* or *Entity-headers*. Finally, after a blank line goes the message body if necessary [2].

The Request-Line format is:

Method *space* **Request-URI** *space* **SIP-Version** CRLF

The *SIP-version* is the protocol version used in the request message, e.g. SIP/2.0.

The *Request-URI* is a SIP URL or a general URI (the terms and generic syntax of URI and URL are defined in RFC 2396 [49]). It indicates the user or service to which the request is being addressed to.

Basically, the Uniform Resource Identifier (URI) is a compact string of characters for identifying an abstract or physical resource. The URI provides a simple and extensible means for identifying a resource. The specification of URI syntax and semantics is derived from concepts introduced by the World Wide Web global information initiative. URI is designed to meet the recommendations laid out in "Functional Recommendations for Internet Resource Locators" (RFC1736) and "Functional requirements for Uniform Resource Names" (RFC1737).

The URI syntax consists of a sequence of components separated by reserved characters such as ";", ":", "@", "?", and "/".

The term "Uniform Resource Locator" (URL) refers to the subset of URI that identify resources via a representation of their primary access mechanism (e.g., their network "location"), rather than identifying the resource by name or by some other attribute(s) of that resource. URL schemes that involve the direct use of an IP-based protocol to a specified server on the Internet use a common syntax for the server component of the URI's scheme-specific data:

<userinfo>@<host>:<port>, where <userinfo> may consist of a user name and, optionally, scheme-specific information about how to gain authorization to access the server. The parts "<userinfo>@" and ":<port>" may be omitted.

server = [[userinfo "@"] hostport]

The user information, if present, is followed by a commercial at-sign "@".

userinfo = *(unreserved | escaped | ";" | ":" | "&" | "=" | "+" | "$" | ",")

The host is a domain name of a network host, or its IPv4 address as a set of four decimal digit groups separated by "."

```
hostport      = host [ ":" port ]
host          = hostname | IPv4address
hostname      = *( domainlabel "." ) toplabel [ "." ]
domainlabel   = alphanum | alphanum *( alphanum | "-" ) alphanum
toplabel      = alpha | alpha *( alphanum | "-" ) alphanum
IPv4address   = 1*digit "." 1*digit "." 1*digit "." 1*digit
port          = *digit
```

Thus, basically the SIP URI follows strictly these recommendations for defining the resources location.

There are six possible *methods* that can be used in a request message. Those methods are *INVITE, ACK, BYE, CANCEL, OPTION* and *REGISTER*. The functions of these methods are described below.

- ❑ *INVITE*. This method indicates that a user or a service is being invited to participate in a session. The message body contains a description of the session to which the callee is being invited. This method has to be supported by a SIP proxy, redirect and user agent servers and user agent clients.

- ❑ *ACK* confirms that the client has received a final response to an *INVITE* request. This method is only used following an *INVITE* method. It may contain a message body with the final session description to be used by the callee. A SIP proxy, redirect and user agent servers and user agent clients must support it.

- ❑ *OPTIONS*. With this method a server is queried as to its capabilities. This method has to be supported by a SIP proxy, redirect and user agent servers and user agent clients.

- ❑ *BYE*. The UAC uses this method to indicate to the server that it wishes to release the call. It MUST be forwarded like an *INVITE* request and may be issued by either the caller or the callee. This method must be supported by proxy servers and should be supported by redirect and UA SIP servers.

- ❑ *CANCEL*. This method is used to cancel a pending request with the same *Call-ID*, *To*, *From* and *Cseq* header fields value. It does not affect a completed request, a request that has been processed or a response that has been sent back. It must be supported by proxy servers and should be supported by all other SIP server types.

- ❑ *REGISTER*. This method is used by a client to register the address listed in the *To* header field with a SIP server. The *REGISTER* request-header fields play an important role in our implementation and are defined as follows:

  Via indicates the path taken by the REGISTER request so far.

  To: This header contains the address-of-record whose registration is to be created or updated.
  From contains the address-of-record of the user responsible for the registration. For first party registration, it is identical to the *To* header field value.

  Request-URI names the destination of the registration request, i.e. the domain of the registrar. The user name must be empty.

  Call-ID: All registration from the same user should use the same *Call-ID* header value.

  Cseq: Registration with the same *Call-ID* must have increasing *Cseq* header values.

Contact: The request may contain a *Contact* header field; future non-REGISTER requests for the URI given in the *To* header field should be directed to the address/es given in the *Contact* header.

The client must send a *REGISTER* method after certain time to refresh the registration. If this does not happen, the server should drop the registration.

Let us look at an example using the REGISTER method for a third party registration. A user at host *tct.hut.fi* registers on start-up, via multicast, with the local SIP server named *hut.fi*.

From the client to the server:

```
REGISTER sip:hut.fi SIP/2.0
Via: SIP/2.0/UDP tct.hut.fi
From: sip:queca@hut.fi
To: sip:queca@hut.fi
Call-ID: 70710@tct.hut.fi
CSeq: 1 REGISTER
Contact: <sip:queca@tct.hut.fi:3890;transport=udp>
Expires: 7200
```

The Expires entity-header field gives the date and time after which the message content expires. This header field is currently defined only for the REGISTER and INVITE methods.

In a REGISTER request, the client indicates how long it wishes the registration to be valid. In the response, the server indicates the earliest expiration time of all registrations. The server may choose a shorter time interval than that requested by the client, but should not choose a longer one[10].

3.3.2 RESPONSE MESSAGE

The response message begins with a *Status-Line* then several headers are inserted [1]. Those headers can be *General-headers, Response-headers* or *Entity-headers* and after a blank line goes the message body, if necessary.
The Response-Line format is:

SIP-version *space* **Status-Code** *space* **Reason-Phrase** CRLF

The *Reason-Phrase* is intended to give a short textual description of the *Status-Code*. The *Status-Code* is a 3-digit integer that indicates the outcome of the attempt to understand and satisfy the request. The first digit of the *Status-Code* defines the class of the response. In SIP there are six types of responses due to the six possible values of the *Status-Code* first digit, and they are listed below:

1xx: **Informational** – the request was received and it is going to be processed.

2xx: **Success** – the action was successfully received, understood and accepted.

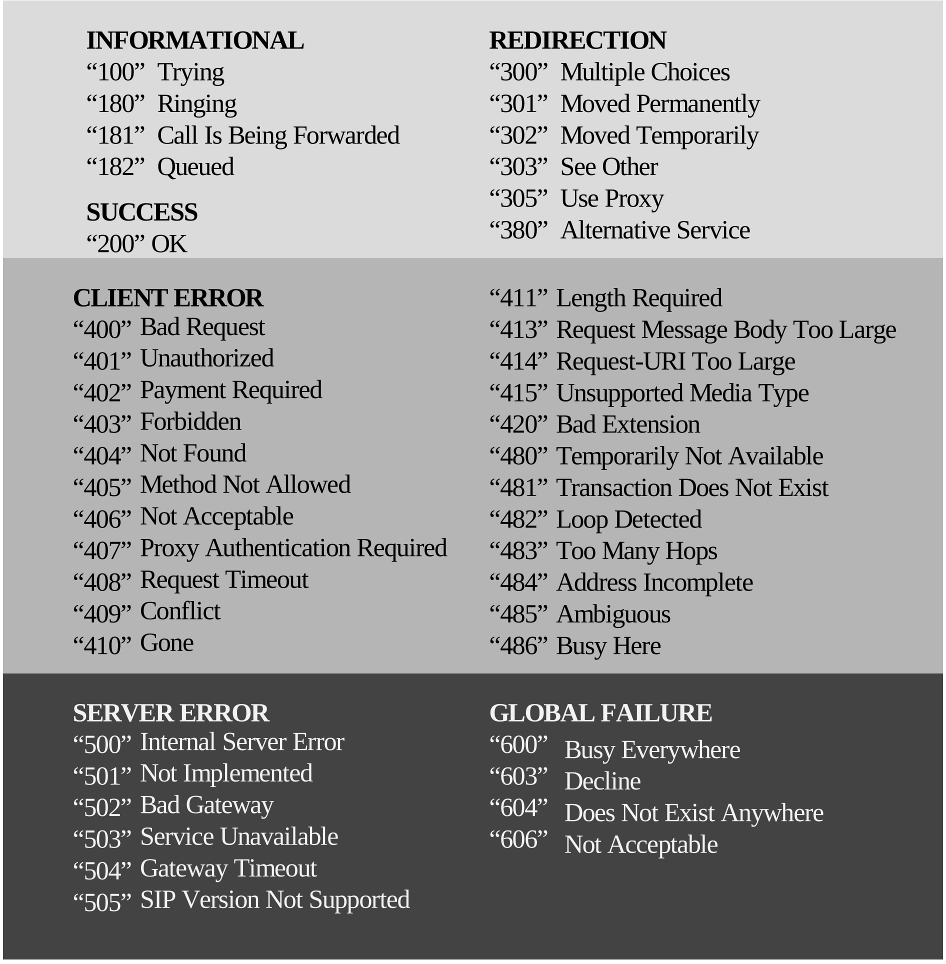
3xx: **Redirection** – further actions need to be taken in order to complete the request.

4xx: **Client Error** – the request contains bad syntax or cannot be fulfilled at this server.

***5xx:* Server Error** – the server failed to fulfil an apparently valid request.

***6xx:* Global Failure** – the request cannot be fulfilled at any server.

The SIP response codes are based on these six response types and can be easily extended because only a few of the possible codes are defined. It is necessary that the applications understand the class of any response code. The response codes defined are enlisted in Figure 3.



INFORMATIONAL

“100”	Trying
“180”	Ringing
“181”	Call Is Being Forwarded
“182”	Queued

SUCCESS

“200”	OK

REDIRECTION

“300”	Multiple Choices
“301”	Moved Permanently
“302”	Moved Temporarily
“303”	See Other
“305”	Use Proxy
“380”	Alternative Service

CLIENT ERROR

“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 Message Body Too Large
“414”	Request-URI Too Large
“415”	Unsupported Media Type
“420”	Bad Extension
“480”	Temporarily Not Available
“481”	Transaction Does Not Exist
“482”	Loop Detected
“483”	Too Many Hops
“484”	Address Incomplete
“485”	Ambiguous
“486”	Busy Here

SERVER ERROR

“500”	Internal Server Error
“501”	Not Implemented
“502”	Bad Gateway
“503”	Service Unavailable
“504”	Gateway Timeout
“505”	SIP Version Not Supported

GLOBAL FAILURE

“600”	Busy Everywhere
“603”	Decline
“604”	Does Not Exist Anywhere
“606”	Not Acceptable

Figure 3. Response Codes

3.3.3 HEADER FIELDS

The ordering of the header fields is not important in general [5]. The only requirement is that the header fields, which are hop-by-hop, must appear before any header fields, which are end-to-end. Other fields can be added as required and a server should ignore the headers that it does not understand. Proxy can add a *Via* field but it cannot reorder the existing headers.
Table 2 shows the set of headers.

GENERAL-HEADER	**ENTITY-HEADER**	**REQUEST-HEADER**	**RESPONSE-HEADER**
Accept	Content-Encoding	Authorization	Allow
Accept-Encoding	Content-Length	Contact	Proxy-Authenticate
Accept-Language	Content-Type	Hide	Retry-After
Call-ID		Max-Forwards	Server
Contact		Organization	Unsupported
Cseq		Priority	Warning
Date		Proxy-Authorization	WWW-Authenticate
Encryption		Proxy-Require	
Expires		Route	
From		Require	
Record-Route		Response-Key	
Timestamp		Subject	
To		User-Agent	
Via			

Table 2. Header Fields

- **GENERAL HEADER FIELD:** It is applied both to request and response messages. They can be extended reliably only in combination with a change in the protocol version, so the unrecognized header fields are treated as *entity-header* fields.

- **ENTITY HEADER FIELD:** They define information about the message-body or about the resource identified by the request.

- **REQUEST HEADER FIELD:** They allow the client to pass additional information about the response, which cannot be placed in the *Status-Line*. They can be extended reliably only in combination with a change in the protocol version. Unrecognized fields are treated as *entity-header* fields.

- **RESPONSE HEADER FIELD:** They allow the server to pass additional information about the response, which cannot be placed in the *Status-Line*. These header fields give information about the server and about further access to the resource identified by the *Request-URI*.

3.3.4 SIP MESSAGE BODY

All the requests MAY have a message body, except the *BYE* request. For the *INVITE*, *ACK* and *OPTIONS* requests it is always a session description. In most cases it will follow the Session Description Protocol (SDP) since SIP and SDP belong to the same protocol family [23]. The SDP is described in chapter 4.

The SIP message body length should be given in the *Content-Length* header field. And the *Content-Type* header field must give the type of the message. If the message is encoded, it must be indicated in the *Content-Encoding* header field.

3.4 Basic protocol functionality and operation

To understand the basic operation of the SIP protocol we have drawn some pictures showing the main transactions and how the different servers take part in the establishment of the desired communication between the caller and the callee.

3.4.1 SIP INVITATION

A successful SIP invitation consists of two requests [9]. The first one is always an *INVITE* and it is followed by an *ACK*. The *INVITE* request asks the callee to join a particular conference or to establish a two-party conversation. After the callee has agreed to participate in the call, the caller confirms that it has received that response by sending an *ACK* request. If the caller no longer wants to participate in the call, it sends a *BYE* request instead of an *ACK*. Figure 4 presents the basic transactions to setup a call.

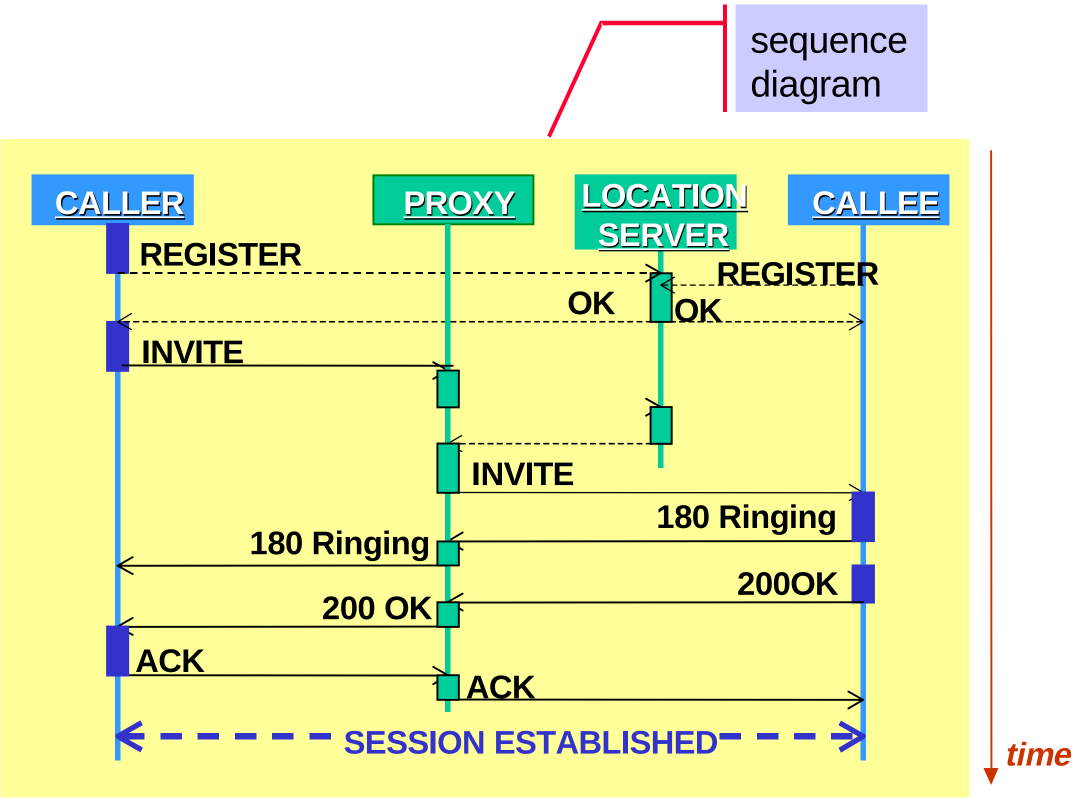


Figure 4. Call Setup (both endpoints registered, proxy routed call setup)

The *INVITE* request typically contains a session description that provides the called party with the information required to be able to join the session.

There are two different ways of establishing the communication depending on the server (proxy or redirect server) that we find on the way from the caller to the callee.

A. If the server is a PROXY SERVER, the establishment of the communication between the two parties is shown in Figure 5:

1. The caller sends the SIP *INVITE* request to the proxy server. The proxy server accepts it.
2. The proxy server contacts the location service with all or parts of the address. The SIP server, according to the *To* field, consults its own database to check if the user's registration already exists. Afterwards, the server tries to find other possible locations of the user. For that purpose, the SIP server sends a query to a remote LDAP server including the SIP URL received in the request. The locations returned by the LDAP server, are included in the Contact field of the following messages [32].
3. The proxy server obtains a more precise location.
4. The proxy server issues a SIP *INVITE* request to the address/es returned by the location service.
5. The user agent server alerts the user and returns a success indication to the proxy server.
6. The proxy server returns the success result (OK) to the original caller.
7. The caller using an *ACK* request confirms the receipt of this message.
8. The *ACK* request is forwarded to the callee and can also be sent directly to the callee by passing the proxy.

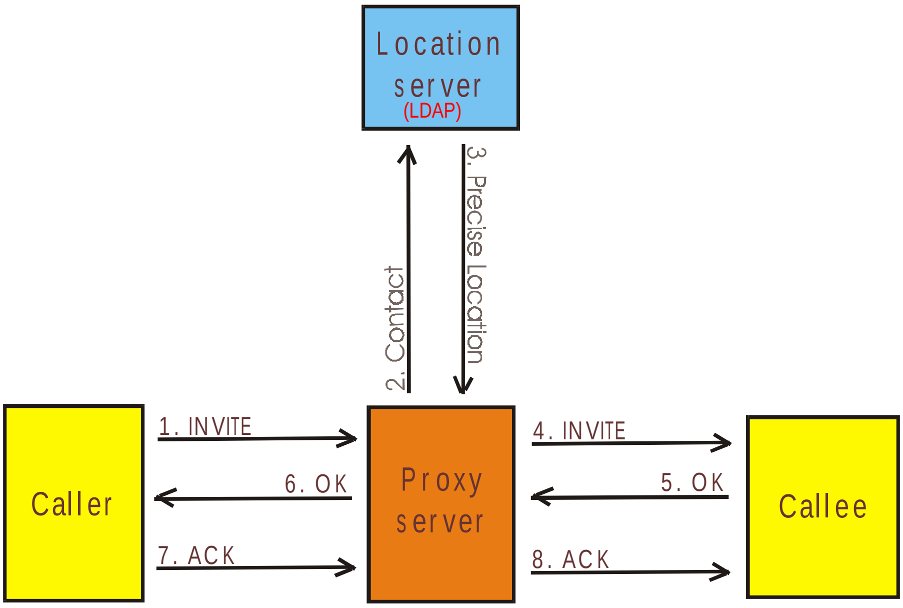


Figure 5. Communication through a proxy server

B. But if the server is a REDIRECT SERVER, the communication is completed in a different way, shown in Figure 6 and Figure 7.

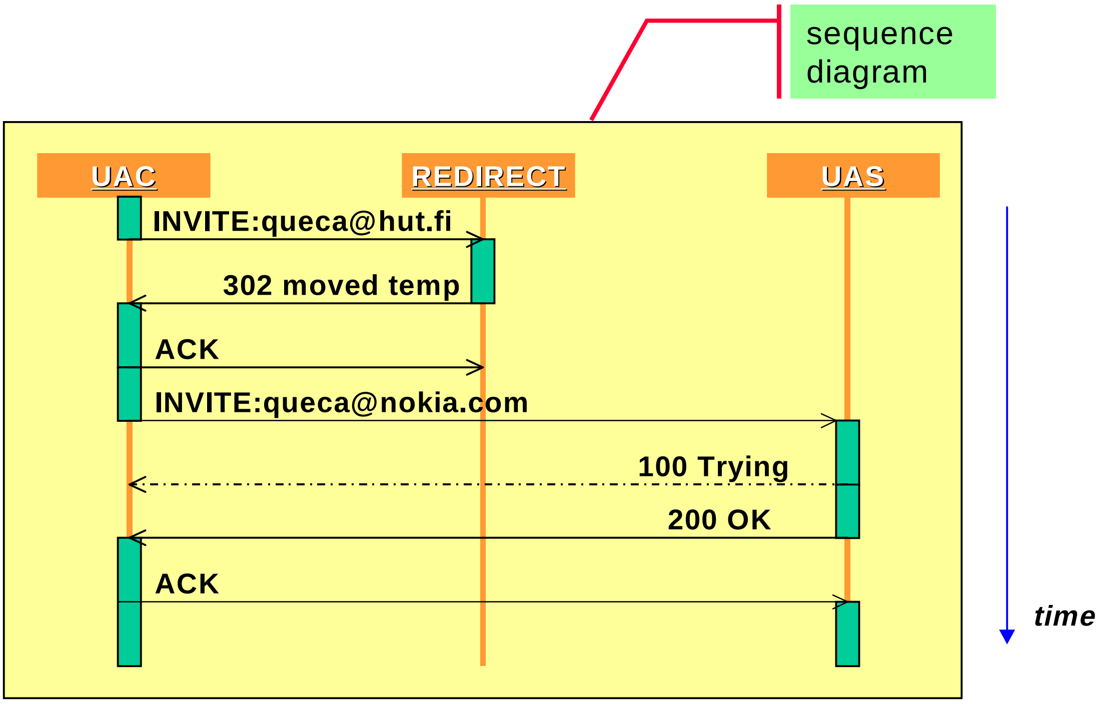


Figure 6. SIP operation in redirect mode

1. The caller sends the SIP *INVITE* request to the redirect server. The redirect server accepts it.
2. The redirect server contacts the location service with all or parts of the address as was explained in the proxy section.
3. The redirect server obtains a more precise location.
4. The redirect server returns the address to the caller.
5. The address is acknowledged from the caller via an *ACK* request.
6. The caller issues a new request, with the same *call-ID* but a higher *CSeq* to the address returned by the first server.
7. The call succeeds (OK).
8. The caller and the callee complete the handshake with an *ACK*.

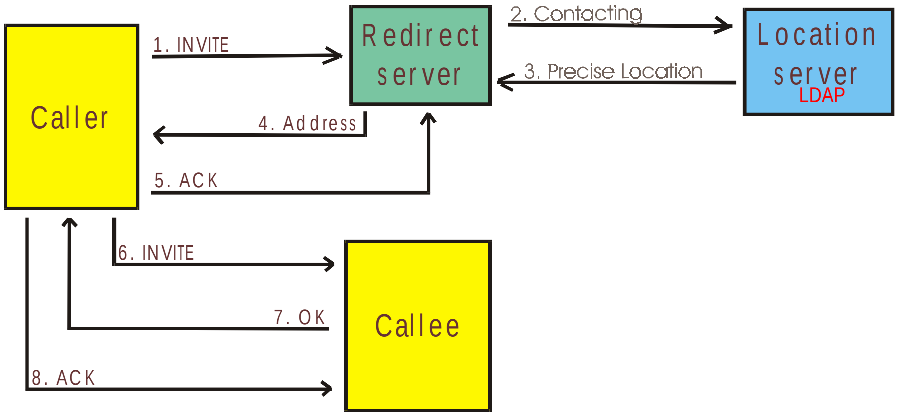


Figure 7. Communication through a redirect server

3.4.2 PROXY SERVER versus REDIRECT SERVER

After analyzing the different ways of establishing the communication, in Table 3 we include a comparison between both kinds of servers. The Redirect server requires a minimal implementation to process the incoming requests and gives back the address of the next server to contact with. Instead, the proxy server needs more computational resources to process the requests and to send back a response according to the request contents. The advantage of the proxy is the flexibility provided for service implementations in the server management request according to user requirements [2].

PROXY SERVER	REDIRECT SERVER
Can be stateful or stateless.	High capacity.
Flexible, network service implementations.	Minimal state overhead.
Reliability achieved through replication.	Service execution pushed to client.
Scalability achieved through partitioning.	
Overload potential, if not properly scaled.	

Table 3. Proxy server vs Redirect server

3.4.3 LOCATING A USER

When a proxy or a redirect server contacts a location server, it can respond with a list of zero or more addresses where the user can be reached. This is because the user may be logged at more than one host or because the location server has inaccurate information. The action taken on receiving a list of locations varies with the type of the SIP server.

A redirect server will tell the caller all the addresses where the callee can be located and it is the caller, which decides what to do.

A proxy server will try the addresses given by the location server sequentially or in parallel until the call is successful or the callee has reclined the call.

To follow the way the message has taken, when a proxy server forwards a SIP request it adds a *VIA-header* to the message with its address. It is done so the response can follow the same way back. The order of the *VIA-headers* is important: a new *VIA-header* must be added to the end of the *VIA-headers* list of the message. This is also useful to prevent loops. A proxy server must not forward a request to a server that is already in the *VIA-header* list.

3.4.4 SIP ADDRESSES

The *objects* addressed by SIP are users at hosts identified by a SIP URL [49]. The SIP URL has the form:

user@host

- ❑ The *user* part is a user name, a civil name or a telephone number.
- ❑ The *host* part is either a domain name having e.g. a numeric network address.

A SIP address can designate an individual, the first available person from a group of individuals or a whole group. SIP URLs are used within SIP messages to indicate the originator, the current destination and the final recipient of a SIP request. Also they are used to specify redirection addresses, although this is not always true because some of the addresses mentioned above may be non-SIP URLs. SIP URLs can also be embedded in web pages or other hyperlinks indicating the use of the *INVITE* method.

3.4.5 SIP MOBILITY EXAMPLE

In this section we make use of one of the most tempting SIP characteristics [6]. It is *supporting advanced personal mobility services.* We describe a possible real situation to present how the call forwarding service works. We have used several pictures to clarify the distinct steps.

A user of the system, Queca, works in Nokia Research Center and she has a nice office there. She also has an office in Helsinki University of Technology (HUT) and she sometimes assists courses in the telecommunications laboratory.

Figure 8a shows how one day, as Queca does not want to loose the calls arriving to her office in Nokia, she decides to use the Call Forwarding service. She sends a REGISTER message to the Nokia SIP Server listing her HUT address (*queca@hut.fi)* as a forwarding address. The same picture shows how the HUT SIP Server is notified of the two Queca's addresses at HUT. For this purpose, she registers both her lab machine *(queca@lab.hut.fi)* and her office machine *(queca@office.hut.fi).*

Last time Queca was at HUT she set up her lab computer to automatically forward calls back to her Nokia address. Forgetting about this, she restarts her client in the lab with the same configuration. This forgetfulness will produce an error later on.

In Figure 8b Queca's friend working for Ericsson (*jose@ericsson.se)*, Jose, decides to call her. He thinks that she is in Nokia so he tries to reach her there. He places a call to *queca@nokia.com.* Using DNS the caller resolves nokia.com to the address of the Nokia SIP server, which receives the call request. The Nokia SIP server checks its registration and policy databases. Attending to the Queca's service registration the SIP server forwards the call request to *queca@hut.fi*. When the request arrives to the HUT SIP server, this one does a policy and database lookup determining that there are two different addresses for Queca at HUT: *queca@lab.hut.fi* and *queca@office.hut.fi*. So the HUT SIP server sends the call to both addresses simultaneously.

Following we can see what happens when both phones receive the call (Figure 8c). The office one starts ringing and the other, according to its outdated configuration, forwards the call back to the Nokia SIP server. This server getting back the call determines that an error has occurred and returns an error response to the laboratory machine, which sends an error code as well to the HUT SIP server. This server does not forward the error to Jose because it simultaneously receives an accepted call from the office. In the case that this server had received an error from both places the error would have been forwarded to the server in Nokia.

Meanwhile, Queca answers the phone accepting the incoming call. To do so she sends a positive response back to the HUT SIP server. It immediately informs the Nokia SIP server about the new status, which forwards an OK response to Jose. Then, the communication between Jose and Queca is established (Figure 8d).
The example illustrates a number of salient features of SIP.

❑ It shows how a call request can cause a "hunt" for a user to ensue hopping between multiple servers until the final target of the call is found.

❑ It demonstrates the loop detection features of SIP.

❑ It proves how a server can fork requests to speed up the process of contacting the desired user.

❑ It confirms how SIP servers are used in a call only for the initial transaction.

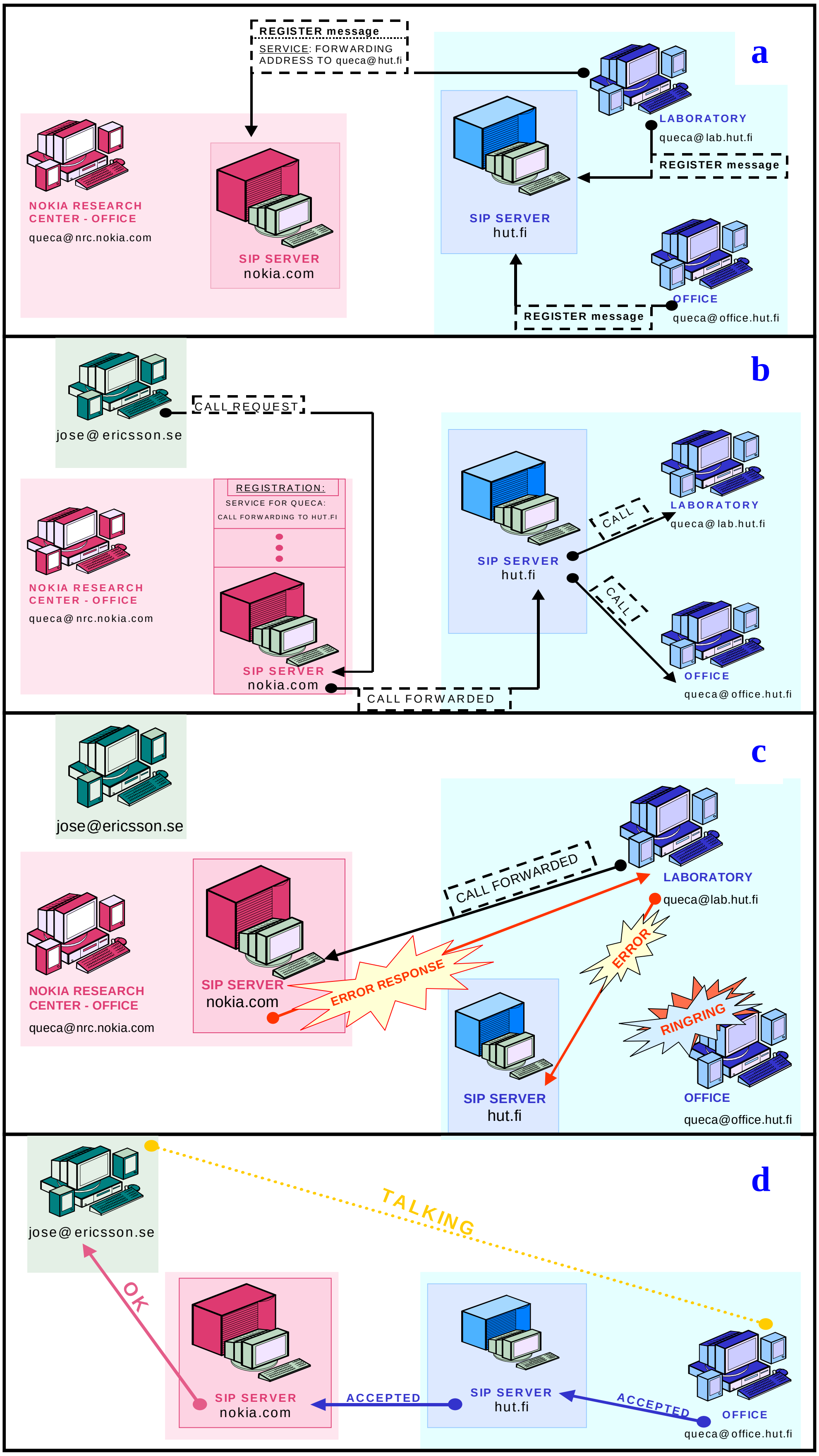


Figure 8. SIP Mobility example

3.5 SIP vs H.323

In order to provide useful services, Internet telephony requires a set of control protocols for connection establishment, capabilities exchange, and conference control [19]. Two standards emerged to meet this need. One is ITU Recommendation H.323, and the other is the IETF Session Initiation Protocol (SIP) presented in this chapter [20]. It is not our proposal to study in detail the H.323 however, here we present briefly the most outstanding characteristics comparing it with the SIP protocol [1].

These two protocols represent very different approaches to the same problem. H.323 embraces the more traditional circuit-switched approach to signaling based on the Q.931 ISDN protocol and earlier H-series recommendations [18]. On the other hand, SIP favors the more lightweight Internet approach based on HTTP by reusing many of its header fields, encoding rules, error codes, and authentication mechanisms.

In both cases, multimedia data will likely be exchanged via RTP, so that the choice of protocol suite does not influence Internet telephony QoS. To conclude this chapter, following we compare the two current protocols in terms of complexity, modularity, transport protocol neutrality, extensibility and services.

3.5.1 COMPLEXITY

H.323 is a rather complex protocol. Its complexity stems from the hundreds of elements that it defines, and from its use of several protocol components. There is no clean separation of these components and many services usually require interaction between several of them to achieve a simple task. Furthermore, H.323 uses a binary representation for its messages based on ASN.1 that usually requires special code-generators to parse.

On the other hand, SIP has only 37 headers each with a small number of values and parameters that contain more information. SIP encodes its messages as text, similar to HTTP and the Real Time Streaming Protocol [25]. This leads to simple parsing and generation, particularly when done with powerful text processing languages such as XML, Perl and so on [37].

3.5.2 MODULARITY

SIP is a quite modular protocol. It encompasses basic call signaling, user location, and registration. Advanced signaling is part of SIP, but within a single extension. QoS, directory accesses, service discovery, session content description and conference control are all orthogonal, and reside in separate protocols. Moreover, the modularity of SIP allows it to be used in conjunction with H.323.

H.323 is less modular than SIP. It defines a vertically integrated protocol suite for a single application. The mix of services provided by the H.323 components encompass capability exchange, conference control, maintenance operations, basic signaling, QoS, registration, and service discovery. Furthermore, these are intertwined within the various sub-protocols within H.323. As they are all integrated in a single protocol, removing any one of these and use a new or separate protocol for this functionality is very difficult.

3.5.3 TRANSPORT PROTOCOL NEUTRALITY

SIP can be carried on either TCP or UDP (Figure 9). If desired, SIP can run directly on top of any protocol offering reliable or unreliable byte stream or datagram services, including ATM AAL5, IPX, X.25, without any changes to the protocol. Unfortunately, H.323 requires the use of a reliable transport protocol.

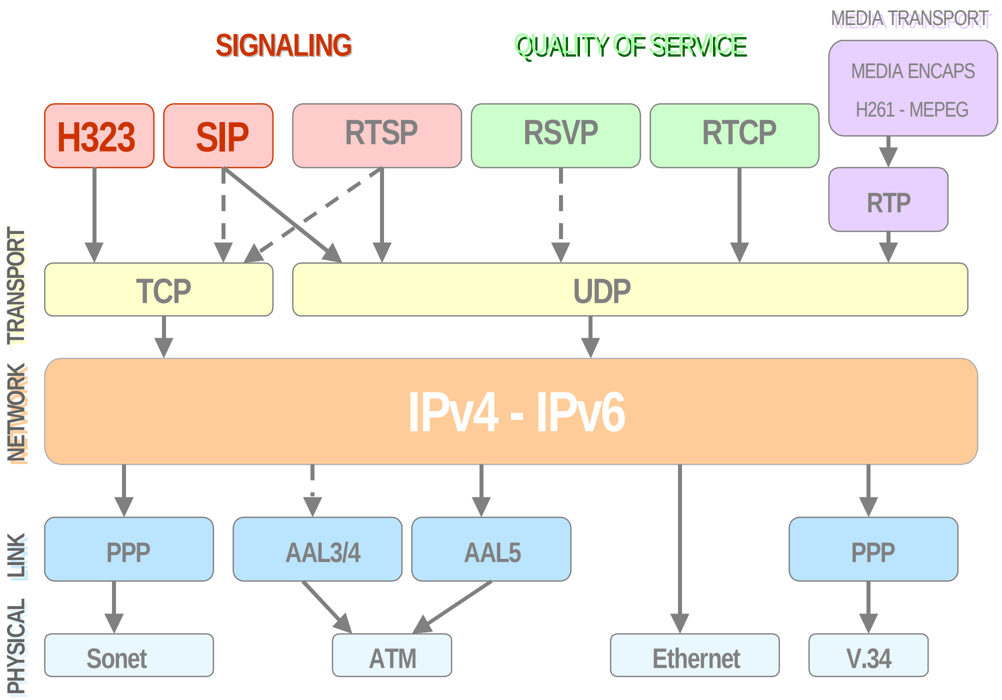


Figure 9. Protocols architecture

3.5.4 EXTENSIBILITY

As the Internet is an open, distributed, and evolving entity, one can expect extensions to IP telephony protocols to be widespread and uncoordinated. Both, SIP and H.323 provide extensibility mechanisms but a critical issue for this are audio and video codecs.

SIP uses the Session Description Protocol to convey the codecs supported by an endpoint in a session. Codecs are identified by string names, so SIP can work with any codec.

In H.323, each codec must be centrally registered and standardized. Currently, only ITU developed codecs have codepoints. As many of these carry significant intellectual property, there is no free sub-28.8 kb/s codec, which can be used in an H.323 system. In general, this presents a significant barrier to entry for small users.

3.5.5 SERVICES

A comparison in these dimensions is somewhat difficult, as SIP and H.323 offer equivalent services [7]. Both protocols support call control services and provide capabilities exchange services. In this latter regard, H.323 has a richer set of functionality. Terminals can express their ability to perform various encodings based on parameters of the codec, and based on which other codecs are in use. However, most implementations do not require these, and the basic receiver-capability indication supported by SIP seems sufficient.

In the area of personal mobility services, SIP provides rich support for that while H.323's support is more limited [6]. H.323 was not engineered for wide area operation; it has no mechanisms for loop detection, preferences cannot be expressed, and it does not allow a gatekeeper to proxy a request to multiple servers.

Both, H.323 and SIP support multiparty conferences with multicast data distribution. The problem is that H.323 requires a central control point, called MC, for processing all signaling and also other mechanisms depending on the conference size. On the other hand, SIP scales to all different conference sizes, does not require an MC, and conference coordination is fully distributed.

H.323 supports conference control service while SIP does not provide it. SIP relies instead on other protocols for this service (e.g. RTCP).

3.5.6 CONCLUSIONS

We have compared SIP and H.323 in terms of complexity, modularity, transport protocol neutrality, extensibility and services. We have found that SIP provides lower complexity, reasonable modularity, and rich extensibility. Roughly, both protocols provide a similar set of services but there are also some differences in the mechanisms required.

More than one year ago, when SIP seemed to be a powerful but quite unknown protocol, H323 protocol was leading the signaling for Internet telephony. However, all big companies quickly started announcing their support for SIP and creating technical teams to develop applications using SIP. H.323 continues there but SIP is growing up very quickly and more and more advanced features are implemented every month.

Definitely anyone looking to converge large voice and data networks also should ensure that the equipment they buy can be upgraded to support SIP. Essentially, to bring advanced voice support into the data networking age, protocols must be targeted directly at actual user services. Products should easily integrate into a real network with little modification to its underlying infrastructure. The protocols should be easily extensible without breaking the existing implementations or relying on a factious standards body for approval. Based on these criteria SIP looks like the winner in the voice over packet service creation race.

In resume, there is still a place for H.323 but SIP is quickly proliferating and the interest in the protocol is growing day by day.

Chapter 4

The Session Description Protocol

This chapter defines the Session Description Protocol (SDP) [23]. This protocol is intended for describing multimedia sessions for the purposes of session announcement, session invitation, and other forms of multimedia session initiation. The SDP is used to advertise multimedia conferences and communicate the conference addresses and conference tool-specific information necessary for participation.

This protocol is a product of the Multiparty Multimedia Session Control (MMUSIC) [30] working group of the Internet Engineering Task Force (IETF), RFC 2327 [29].

A general overview and a session example analysis are covered in this chapter. Further information can be found in [23].

4.1 The Session Description Protocol Overview

The Session Description Protocol (SDP) is used to describe a multimedia session within a SIP request. The purpose of this protocol is to convey information about media streams in multimedia sessions allowing people receiving a session description to participate in a session.

SDP is purely a format for session description. It does not incorporate a transport protocol, and it is intended to use different transport protocols as appropriate including the Session Announcement Protocol [26], Session Initiation Protocol [1], Real-Time Streaming Protocol [25], electronic mail using the MIME extensions, and the Hypertext Transport Protocol.

In general, SDP must convey sufficient information to enable joining a session (with the possible exception of encryption keys) and to announce the resources to be used to non-participants that may need to know.

An SDP description includes the following information:

- Session name and purpose.
- Time/s the session is active.
- The media comprising the session.
- Information on how to receive those media such as addresses, ports and formats.
- Additional information like the bandwidth to be used in the conference and contact information for the person responsible of the session established.

SDP is intended to be of general purpose so that it can be used for a wider range of network environments and applications than just multicast session directories. However, it is not intended to support negotiation of session content or media encodings.

4.1.1 MEDIA INFORMATION

In general, there is a lot of information carried in a session description and the media information is essential for our implementation. It includes:

- The type of media (video, audio, etc).
- The transport protocol (RTP/UDP/IP, H.320, etc).
- The format of the media (H.261 video, MPEG video, etc).

For an IP multicast session, the following information is also conveyed:

- Multicast address for media.
- Transport Port for media.

This address and port are the destination address and destination port of the multicast stream, whether being sent, received, or both.

For an IP unicast session, the following information is conveyed:

- Remote address for media.
- Transport port for contact address.

The semantics of the address and the port depend on the media and transport protocol defined. By default, this is the remote address and remote port to which data is sent, and the remote address and local port on which to receive data. However, some media may define to use these to establish a control channel for the actual media flow.

4.1.2 AN SDP DESCRIPTION STRUCTURE

An SDP session description consists of a number of lines of text of the form: **<*type*>=<*value*>** Where <*type*> is always exactly one character and is case-significant, and <*value*> is a structured text string whose format depends on <*type*>. It also will be case-significant unless a specific field defines otherwise.

Whitespace is not permitted either side of the '=' sign. In general <*value*> is either a number of fields delimited by a single space character or a free format string.

An announcement consists of a session-level section followed by zero or more media-level sections. The session-level part starts with a '*v=*' line and continues to the first media-level section. The media description starts with a '*m=*' line and continues to the next media description or end of the whole session description.

The permitted type fields in a SDP session are shown in Table 4. Some lines are obligatory and others are optional (marked with a "*") but all of them must exactly appear in the order given here.

SESSION DESCRIPTION

TYPE	DESCRIPTION
v	*PROTOCOL VERSION*
o	*OWNER/CREATOR AND SESSION IDENTIFIER*
s	*SESSION NAME*
i*	*SESSION INFORMATION*
u*	*URI OF DESCRIPTION*
e*	*EMAIL ADDRESS*
p*	*PHONE NUMBER*
c*	*CONNECTION INFORMATION (**)*
b*	*BANDWIDTH INFORMATION*
z*	*TIME ZONE ADJUSTMENTS*
k*	*ENCRYPTION KEY*
a*	*ZERO OR MORE SESSION ATTRIBUTE LINES*

(**) NOT REQUIRED IF INCLUDED IN ALL MEDIA

TIME DESCRIPTION

TYPE	DESCRIPTION
t	*TIME THE SESSION IS ACTIVE*
r*	*ZERO OR MORE REPEAT TIMES*

MEDIA DESCRIPTION

TYPE	DESCRIPTION
m	*MEDIA AND TRANSPORT ADDRESS*
i*	*MEDIA TITLE*
c*	*CONNECTION INFORMATION (**)*
b*	*BANDWIDTH INFORMATION*
k*	*ENCRYPTION KEY*
a*	*ZERO OR MORE SESSION ATTRIBUTE LINES*

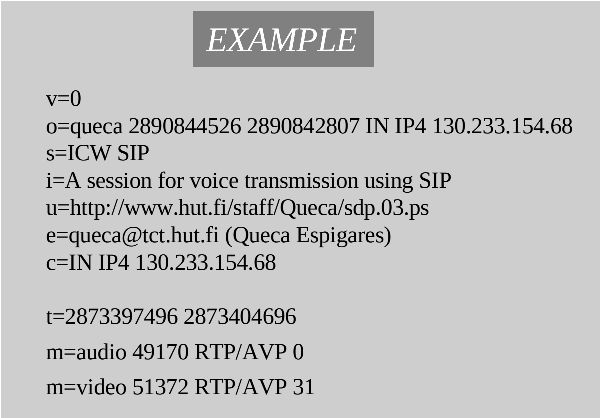
(**) OPTIONAL IF INCLUDED AT SESSION-LEVEL

Table 4. SDP descriptions permitted type fields

SDP parsers must ignore any announcement that contains a <type> letter that it does not understand. The attribute mechanism is the primary means for extending SDP and tailoring it to particular applications or media. Some attributes have a defined meaning but others may be added on an application, media or session specific basis. A session directory must ignore any attribute that it does not recognize.

4.2 A SDP Session Example

In order to clarify the concepts presented previously, next we will introduce an SDP session example, see Figure 10. Some fields are mandatory and others are suggested depending on the application. The most relevant ones such as protocol version, origin, session name, time, connection data and media announcement are described in detail.



EXAMPLE

```
v=0
o=queca 2890844526 2890842807 IN IP4 130.233.154.68
s=ICW SIP
i=A session for voice transmission using SIP
u=http://www.hut.fi/staff/Queca/sdp.03.ps
e=queca@tct.hut.fi (Queca Espigares)
c=IN IP4 130.233.154.68

t=2873397496 2873404696
m=audio 49170 RTP/AVP 0
m=video 51372 RTP/AVP 31
```

Figure 10. SDP Session Example

4.2.1 PROTOCOL VERSION

The ‘v=’ field gives the version of the Session Description Protocol. There is no minor version number.
In the example: *v=0*

4.2.2 ORIGIN

The ‘o=’ field gives the originator of the session (their username and the address of the user's host) plus a session id and session version number.
In the example: *o=queca 2890844526 2890842807 IN IP4 130.233.154.68*

The general field structure is:

o=<username> <session id> <version> <network type> <address type><address>

- *<username>* is the user's login on the originating host, or it is '-' if the originating host does not support the concept of user ids. *<username>* must not contain spaces.

- *<session id>* is a numeric string such that the tuple of <username>, <session id>, <network type>, <address type> and <address> form a globally unique identifier for the session.

  - The method of *<session id>* allocation is up to the creating tool, but it has been suggested that a Network Time Protocol (NTP) timestamp be used to ensure uniqueness [31].

  - *<version>* is a version number for this announcement. It is needed for proxy announcements to detect which of several announcements for the same session is the most recent. Again its usage is up to the creating tool, so long as *<version>* is increased when a modification is made to the session data. For this reason, it is recommended that an NTP timestamp is used.

  - *<network type>* is a text string giving the type of network. Initially 'IN' is defined to have the meaning 'Internet'.

  - *<address type>* is a text string giving the type of the address that follows. Initially 'IP4' and 'IP6' are defined.

  - *<address>* is the globally unique address of the machine from which the session was created. For an address type of IP4, this is either the fully qualified domain name of the machine, or the dotted-decimal representation of the IP version 4 address of the machine. For an address type of IP6, this is either the fully qualified domain name of the machine, or the compressed textual representation of the IP version 6 address of the machine. For both IP4 and IP6, the fully qualified domain name is the form that SHOULD be given unless this is unavailable, in which case the globally unique address may be substituted. A local IP address MUST NOT be used in any context where the SDP description might leave the scope in which the address is meaningful.

In general, the 'o=' field serves as a globally unique identifier for this version of this session description, and the subfields excepting the version taken together identify the session irrespective of any modifications.

4.2.3 SESSION NAME

The 's=' field is the session name. There must be one and only one 's=' field per session description, and it must contain ISO 10646 characters.
In the example: *s=ICW SIP*

4.2.4 CONNECTION DATA

The 'c=' field contains connection data. In the example: *c=IN IP4 130.233.154.68*

The general field structure is:

c=<network type> <address type> <connection address>

A session announcement must contain one ‘c=’ field in each media description or a ‘c=’ field at the session-level. It may contain a session-level ‘c=’ field and one additional ‘c=’ field per media description, in which case the per-media values override the session-level settings for the relevant media.

- ❑ *<network type>* is a text string giving the type of network. Initially ‘IN’ is defined to have the meaning ‘Internet’.
- ❑ *<address type>* allows SDP to be used for sessions that are not IP based. Currently only IP4 is defined.
- ❑ *<connection address>* is the third sub-field. For IP4 addresses, the connection address is defined as follows:

  If the session is not multicast, then the connection address contains the fully qualified domain name or the unicast IP address of the expected data source or data relay or data sink as determined by additional attribute fields.

4.2.5 TIME SESSION IS ACTIVE

In the ‘t=’ field, the first and second sub-fields give the start and stop times for the conference respectively. These values are the decimal representation of Network Time Protocol (NTP) time values in seconds [31].

In the example: *t=2873397496 2873404696*

The general time field structure is:

t=<start time> <stop time>

If the stop-time is set to zero, then the session is not limited, though it will not become active until after the start-time. If the start-time is also zero, the session is regarded as permanent.

4.2.6 MEDIA ANNOUNCEMENTS

A session description may contain a number of media descriptions. Each media description starts with an ‘m=’ field, and is terminated by either the next ‘m=’ field or by the end of the session description.

In the example: *m=audio 49170 RTP/AVP 0*
m=video 51372 RTP/AVP 31
m=application 32416 udp wb

The general media field structure is:

m=<media> <port> <transport> <fmt list>

A media field also has several sub-fields:

1. The first sub-field is the media type. Currently defined media are included in Table 5.

MEDIA TYPE
Application
Audio
Control
Data
Video
New emerging communication modalities

Table 5. Media type

The difference between *application* and *data* is that the former is a media flow such as whiteboard information, and the latter is bulk-data transfer such as multicasting of program executables which will not typically be displayed to the user. *Control* is used to specify an additional conference control channel for the session.

2. The second sub-field is the transport port to which the media stream will be sent. The meaning of the transport port depends on the network being used as specified in the relevant ‘c=’ field and on the transport protocol defined in the third sub-field. Other ports used by the media application, such as the RTCP port, should be derived algorithmically from the base media port.

   For transports based on UDP, the value should be in the range 1024 to 65535 inclusive. For RTP, only the even ports are used for data and the corresponding one-higher odd port is used for RTCP [24]. For example, ‘*m=video 49170/2 RTP/AVP 31*’ would specify that ports 49170 and 49171 form one RTP/RTCP pair and 49172 and 49173 form the second RTP/RTCP pair. RTP/AVP is the transport protocol and 31 is the format.

3. The third sub-field is the transport protocol. The transport protocol values are dependent on the address-type field in the ‘c=’ fields. Thus a ‘c=’ field of IP4 defines that the transport protocol runs over IP4. For IP4, it is normally expected that most media traffic will be carried as RTP over UDP. The following transport protocols are preliminarily defined:

   - RTP/AVP - the IETF Real-time Transport Protocol using the Audio/Video Profile carried over UDP.
   - UDP - User Datagram Protocol.

   The main reason to specify the transport-protocol in addition to the media format is that the same standard media formats may be carried over different transport protocols even when the network protocol is the same. In addition, relays and monitoring tools that are transport-protocol-specific but format-independent are possible.

4. The fourth and subsequent sub-fields are media formats. For audio and video it will normally be a media payload type as defined in the RTP/AVP.
   When a list of payload formats is given, this implies that all of these formats may be used in the session, but the first of these formats is the default format for the session.

   For media whose transport protocol is RTP, SDP can be used to provide a dynamic binding of media encoding to RTP payload type.

   An example of a dynamic payload type is 16 bit linear encoded stereo audio sampled at 16KHz. If we wish to use dynamic RTP/AVP payload type 98 for such a stream, additional information is required to decode it:

   ```
   m=video 49232 RTP/AVP 98
   a=rtpmap:98 L16/16000/2
   ```

   The general form of a rtpmap attribute is:

   ```
   a=rtpmap:<payloadtype><encodingname>/<clockrate>[/<encodingparameters>]
   ```

   For audio streams, <*encoding parameters*> may specify the number of audio channels. This parameter may be omitted if the number of channels is one provided no additional parameters are needed. For video streams, no encoding parameters are currently specified.

Chapter 5

Columbia University SIP Server[1]

In this chapter, we describe the SIP Server from Columbia University [58]. It is called *sipd* and we have used it for the development and testing of the Internet Call Waiting service that will be introduced in the next chapter.

The code runs on Solaris, FreeBSD 3.0, Linux, Windows NT/95/98, and other Unix platforms. It is written in C to perform efficiently behavior in all kinds of possible architectures. We have chosen the Unix version to be run in one of the workstations of the Telecommunications Technology Laboratory at HUT in Espoo, Finland [57].

An implementation overview, analysis of the main functions and additional libraries required are covered in this chapter.

[1] Copyright 1998-1999 by Columbia University; all rights reserved Sipd is subject to licensing. Permission to use, copy, modify, and distribute this software and its documentation for not-for-profit research and educational purposes and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that the copyright notice and warranty disclaimer appear in supporting documentation, and that the names of the copyright holders or any of their entities not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. Use of this software in whole or in parts for commercial advantage and by for-profit organizations requires a license.

The copyright holders disclaim all warranties with regard to this software, including all implied warranties of merchantability and fitness. In no event shall the copyright holders be liable for any special, indirect or consequential damages or any damages whatsoever resulting from loss of use, data or profits, whether in an action of contract, negligence or other tortuous action, arising out of or in connection with the use or performance of this software.

5.1 Columbia University implementation

Sipd is a SIP redirect, proxy and registration server that provides name mapping, user location and scripting services (Figure 11) [1]. It can use external routines to do the actual work of resolving aliases (including group names), mapping names and locating users. It also allows users to register their current location with the server. Users can be registered at multiple locations [3]. Each user can register a script in any scripting language that will be executed when receiving a call [55].

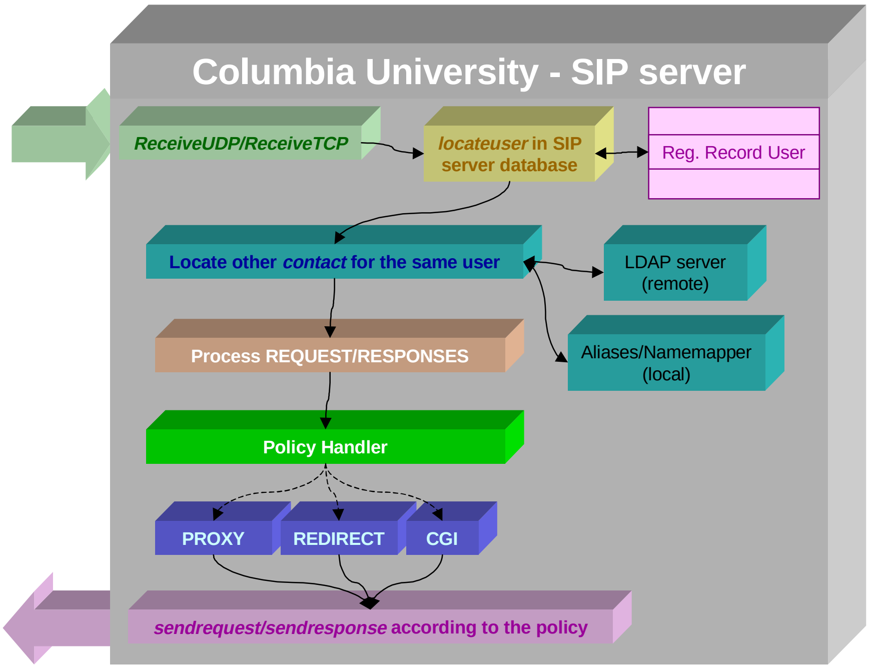

Columbia University - SIP server
ReceiveUDP/ReceiveTCP
locateuser in SIP server database
Reg. Record User
Locate other contact for the same user
LDAP server (remote)
Aliases/Namemapper (local)
Process REQUEST/RESPONSES
Policy Handler
PROXY
REDIRECT
CGI
sendrequest/sendresponse according to the policy

Figure 11. Columbia University SIP server

The server currently understands the following requests:

- ACK
- BYE
- CANCEL
- INVITE (can be authenticated)
- OPTIONS
- REGISTER (can be authenticated).

If the user is not registered or cannot be found using the dynamic user location program, the server returns a client error: 'Temporarily Not Available' code number 480.

5.1.1 SYNOPSIS

Running the server can be done in different ways depending on the arguments added to the *main* program.

sipd [-*s configfile*][-*p port*][-*v*][-*X*]

- [-*s ServerRoot*]
  It sets *ServerRoot* where the server expects to find the configuration file.
- [-*v*]
  It makes the server print out debugging information to stdout.

- [-*p port*]
  It overrides the *Port* configuration variable.
- [-*X*]
  It runs in single-process mode for internal debugging purposes only.

5.1.2 CONFIGURATION

The server is configured through the file *sipd.conf*. The server looks for the file by default in the directory where the server was started. Alternatively, the location of the configuration file can be specified via the -*s* command line parameter. The file should be in the following format:

- **Aliases**

It is the path name of the name mapping function invoked first, e.g., *aliases*. The server appends the user name (not the whole request URI) to the name given. The program is expected to return on *stdout* zero or more lines, each containing one entry of the form *user*, *user@host* or *user@host.domain*. If the response has more than one line, it is assumed that the name refers to a group rather than an individual. In that case, the server returns a redirection response ‘Multiple Choices’ (300 code number) and the list of responses. If there is one match, it is used as an index into the registration database.

- **AuthUserFile**

The *AuthUserFile* directive sets the name of a textual file containing the list of users and passwords for user authentication. Filename is the path to the user file. If it is not absolute (if it doesn't begin with a slash) then it is treated as relative to the *ServeRoot*. Each line of the user file contains a *username* followed by a colon, followed by the *crypt()* encrypted password. The behavior of multiple occurrences of the same user is undefined. The program automatically converts the file into an in-memory hash table when first used, so that access to passwords is efficient.

- **CGITimeout**

It is the time in seconds that the program waits for CGI script to return an answer. Default is set to 15 seconds [8].

- **DefaultReg**

If a registration contains a contact header that does not have an action parameter, it fills in this action. Default is redirect.

- **DisplayAmbiguous**

This flag determines whether the server will return the contact list in case of ambiguous response (more than one match). Default is false.

- **Domain reg-ex**

A regular expression defining the domains that the server accepts registrations from. If the host part of the To header does not match the regular expression, the server returns a status of ‘Client Not Found’ (404 code number).

- **ErrorLog**

The *ErrorLog* directive sets the name of the file to which the server will log any errors it encounters. If the *logfile* does not begin with a slash (/), then it is assumed to be relative to the server root. If the *logfile* begins with a pipe (|), then it is assumed to be a command to spawn to handle the ErrorLog. The default is *stderr*.

- **Expires**

Registrations expire after the number of seconds given in the *Expires* configuration option. The default is 3600 that means one hour.

- **ForeignDomain**

If the request contains a URI with a domain that does not match the Domain reg-ex, the server can either proxy the request, redirect it (possibly substituting the IP address for the host name) or reject the request.

- **Group**

The Group directive sets the group under which the server will answer requests. In order to use this directive, the stand-alone server must be run initially as root. *Unix-group* is one of:

  - ❑ A group name: refers to the given group by name.
  - ❑ # followed by a group number: refers to a group by its number.

It is recommended to set up a new group specifically for running the server. Some administrators use nobody as a user, but this is not always possible or desirable.

- **HelpResolve**

The server returns redirect responses with numeric (dotted-quad) IP addresses instead of host names, relieving the client from doing name resolution. Defaults to ‘false’.

- **LdapServerName**

It defines the hostname of LDAP server to contact for user lookups [32].

- **LdapPort**

It defines the port number of LDAP server. Defaults to 389.

- **LdapBaseObject**

Specifies the DN of the base object for LDAP search. The search proceeds down the object's subtree.

- **Namemapper**

The path name of the name mapping function invoked second, e.g., namemapper. The program is invoked only if the first one did not return any results. The server appends the user name (not the whole request URI) to the name given. The program is expected to return on stdout zero or more lines each containing one entry of the form 'X user [displayname]', 'X user@host [displayname]' or 'X user@host.domain [displayname]', where 'X' is an arbitrary string and the optional displayname designates the real name of the user. (Namemapper uses the string X to designate the type of match found.) If there is more than one match, the server returns 485 (Ambiguous) and the list of matches in Location headers, with the displayname as a comment. If there is one match, it is used to look up the user in the registration database. If there is no match, the server returns 604 (Does not exist). Thus, even if the username given in the request URI is already a system user name, the namemapper program must return this name.

- **PidFile**

The PidFile directive sets the file to which the server records the process id of the daemon. If the filename does not begin with a slash (/), then it is assumed to be relative to the ServerRoot. It is often useful to be able to send the server a signal, so that it closes and then reopens its ErrorLog and TransferLog, and re-reads its configuration files. This is done by sending a SIGHUP (kill -1) signal to the process id listed in the PidFile.

- **Port portnumber**

The port number the server listens on for both TCP and UDP. Default is 5060.

- **PrivateKey**

It is the key for digest authentication.

- **ProxyName**

The hostname which this machine should put in its Via headers. Default is the canonicalized local hostname. Must be a name, which DNS resolves to this server.

- **ProxyRecursion**

Whether the server should recursively process 3xx-responses containing Contact headers with sip URI's when proxying, or just return 3xx responses to the originator the same way any other failure status would be returned.

- **Registry**

The name of the database file used to record user locations, relative to ServerRoot. Defaults to ‘registry’.

- **RequestLog**

Each RequestLog entry creates one request log file. The first argument gives the name of the file to log to, the second the format to be used. If the name of the file starts with '|', it is assumed to be a program that will receive, via stdin, the log file output.

Table 6 shows the format. It is a string containing literal text and the following special escape sequences, borrowed from the Apache log file configuration. The format can be omitted and defaults to "%h %l %u %t \"%r\" %s %b".

%...a	Remote IP-address
%...{env}e	The contents of the environment variable env.
%...h	Remote host name (currently also the IP address).
%...{header}i	The input (request) header with name header.
%...{header}o	The output (response) header with name header.
%...P	The process ID of the child that serviced the request.
%...r	First line of request.
%...s	Status of response.
%...U	The Request-URI.
%...u	Remote user if authentication is used, ‘-‘ otherwise.
%...{timeformat}t	The current time, in the form given by timeformat, which should be in a form acceptable to strftime(3). The timeformat defaults to the common log file date format, ‘[day/month/year:hour:minute:second zone]’.
%...T	The time taken to serve the request, in seconds, with a resolution of 1/100 second.

Table 6. RequestLog file format

- **ServerName FQDN**

Defines the domain name of the server, e.g., cs.columbia.edu

- **ServerRoot directory**

Defines the location of the server configuration files, typically '.'

- **ThirdPartyReg true | false**

Whether the server should allow third parties to register each other, or insist that users only register themselves.

- **TimeOut seconds**

The TimeOut directive defines the amount of time sipd will wait for completion of a batch of proxied requests. A batch is all requests with similar quality (q) values. Default is 30 seconds.

- **User unix-userid**

The User directive sets the userid as which the server will answer requests. In order to use this directive, the standalone server must be run initially as root. Unix-userid is one of:

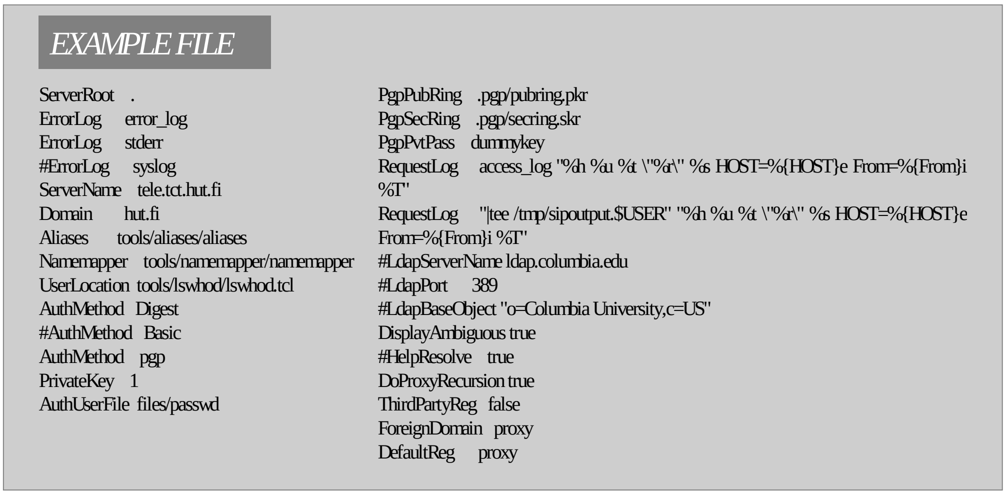
- ❑ A username refers to the given user by name.
- ❑ # Followed by a user number refers to a user by their number.

The user should have no privileges which result in it being able to access files which are not intended to be visible to the outside world, and similarly, the user should not be able to execute code which is not meant for sipd requests. It is recommended that you set up a new user and group specifically for running the server. Some administers use user nobody, but this is not always possible or desirable.

If the server is started as a non-root user, it will fail to change to the lesser-privileged user, and will instead continue to run as that original user.

- **UserLocation**

It is the path name of the user location program such as lswhod. The program is invoked with the user name and is expected to return on stdout zero or more lines each containing an entry. Figure 12 gives an example of a configuration file presenting all the directives described previously.



EXAMPLE FILE

```
ServerRoot    .
ErrorLog      error_log
ErrorLog      stderr
#ErrorLog     syslog
ServerName    tele.tct.hut.fi
Domain        hut.fi
Aliases       tools/aliases/aliases
Namemapper    tools/namemapper/namemapper
UserLocation  tools/lswhod/lswhod.tcl
AuthMethod    Digest
#AuthMethod   Basic
AuthMethod    pgp
PrivateKey    1
AuthUserFile  files/passwd

PgpPubRing    .pgp/pubring.pkr
PgpSecRing    .pgp/secring.skr
PgpPvtPass    dummykey
RequestLog    access_log "%h %u %t \"%r\" %s HOST=%{HOST}e From=%{From}i
%T"
RequestLog    "|tee /tmp/sipoutput.$USER" "%h %u %t \"%r\" %s HOST=%{HOST}e
From=%{From}i %T"
#LdapServerName ldap.columbia.edu
#LdapPort     389
#LdapBaseObject "o=Columbia University,c=US"
DisplayAmbiguous true
#HelpResolve  true
DoProxyRecursion true
ThirdPartyReg  false
ForeignDomain  proxy
DefaultReg     proxy
```

Figure 12. Configuration file example.

5.2 Sipd functions

The program starts by reading the configuration file using the routines in *config.c*. Then it opens a UDP and TCP socket to receive requests. The thread *ReceiveTCP*, with each new connection obtaining a new thread running *ReceiveTCPRequests* handles TCP connections. The thread *ReceiveUDP* handles UDP requests (unicast or multicast).

Each request is read and, once completely read, handed to *RequestProcess()*. That routine parses the SIP-specific headers and then invokes the request-specific function. The ACK request simply sets the ACK flag in the appropriate request. Each INVITE request creates a new thread, while REGISTER requests are handled within the same thread. To understand the behavior we need to know which are the *sipd* functions and what are they used for.

base64.c is a group of utility functions that convert to and from base-64 strings, as used in Authorization headers.

config.c reads server configuration files, where each line is of the form '*parameter: value*'.

dstring.c contains dynamic string utility functions.

error.c contains error handling and debugging utility functions.

host2ip.c converts host name to IP address.

http.c parses generic HTTP headers like Authenticate, Authorization, Content-Length, Content-Type, Date, Server and User-Agent.

invite.c handles the INVITE request.

locate_user.c checks if the user is currently logged in or not by calling the *lswhod.pl*. If the user is currently logged in it gives the machine names, otherwise, it calls aliases to find the user login name. If it can not find in aliases, then it calls to the namemapper. Next, it returns the result if the locating was successful and a null string otherwise. Since the namemapper takes time, it sends back a 100 message to the client.

mime.c handles MIME datatype mapping.

parser.c parses RFC822-style requests and responses.

register.c handles REGISTER request.

request.c. Requests are stored in an internal data structure so that the repeated ones can simply be answered by the current status of the request. A pointer to the Request structure is created the head of list of incoming request.

- ❑ *RequestCreate* initilizes and allocates memory for a request.

- ❑ *RequestSearch* searches the list of requests to check if the request with unique Call-ID+SeqNumber exists already or not. Then it returns true or false accordingly. If the request exists and it is an ACK request, it deletes the request entry. Otherwise it does nothing.

- ❑ *RequestFree* frees the given SIP request.

- ❑ *RequestProcess* obtains the version, method, protocol, seq number, call ID and other header fields of the SIP request.

response.c generates provisional and final SIP responses.

- ❑ *void ResponseInit(void)* initializes a response.

- ❑ *int Response(Request *r, int status, char *reason, DString *location, char *content_type, char *body, int content_length)* sends the message back to a client including the From, To, Via, Call-ID, Cseq and Location headers. If the user is not found, no Location header will be returned.

sip.c parses SIP-specific headers like Call-ID, CSeq, Expires, From, Location, Subject, To, Via.

sip-rs.c is the main program.

strdupn.c creates duplicate of string, with its own memory.

tcp.c handles TCP connections.

udp.c handles UDP requests.

uri.c parses URIs into their components.

5.3 Sipd additional libraries

Sipd uses two libraries that are not built with it: CLC and gdbm. The latter was developed by a free software foundation called GNU [56] while the CLC is part of a C Collection libraries from Columbia University [50]. The following is a brief idea about their main purposes.

Gdbm is a library of database functions that uses extensible hashing and works similar to the standard UNIX database management functions.

The basic use of *gdbm* is to store key-data pairs in a data file. Each key must be unique and each key is paired with only one data item. The keys can not be directly accessed in sorted order. The key-data pairs are stored in a *gdbm* disk file, called a *gdbm* database. An application must open a *gdbm* database to be able to manipulate the keys and data contained in the database. Gdbm allows an application to have multiple databases open at the same time.

On the other hand, CLC is the acronym for my C Libraries Collection, a group of general purpose C libraries: dict, fsma, misc, pq, pset, sio, str, timer and xlog. Next we will give a short description of what fsma and dict do. Those are just the ones that *sipd* needs from CLC.

Dict: support for various types of data structures, including double linked lists, hash tables and binary search trees (which can also be balanced using the red-black balancing algorithms). The data structures support operations for insertion/deletion/location of an object, as well as operations for finding the minimum/maximum object and for enumeration of all objects.

Fsma: support for quick memory allocation-deallocation of fixed size objects.

A basic build of *sipd* on our laboratory machines looks like this:

```
./configure --with –gdbm
=/home/queca/SIP_1/sip_code/libreria/gdbm/lib --with-
clc=/home/queca/SIP_1/sip_code/libreria/clc/libs/lib

make depend

make
```

This builds everything to start running the SIP CU server.

Chapter 6

Scripting new services: The Call Processing Language

The Call Processing Language (CPL) is a language that can be used to describe and control Internet telephony services [34]. This mechanism is not tied to any particular signaling architecture or protocol (H.323, SIP).

In this chapter, we outline the tools used to define the Internet Call Waiting service and compare the XML with other languages such as HTML [33].

The Internet Call Waiting (ICW) service is studied in detail in Chapter 7 and implemented using a CPL. For that purpose, we create a script describing the service that later is uploaded to a SIP server. So, it is the user who easily writes the service desired. The SIP server execution stores the service description for further performance.

6.1 The Call Processing Language

The Call Processing Language (CPL) is designed to be implementable on either network servers or user agent servers (UAS) [35]. It is 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.

Syntactically, CPL scripts are represented by XML documents, so parsing them is easy and many parsers for them are publicly available [41]. The structure of the language maps closely to its behavior, so an editor can understand any valid script, even ones written by hand. The language is also designed so that a server can easily confirm scripts validity at the time they are delivered to it, rather that discovering them while a call is being processed.

6.1.1 THE CPL NETWORK MODEL

In this model, an Internet telephony network contains two types of components: end systems and signaling servers. End systems are devices, which originate and/or receive signaling information and media. An end system can originate, accept, reject a call, or forward incoming calls. Signaling servers are devices, which relay or control signaling information. In SIP, they are proxy servers, redirect servers, or registrars [1].

Signaling servers can perform three types of actions on call setup information. They can forward it on to one or more other network or end systems, returning one of the responses received (*proxy it*). They can also return a response informing the sending system of a different address to which it should send the request (*redirect it*). Finally, they can inform the sending system that the setup request could not be completed (*reject it*).

When an end system places a call, the call establishment request can proceed by a variety of routes through components of the network. To begin with, the originating end system must decide where to send its requests. There could be, for example, two possibilities: the originator may be configured so that all its requests go to a single local server; or it may resolve the destination address to locate a remote signaling server or end system to which it can send the request directly.

Once the request arrives at a signaling server, that server uses its user location database, its local policy, DNS resolution, or other methods, to determine the next signaling server or end system to which the request should be sent. A request may pass through any number of signaling servers: from zero (in the case when end systems communicate directly) to every server on the network.

6.1.2 SCRIPTS: WHAT, WHICH, WHERE AND HOW

In this section, we answer some questions to clarify scripts behavior:

1) **What** does a script do?
   Specifically, a script replaces the user location functionality of a signaling server. Signaling server typically maintains a database of locations where a user can be reached; it makes its proxy, redirect, and rejection decisions based on the contents of that database. A CPL script replaces this basic database lookup functionality; it takes the registration information, the specifics of a call request, and other external information it wants to reference (e.g. services requested), and chooses the signaling actions to perform [34].

   Abstractly, a script can be considered as a list of condition/action pairs; if some attribute of the registration, request, and external information matches a given condition, then the corresponding action is taken.

2) **Where** can users have scripts?
   Users can have CPL scripts on any network server which their call establishment requests pass through and with which they have a trust relationship. Scripts

would typically perform different functions, related to the role of the server on which they reside.

3) **Which** script is executed in the server? And when?
CPL scripts are usually associated with a particular Internet telephony address. When a call establishment request arrives at a signaling server which is a CPL server, that server associates the source and destination addresses specified in the request with its database of CPL scripts; if one matches, the corresponding script is executed.

   Once the script has been executed, if it has chosen to perform a proxy action, a new Internet telephony address will result as the destination of that proxying. Once this has occurred, the server again checks its database of scripts to see if any of them are associated with the new address; if one is, that script is also executed.

   In general, in an Internet telephony network, an address will denote one of two things: either a user, or a device. A user address refers to a particular individual, for example *sip:queca@example.com*, regardless of where that user actually is or what kind of device she is using. A device address, by contrast, refers to a particular physical device, such as *sip:x26063@phones.example.com*.

6.2 XML: the bridge between SGML and HTML

XML is the language chosen to implement the ICW service. In this section we briefly describe and compare it with other also popular languages. XML characteristics such as extensibility, easily writeable and parseable by both humans and machines, easily implementable and independent of underlying signaling details are commented [39].

6.2.1 NOTES OF XML STORY

As far back as the sixties, IBM scientists were working on a Generalized Markup Language (GML) for describing documents and their formatting. In 1986 the International Standards Organization (ISO) adopted a version of this standard called Standard Generalized Markup Language (SGML-ISO 8879). SGML offers a highly sophisticated system for marking up documents so that their appearance is independent of specific software applications. It is big, powerful, filled with options, and well suited for large organizations that need exacting document standards.

But early in the game, it became apparent that SGML's sophistication made the language quite unsuitable for quick and easy applications. For that, we needed a simplified markup system, one at which practically anyone could quickly gain proficiency. Enter HyperText Markup Language (HTML), which is little more than one specific SGML document type, or Document Type Definition (DTD) [38]. It is easy to learn and to implement, but the problem with HTML, however, was that it quickly proved to be too simple. As soon as authors started demanding multimedia and design capabilities, the language started experiencing severe growing pains. Undoubtedly, the main problem is that HTML is not extensible while SGML is.

The question, then, is how to capture SGML's extensibility without retaining the complexity. In other words, the issue is how to bridge the gap between SGML and HTML. The answer is Extensible Markup Language, better known as XML [38].

6.2.2 WHY XML?

Nowadays, other known programming languages such as PERL, TCL, HTML, SGML and C/C++ could have been chosen [54]. So, why XML?

XML is more than a markup language it is a metalanguage. This means that XML is a language that allows you to describe languages.

XML lets developers to set standards defining the information that should appear in a document, and in what sequence. XML makes it possible to define the content of a document separately from its formatting, making it easy to reuse that content in other applications or for other presentation environments.

XML provides a basic syntax that can be used to share information between different kinds of computers, different applications, and different organizations without needing to pass through many layers of conversion.

XML provides a simple format that is flexible enough to accommodate diverse needs. Even developers performing tasks on different types of applications with different interfaces and different data structures can share XML formats and tools for parsing those formats into data structures that the applications can use. XML offers many advantages, including:

- **Simplicity**. XML documents are built upon a core set of basic nested structures. While the structures themselves can grow complex as layers and layers of detail are added, the mechanisms underlying those structures require very little implementation effort, from either authors or developers. Furthermore, XML rigid set of rules helps make documents more readable to both humans and machines.

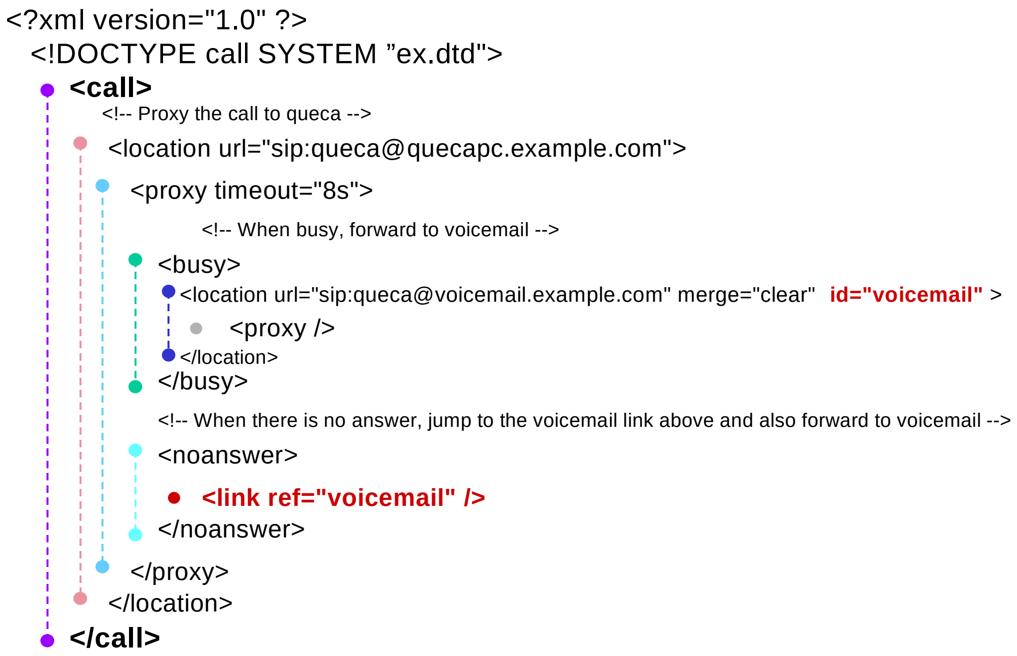
- **Extensibility**. XML is extensible in two senses. First, it allows developers to create their own Document Type Definition (DTD) [38], effectively creating 'extensible' tag sets that can be used for multiple applications. Second, XML itself is being extended with several additional standards that add styles (e.g. XSL), linking, and referencing ability to the core XML set of capabilities.

- **Interoperability**. XML can be used on a wide variety of platforms and interpreted with a wide variety of tools. Because the document structures behave consistently, parsers that interpret them can be built at relatively low cost in any of a number of languages such as C++, C, JavaScript, Tcl, and Python [51].

- **Openness**. XML documents are considerably open and anyone can parse a well-formed XML document.

- **Applications.** XML can be used in a couple of different ways. One is for data interchange between humans and machines, such as from a Web server to a user's browser. The other is for data exchange between applications, or from machine to machine.

Linking possibilities could be also included in advantages list. To illustrate this, Figure 13 presents a script that rings a call at a standard location and, if the recipient is not available there, forwards the call to a voicemail server instead. Since we want the same action to occur on busy as on no answer, we define a link on one node. This allows other nodes to reference that link rather than repeat parts of the script.



```
<?xml version="1.0" ?>
  <!DOCTYPE call SYSTEM ”ex.dtd">
   <call>
     <!-- Proxy the call to queca -->
     <location url="sip:queca@quecapc.example.com">
       <proxy timeout="8s">
              <!-- When busy, forward to voicemail -->
         <busy>
          <location url="sip:queca@voicemail.example.com" merge="clear"  id="voicemail" >
              <proxy />
          </location>
         </busy>
         <!-- When there is no answer, jump to the voicemail link above and also forward to voicemail -->
         <noanswer>
             <link ref="voicemail" />
         </noanswer>
       </proxy>
     </location>
   </call>
```

Figure 13. XML example

Document Type Definition (DTD) is a well-known self-describing and structured information required at the beginning of the validated scripts. The DTD identifies the root element of the document and may contain additional declarations. All XML documents must have a single root element that contains all of the content of the document. Additional declarations may come from an external DTD, called the external subset, or be included directly in the document, the internal subset, or both [38]. In Annex A we present a complex XML example that references an external DTD.

Writing in XML seems to be quite easy but also different technologies can be followed in this area: Document Type Definitions and XML schemas. Although, neither are strictly required for XML development, both DTDs and XML Schemas are important parts of the XML toolbox. DTDs have been around for over twenty years as a part of SGML, while XML Schemas are relative newcomers. Though they use very different syntax and take different approaches to the task of describing document structures, both mechanisms definitely occupy the same turf. The W3C seems to be grooming XML Schemas as a replacement for DTDs, but it is not yet clear how quickly the transition will be made. DTDs are here-and-now, while XML Schemas, in large part, are just *coming*. Brother languages such as Perl, Tcl and so on could also wake up the attention of people describing data.

Chapter 7

The Internet Call Waiting Service

The Internet Call Waiting Service (ICW) enables online users with a single analog telephone line to monitor and manage incoming phone calls from a personal computer while they were using the same line for Internet access or attending another call.

In this chapter, first we explain how the service works and we conclude analyzing step by step the project structure paying attention to all the components involved in: the XML parser, the UAS/UAC and the CPL method [42][35].

7.1 Defining the service

The idea of the ICW implementation is based on a graphical oriented way to notify the new call. ICW service will pop-up a dialog box advising the callee of a new incoming call. For that reason we decided to design the window using bright colors and attractive figures. We also made our own logotype for the program called "IPtele". The logotype including snowflakes gets the idea from the country where we developed the project, Finland (Figure 14).

Figure 14. IPtele logotype

The service works as follows. Firstly, the user has to send the registration information to the server. That information contains the XML script defining the ICW service, which will be stored in the SIP server to manage future calls [3]. Figure 15 shows the main IPtele window where the user can choose between: opening two different boxes "REGISTRATION" and "SET UP CALL" or exiting via "EXIT" button.

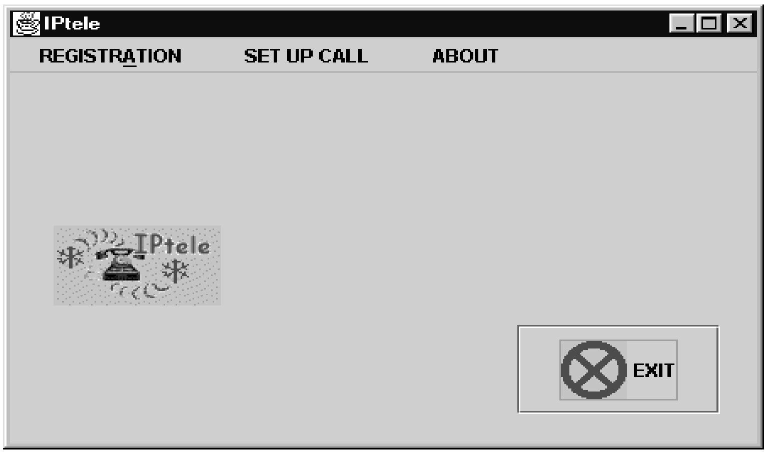


Figure 15. IPtele window

For registering, the user selects "REGISTRATION" radio buttons box (Figure 16). Thereafter he/she chooses a service (call forwarding not supported) and then goes to "INSERT REG INFO". After selecting that button a new window appears on the screen to insert his/her name and domain (Figure 17). The user can also return to the main IPtele window using "CANCEL" button in the "REGISTRATION" window.

As an example, to register the user "*queca@pc2.tct.hut.fi*" we would have to write:

- "NAME" = *queca*
- "DOMAIN" = *pc2.tct.hut.fi*

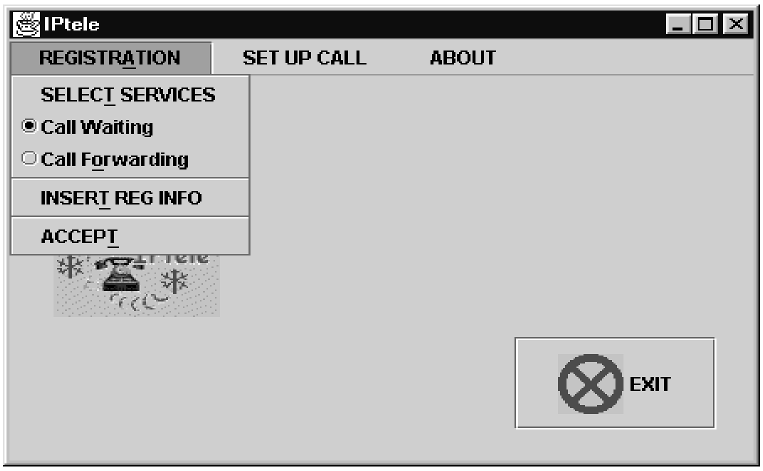


Figure 16. Selecting services window

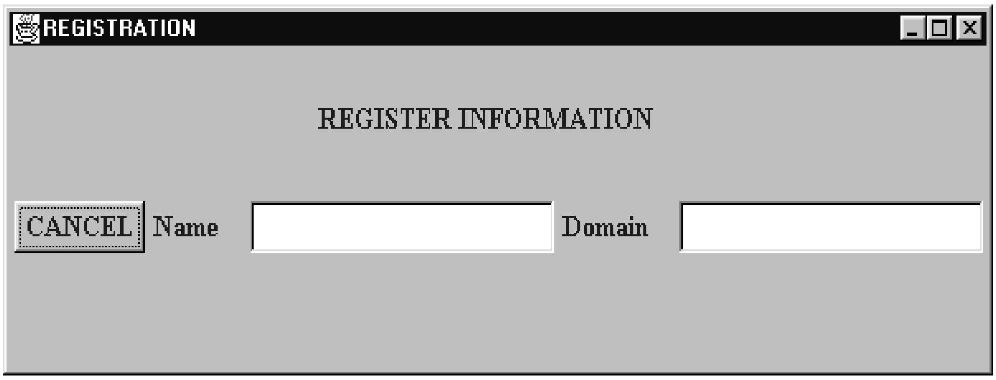


Figure 17. Registering window

When the name and domain are written, the user has to press ENTER button on the keyboard. After that, it is necessary to select "ACCEPT" to complete the registration.

If the user wishes so, he/she can also make a call. Opening the "SET UP CALL" he/she must select "INSERT THE ADDRESS"(Figure 18). After that a new window has to be filled in. This window is required to have the callee information to create the INVITE request to be sent to the SIP server.

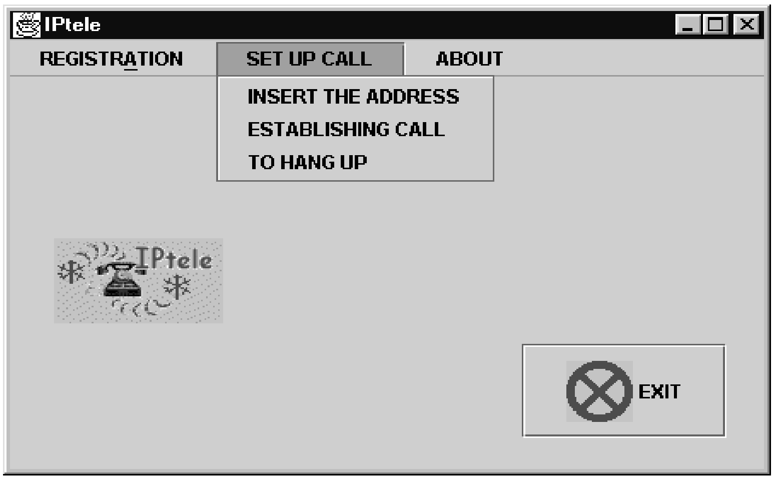


Figure 18. Setting up call

Figure 19 shows the "CALLEE INFORMATION" window where again the name and the domain have to be written. Now these two fields are information from the callee so they are different than the ones inserted in the registration.

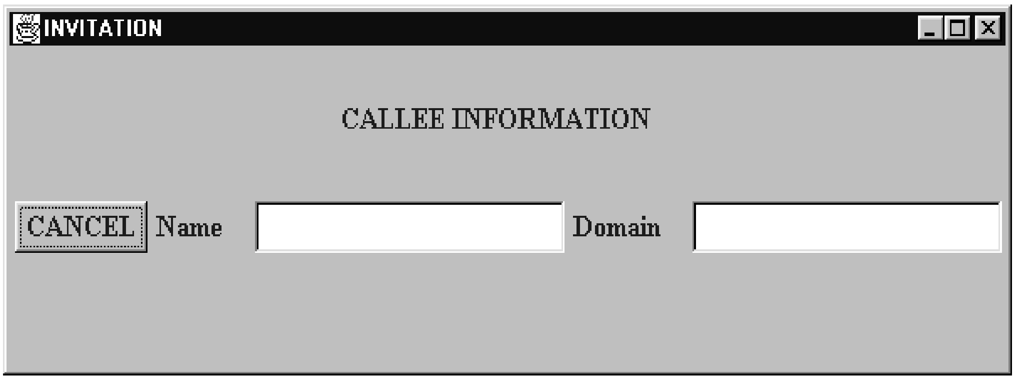
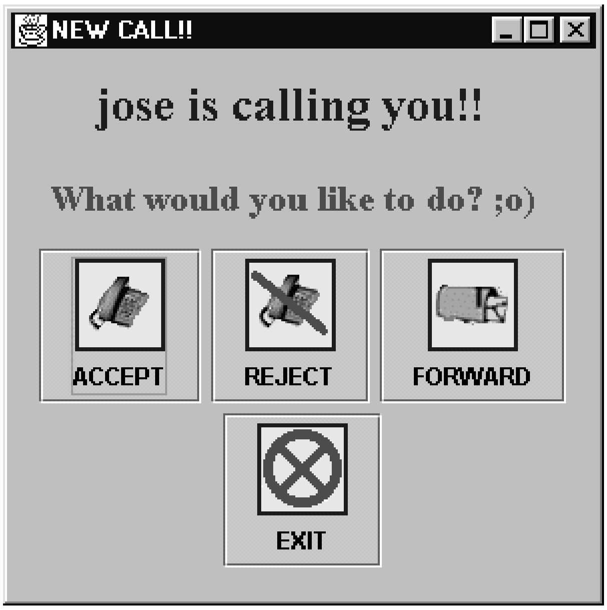


Figure 19. Callee information window

To follow the example, if the user *queca* wishes to call the user "*mary@pc64.tct.hut.fi*" she would have to write:

- "NAME" = *mary*
- "DOMAIN" = *pc64.tct.hut.fi*

When the name and domain are written, the user has to press ENTER button on the keyboard. The invitation is made after pressing "ESTABLISHING CALL". If the callee's (*Mary)* line is not busy and she answers the call then they can start talking. While they are talking, *Jose* (a new user) calls *queca*. At that moment, as *queca* is registered for the ICW service, a "NEW CALL!!" window appears on her screen (Figure 20). Thanks to that new window she does not miss *Jose*'s call if she wishes so.



NEW CALL!!

jose is calling you!!

What would you like to do? ;o)

Figure 20. New incoming call window

In the "NEW CALL!!" window we have designed four different buttons:

- "ACCEPT": selecting this button the user (*queca)* accepts the incoming call (from *Jose* in the example case). She decides to finish the communication with *Mary* and starts talking with *Jose* using the headphones on her PC.

- "REJECT": selecting this button the user (*queca*) rejects the incoming call (from *Jose* in the example case). Then *Jose* gets a message "BUSY" and he can try later on.

- "FORWARD": this program does not support this option. It consists in forwarding the incoming call to a voicemail machine. The caller (*Jose)* can leave a message if he wishes so.

- "EXIT": this button is just to close the "NEW CALL!!" window.

The messages transferred between the IPtele client and the SIP server in a situation like the one presented above are depicted in the following control panel (Figure 21). In this case, *UserA* is talking with *UserB* when *UserC* sends an INVITE request to *UserB*. We consider that UserB was previously registered with the ICW service. *UserB* can accept or reject *User'sC* invitation and messages involved in both cases are included in the figure.

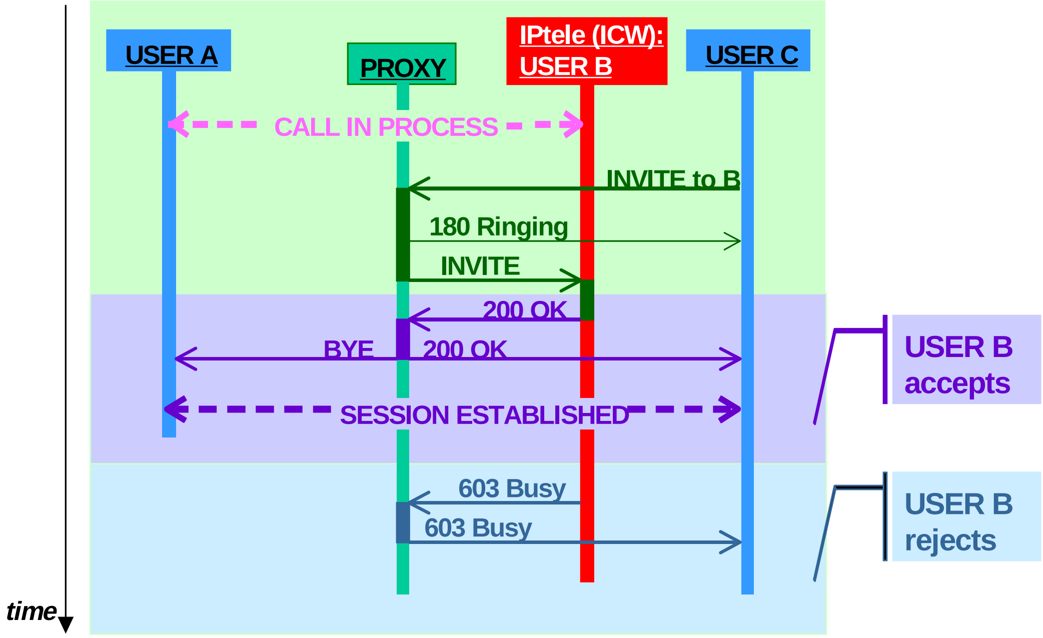


Figure 21. Control panel

7.2 The project structure

In the previous section we presented the Internet Call Waiting service (ICW) to give a general idea of the aim of this project, the service. Now, in this section we describe step by step all the relevant details and tools used to achieve our implementation.

Figure 22 helps to understand the structure of the project. Boxes in red represent the modules developed in Helsinki University.

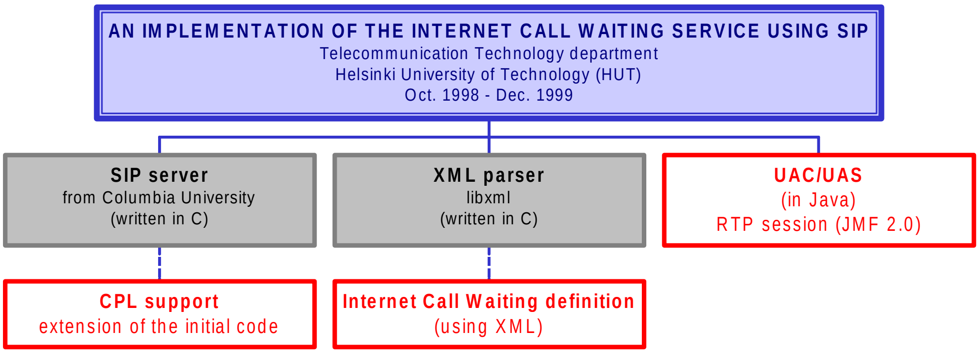

AN IMPLEMENTATION OF THE INTERNET CALL WAITING SERVICE USING SIP
Telecommunication Technology department
Helsinki University of Technology (HUT)
Oct. 1998 - Dec. 1999
SIP server
from Columbia University
(written in C)
XML parser
libxml
(written in C)
UAC/UAS
(in Java)
RTP session (JMF 2.0)
CPL support
extension of the initial code
Internet Call Waiting definition
(using XML)

Figure 22. Project organization chart

7.2.1 MAIN LEGACY ITEMS

The main SIP page [2] was the initial point to start with before trying to develop anything. Understanding the behavior of the protocol was essential to be able to work on it. We also studied the Session Description Protocol [23] for media requirements.

The first “SIP Bake-off “ (New York, Apr. 1999) gave us the opportunity to contact with other implementors and get the license for using the Columbia SIP server. Installing the server in *tele* with the external libraries that it requires (clc, gdbm), was not an easy task.

The next step was writing a JAVA program implementing the User Agent Client (UAC) and the User Agent Server (UAS). The final JAVA code has around 2200 lines. At this point it was also necessary to study and analyze the Java Media Framework Module (JMF 2.0) to insert an RTP session within the code.

This was in the client-server side. Jumping to the service we decided to use the Call Processing Language (CPL) [34][[35] method to implement the service. For that purpose we had to find a suitable parser (libxml) to be embedded in the SIP server to manage the service. This part of the project also required learning XML to write the document describing the Internet Call Waiting service. Installing and adding the parser to the server (SIP server extension) was done very carefully to have all the libraries well fixed. Figure 23 shows in red the extension of the Columbia University SIP server. This figure can be compared with Figure 11 depicted in Chapter 5.

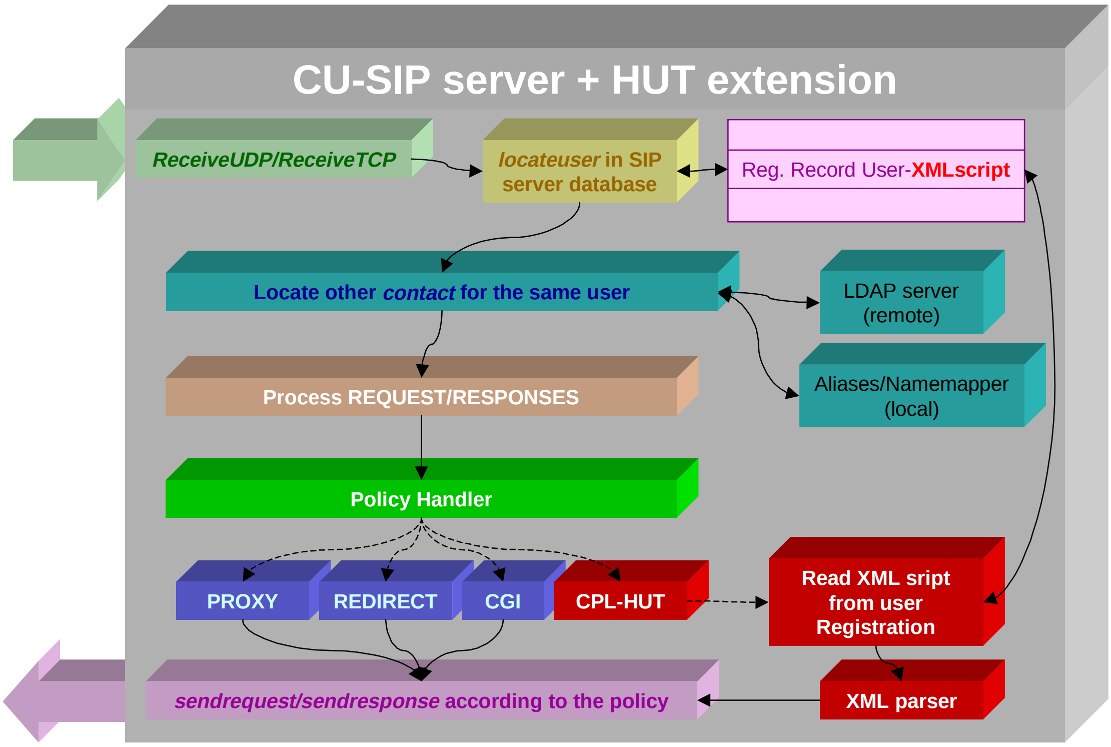

CU-SIP server + HUT extension
ReceiveUDP/ReceiveTCP
locateuser in SIP server database
Reg. Record User-XMLscript
Locate other contact for the same user
LDAP server (remote)
Aliases/Namemapper (local)
Process REQUEST/RESPONSES
Policy Handler
PROXY
REDIRECT
CGI
CPL-HUT
Read XML sript from user Registration
XML parser
sendrequest/sendresponse according to the policy

Figure 23. Software panel: CU-SIP server and HUT extension

As the SIP server was previously presented in chapter 5, we start introducing the XML parser and the CPL method to follow with the JAVA program structure.

7.2.2 THE XML PARSER AND THE CALL PROCESSING LANGUAGE

Nowadays, the popularity of XML has by far exceeded expectations and many books [36], papers[39] and also free parsers for XML documents are publicly available [41]. You can find parsers written in C, C++, JAVA, with modules in Tcl, Perl, Python and so on. There are so many parsers available that it was a big deal to decide the one we were going to use in our project [41].

As the SIP server code is totally written in C, it was better to choose an XML parser also written in C. Studying all the possibilities (Expat, LT XML, etc.) we finally decided to work with the XML Library called *libxml* built by Daniel Veillard [44].

Libxml is integrated in GNU Network Object Model Environment (GNOME) [46] which is part of the GNU project of the Open Source Initiative (OSI) organization [45].

Libxml understands the structure of XML Document Type Definitions (DTDs) [38] and can validate documents against them. While developed and mainly used by its author under Linux, it is a fairly portable library under different platforms. We have used it under Sun Solaris system.

The concepts of valid and well-formed documents, and validating and non-validating parsers are defined in the W3C recommendation XML 1.0. To clarify the concepts, we briefly present the differences. For further information about XML you can go back to chapter 6 or check the REC-xml-19980210 [38].

❑ Valid and well-formed XML files.

  The DTD system is only one method of creating XML documents. As a result, there are two types of XML document, those with DTDs and those without them. Those with DTDs that conform to the SGML standard are called valid files. Documents that exclude DTDs must be well formed; that is, they must conform to a specific set of standards predefined. Valid files must be well formed too.

❑ Validating and non-validating parsers.

  All XML parsers check to see if a document is well formed, but only validating parsers check to see if a document is valid. It means that only validating parsers can understand DTDs structures of data.

To invoke the parser the first thing to do is to read an XML document. The parser accepts to parse both memory mapped documents or direct files. The functions are defined in "parser.h":

❑ *xmlDocPtr xmlParseMemory(char *buffer, int size);*

  parse a zero terminated string containing the document.

❑ *xmlDocPtr xmlParseFile(const char *filename);*

  parse an XML document contained in a file.

This returns a pointer to the document element or NULL in case of failure.

The way of traversing the tree to explore the complete XML document consists in jumping from one node to the others using the *xmlNodePtr*. Depending on the branch taken the final *text* output is different. There are multiple possibilities, following we present some examples.

Document->root element->childs->childs

Document->root element ->childs->next->childs->childs

Document->root element->childs->next->next->childs

Functions to modify the tree are also available.

- ❑ *xmlAttrPtr xmlSetProp(xmlNodePtr node, const xmlChar *name, const xmlChar *value);*

  This function sets or changes an attribute carried by an *element* node.

- ❑ *const xmlChar *xmlGetProp(xmlNodePtr node, const xmlChar *name);*

  This function returns a pointer to the property content.

Two functions are used to read and write the text associated with elements:

- ❑ *xmlNodePtr xmlStringGetNodeList(xmlDocPtr doc, const xmlChar *value);*

  This function takes an "external" string and converts it to one text node or possibly to a list of entity and text nodes. All non-predefined entity references will be stored internally as an entity node; hence the result of the function may not be a single node.

- ❑ *xmlChar *xmlNodeListGetString(xmlDocPtr doc, xmlNodePtr list, int inLine);*

  This is the dual function, which generates a new string containing the content of the text and entity nodes

The validation is still *work in progress*. Until a real validation interface is defined the way to do it is to define and set the *xmlDoValidityCheckingDefaultValue* external variable to 1.

To understand the parser behavior we analyze the tree out built by the parser. The parser returns a tree built during the document analysis. The value returned is an *xmlDocPtr*. This structure contains information like the file name, the document type, and a root pointer, which is the root of the document (or more exactly the first child under the root, which is the document). The tree is made of *xmlNodes*, chained in double linked lists of siblings and with *childs-parent* relationship. An *xmlNode* can also carry properties (a series of *xmlAttr* structures). An attribute may have a value, which is a list of TEXT node.

The Call Processing Language (CPL) definition was introduced in chapter 6 including details of how to manage scripts and how to write CPL documents using XML.

Now we present the application called *hut-cpl* based on the CPL [34][35].

Embedding the XML parser (libxml) in the SIP server code, we provide the server XML document management and somehow *service understanding*.

The Internet Call Waiting is defined by the following structure (icw.xml), which will be explored by the XML parser.

```
<?xml version="1.0"?>
<call Type="ICW">
  <proxy>
    <icw>
      <forward>
        <link ref="voicemail"/>
      </forward>
      <success>
        <location url="queca@pc2.tct.hut.fi"/>
      </success>
      <reject>The user is Busy now</reject>
    </icw>
    <busy/>
    <noanswer/>
    <failure/>
  </proxy>
  <response status="busy"/>
</call>
```

When the script is received in the SIP server, the XML parser translates all the information about the user requirements to data structures. Those will be used to perform the right decision when another request is coming.

Figure 24 depicts the steps followed by the parser to explore the *icw.xml* file. Looking at the top of the structure, the pointer *xmlDocPtr* stores the memory address where the xml script is deposited. The functions *xmlParseFile* and *xmlParseMemory* return this pointer. From this point the parser starts analysing the script. The *xmlNodePtr* guides to the memory addresses where the next nodes of the tree are stored. The *doc->root* function gets the root node of the tree, in our case the <*call*> tag. From the root, the parser can follow different branches according to the *next* and *childs* functions. The latter goes down one level in the *parents-children* hierarchy. Hence, from <*call*> the parser could take two possible *childs* nodes: <*proxy*> and <*response*>. Following the same criteria, the <*proxy*> node presents four possible *childs* branches *(<icw>, <busy>, <noanswer>* and <*failure*>) and, at the same time, the <icw> has three *childs* more *(<forward>, <success>* and <*reject*>). The structure continues like this with two *childs* more; < *link*> for <*forward*> and <*location*> for <*success*>.

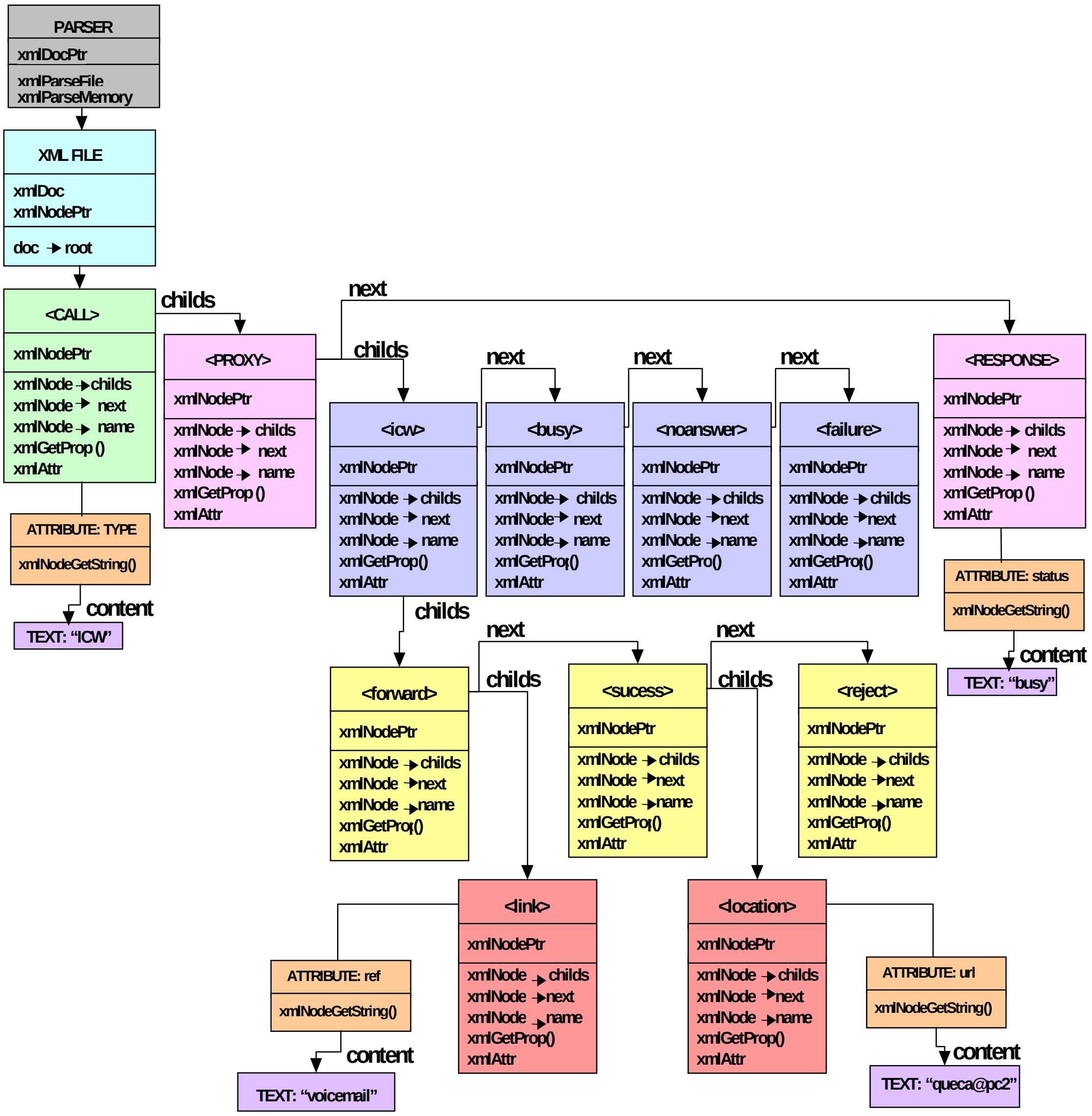


Figure 24. A tree built by the XML parser

Furthermore, *xmlGetProp()* function gives the content of the each node property, e.g. the content of *TYPE* in *<call>* would be *"ICW"*. On the other hand, *xmlNodeGetString()* function returns the attribute information between the same node tags, e.g. in *<reject>* node would be *The user is Busy now.*

7.2.3 UAS/UAC

We implemented the User Agent Client (UAC) and the User Agent Server (UAS) in JAVA [51].

The main program features are:

- ❑ handling windows and managing user actions.
- ❑ creating a window to act as an interface with the user, *IPtele window*.
- ❑ creating the SIP User Agent, *SIP_UA myagent.*

- Finally, giving the control to both modules (*window*, *myagent*).

To clarify its behavior Figure 25 depicts the main program diagram.

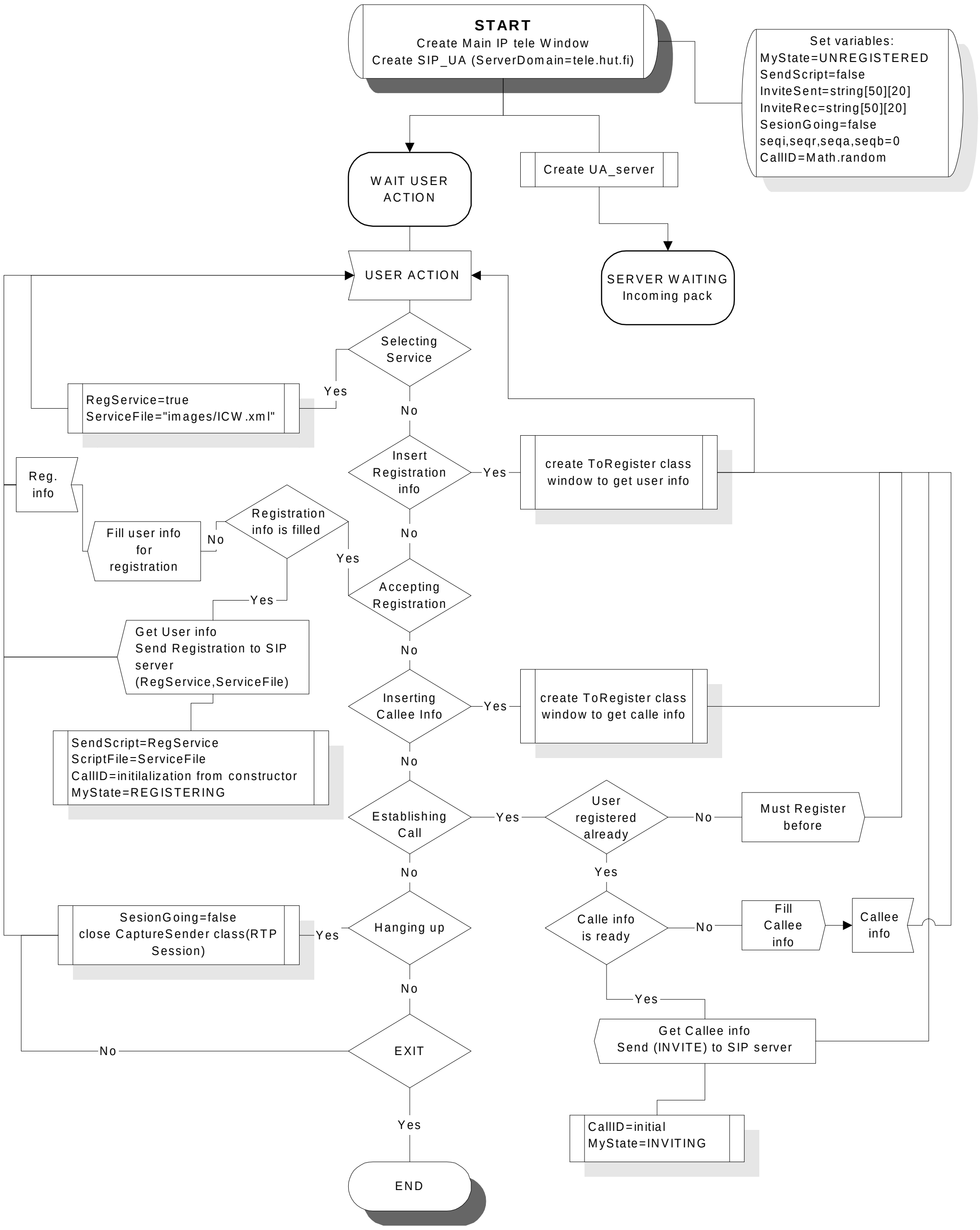


Figure 25. Main program diagram

The SIP_UA created by the main program presents four modules:

- **User Agent Client (UAC) module.**

This class is invoked when the SIP_UA is created. The UAC constructor initializes all the variables for incoming/outgoing messages and the sockets needed for the communication. This class also implements different methods to send requests and responses on behalf user actions and upon variable values or status.

Three modules are created inside the UAC: the UAS (*FormPack*), the *send()* function and the *CaptureSender()* class.

The UAS is created through an independent thread to handle all the incoming messages. Thus, the UAS runs in standalone process only to be aware of any incoming packet. Here *FormPack* function is initialized to create the messages according to the SIP syntax. *FormPack* requires a well-formed SIP packet and the SDP declaration message based on user resources [23]. Following, the code lines for the INVITE method and the consequent INVITE request are included as examples [4]. (An SDP fields overview is included in chapter 4).

```
public String FormPack(String Type){

         ............. ............

   if (Type=="invite"){

         info = new StringBuffer("INVITE
         sip:"+callee+"@"+DestDomain+" SIP/2.0\n");

         Via=new String("Via: SIP/2.0/UDP
         "+myhostaddress+":5060\n");

         From=new String("From:
         sip:"+myname+"@"+mydomain+"\n");

         To=new String("To:
         sip:"+callee+"@"+DestDomain+"\n");
               CallID=new String("Call-ID:
         "+Callid+"@"+mydomain+"\n");

         seqi++;

         CSeq=new String("CSeq: "+seqi+" INVITE\n");

         Contact=new String("Contact:
         <sip:"+myname+"@"+mydomain+":5060;transport=udp>\n"
         );

         ContentType=new String("Content-Type:
         application/sdp\n");

         /* SDP DESCRIPTION */

         StringBuffer sdpinfo= new StringBuffer("v=0\n");

         String o=new String("o="+myname+" 2890844526
         2890842807 IN IP4 "+myhostaddress+"\n");

         String s=new String("s=ICW SIP\n");

         String i=new String("i=A Session for voice
         transmission using SIP\n");

         String e=new
         String("e="+callee+"@"+DestDomain+"\n");

         String p=new String("p=+358 9 4514785 \n");

         String c=new String("c=IN IP4
         "+myhostaddress+"\n");
         String t=new String("t=0 0\n");

         String m=new String("m=audio 50250 RTP/AVP 0\n");

         String a=new String("a=rtpmap:0 PCMU/8000\n\n");

         sdpinfo.append(o).append(s).append(i).append(e).app
         end(p).append(c).append(t).append(m).append(a);

         String sdpto=sdpinfo.toString();

         ContentLength=new String("Content-Length:
         "+sdpto.length()+"\n\n");
         info.append(Via).append(From).append(To).append(Cal
         lID).append(CSeq).append(Contact).append(ContentTyp
         e).append(ContentLength).append(sdpto);
         }............ ............
```

Resulting, for example, the following INVITE request:

```
INVITE sip:queca@pc2.tct.hut.fi SIP/2.0
Via: SIP/2.0/UDP 130.233.154.64:5060
From: sip:jose@pc64.tct.hut.fi
To: sip:queca@pc2.tct.hut.fi
Call-ID: 3062@pc64.tct.hut.fi
CSeq: 1 INVITE
Contact:<sip:jose@pc64.tct.hut.fi:5060;transpo
rt=udp>
Content-Type: application/sdp
Content-Length: 227

v=0
o=jose 2890844526 2890842807 IN IP4
130.233.154.64
s=ICW SIP
i=A Session for voice transmission using SIP
e=queca@pc2.tct.hut.fi
p=+358 9 4514785
c=IN IP4 130.233.154.64
t=0 0
m=audio 50250 RTP/AVP 0
a=rtpmap:0 PCMU/8000
```

The *send()* function is created in the UAC constructor and is used to establish a link with the SIP server for sending the packets.

The last module is the *CaptureSender()* class. After the message transactions a call is established and thereafter the UAC initializes this class. This function is based on JAVA JMF 2.0 module and its responsibility is helping the RTP module to establish an RTP session for media transmission [51][24].

- **<u>User Agent Server (UAS) module.</u>**

This class waits for incoming packets from the SIP server and also invokes the parser to manage the scripts.

Figure 26 shows the UAS diagram.

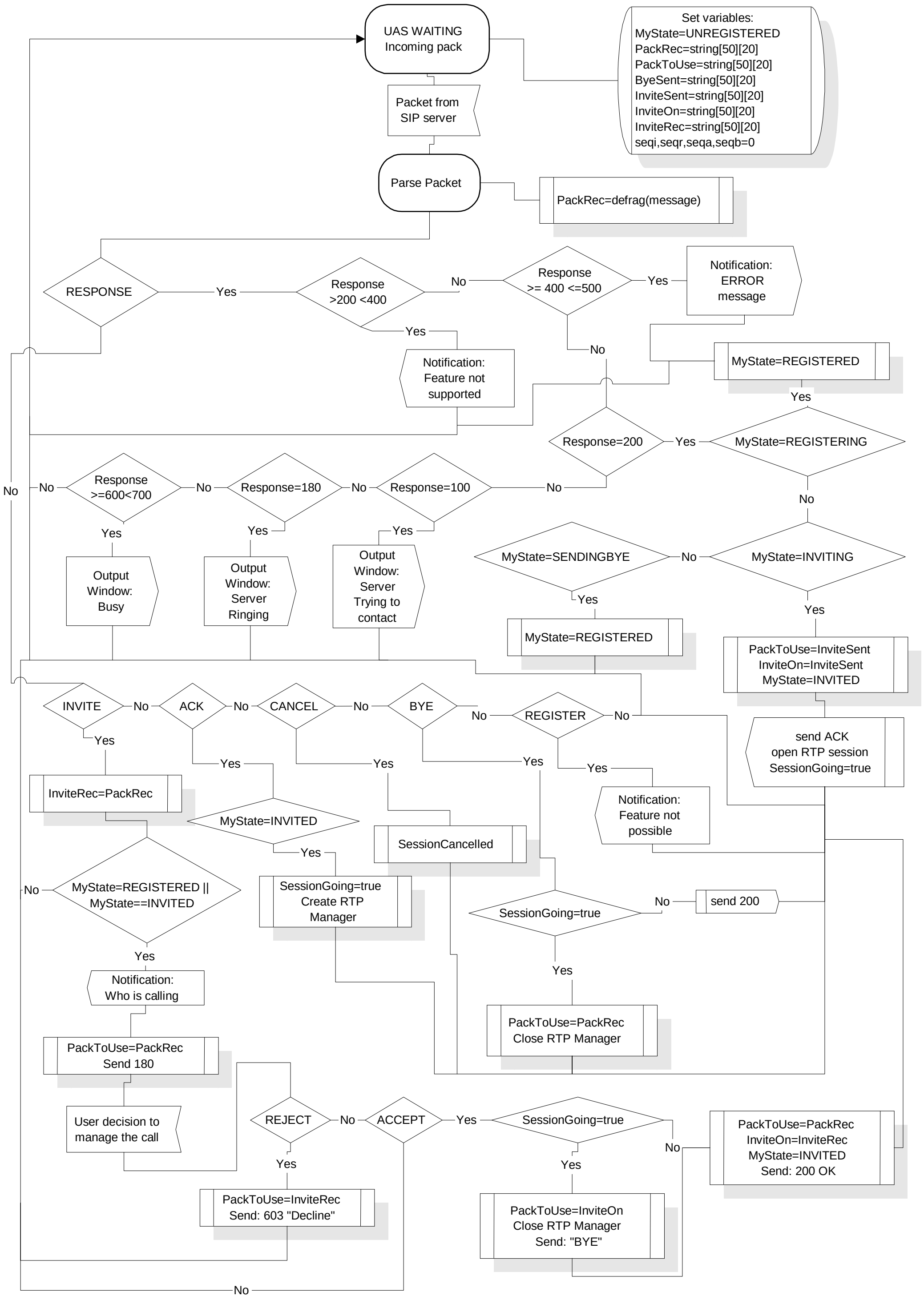


Figure 26. UAS module

- **<u>SIP *parse* module.</u>**

This class is created at the UAC initialization. Its structure includes SIP messages syntax to be able to strip out the incoming messages sent by the UAS. It also sets the environment variables properly for posterior UA action.

- **Real Time Protocol (RTP) module.**

This module is implemented using JAVA JMF 2.0. To establish the RTP session, it captures the participant IP addresses sent by the UAC and the media type required according to the SDP declaration included in the messages [24].

After the initial SIP message transactions, this module creates a media session between the two addresses previously captured. The media requirements are read from the environment variables filled from the SDP packets received.

Finally, when the call is established this module is re-called. Afterwards, the voice is taken from the microphone and sent through this session to the remote user. This function is also aware of the RTP incoming packets carrying the voice to be played running the users' sound card.

Chapter 8

Conclusions and future work

IP telephony is becoming the new leader in the telephony world. Day by day the number of companies interested in IP telephony is continuously increasing and new attractive services are in progress. Users demand new facilities and services that must be provided by the companies using the best tools to achieve their objectives.

Over the last few months, the Internet Call Waiting service is becoming increasingly popular. The service (implementated by Nortel networks) is already available in some areas of the United States but it uses the H.323 protocol for signaling instead of the Session Initiation Protocol (SIP). Currently, we do not know about any other implementation of ICW using the Call Processing Language (CPL) and SIP. We believe that our implementation is the first one of its characteristics and we are very proud of working in such a *hot* telecommunications sector.

The development of this project has been hard but at the same time quite interesting. Different knowledge was required to achieve our goals like C programming, JAVA, XML, and protocols behavior in general. Building the implementation required writing quite a lot of Java code but also of lecturing and consulting.

The project started at the end of November 1998. Thus, it has taken around one year to be completed, including the writing of this document that has not been an easy task. Most of the project time was spent on learning new tools, and writing the JAVA program that creates the SIP client (UAC/UAS). Furthermore, the XML gave us work in the sense that it is a very new language and there is information about it but sometimes the information is confusing and incomplete.

Today, the Internet Call Waiting is the only service that Iptele program supports. Anyway, defining new services is quite easy using the payload of SIP REGISTER messages, and their responses as the media to transport the scripts to registration servers alongside the user's registration. The end user would be the one who decides which kind of service he/she would like to have. Therewith, writing a new XML file defining the service and with a few extensions in the JAVA program different new services would be also supported by Iptele and managed by the SIP server.

Something can always be left for future time. In our case new features can be supported by the IPtele but also the SIP server can have new extensions to manage the possible incoming future services. Actually, the SIP server from Columbia University is a programmable proxy, redirect and registrar server. To achieve new features maybe different headers or responses could be generated.

Defining the service can be done using different languages and structures. As it is explained in chapter 6, we use XML for writing the service files. Right now there are too many options for describing the data, but in the future there will be even more. People study new fashionable languages to find the easiest and more understandable way to describe the data. New syntax, new schemes, new tools, and new preferences are always appearing and developers must choose the best one according to users' demands.

By now nearly everyone in the software industry is aware of the whirlwind of attention being focused on XML. There can be little doubt that XML has garnered its fair share of attention from the press and seen a lot of action at this year's conferences and trade shows. But the question remains: Can XML make it possible to create new systems for data management and organization without many of the incompatibilities and complexities that plagued older systems?

The XML specification itself is a technical, complex document that lays out the form of XML in minute detail but does little to shed light on its practical potential. Truly, XML is important because it represents an open, standards-based language and allows developers to go their own ways on markup elements. It is an established syntax with freely available parsers and is easy to parse and write for both humans and computers. In fact, XML can be part of many imaginative scenarios involving business-to-business interaction, application integration, and so on, in which XML plays the role of the trusty supporting actor: acting as the universal data format for heterogeneous applications, databases, and legacy systems.

References

SIP and related articles

[1] M. Handley, H. Schulzrinne, E. Schooler and J. Rosenberg, "SIP: Session Initiation Protocol," *Request for Comments 2543*, *Internet Engineering Task Force*, Mar. 1999.

[2] H. Schulzrinne, "Session Initiation Protocol," online <URL: http://www.cs.columbia. edu/~hgs/sip/>.

[3] J. Lennox and H. Schulzrinne, "Transporting User Control Information in SIP REGISTER Payloads," *Internet Draft*, *Internet Engineering Task Force*, Feb. 1999. *Work in progress.*

[4] J. Rosenberg, J. Lennox and H. Schulzrinne, "Programming Internet Telephony Services," *IEEE Internet Computing Magazine*, May/Jun. 1999.

[5] J. Rosenberg and H. Schulzrinne, "The Session Initiation Protocol: Providing Advanced Telephony Services across the Internet," *Bell Labs Technical Journal*, Vol. 3, No. 4, Oct/Dec. 1998, pp. 144-160.

[6] E. Wedlund and H. Schulzrinne, "Mobility support using SIP," *Second ACM/IEEE International Conference on Wireless and Mobile Multimedia (WoWMoM'99)*, Aug. 1999, Seattle, Washington.

[7] J. Lennox, H. Schulzrinne and T. La Porta, "Implementing Intelligent Network Services with the Session Initiation Protocol," *Columbia University Computer Science Technical Report CUCS-002-99*, Jan. 1999.

[8] J. Lennox, J. Rosenberg and H. Schulzrinne, "Common Gateway Interface for SIP," *Internet Draft, Internet Engineering Task Force*, May 1999. *Work in progress.*

[9] A. Johnston, S. Donovan, R. Sparks, C. Cunningham and K. Summers, "SIP Telephony Call Flow Examples," *Internet draft*, *Internet Engineering Task Force*, Oct.1999. *Work in progress.*

[10] R. Sparks, C. Cunningham, A. Johnston, S. Donovan and K. Summers, "SIP Telephony Service Examples With Call Flows," *Internet draft*, *Internet Engineering Task Force,* Oct. 1999. *Work in progress.*

[11] J. Rosenberg and I. Slepchin, "Siphone: a Session Initiation Protocol GUI based user agent," online <URL: http://www.bell-labs.com/project/sip/>.

[12] J. Pulver, "Notes from Fall '99 Voice on the Net", online <URL: http://www.pulver.com/reports/> Atlanta, Georgia, Sep. 1999.

[13] S. Breidenbach, "Got the urge to converge? " *Network World*, online <URL: http://www.nwfusion.com/buzz99/buzzcon.html>, Sep. 1999.

[14] D. Willis "The Future is SIP", *Network Computing*, online <URL: http://www.networkcomputing.com/1019/1019colwillis.html>, Sep. 1999.

[15] V. Vittore, "SIP gains momentum," *Telephony*, p.14, Aug. 23, 1999.

[16] G. Camarillo, "IP Telephony Gateways," *Master Thesis*, Ericsson Telecom AB, Department of Teleinformatics, KTH, Nov. 1998.

[17] H. Schulzrinne, "SIP: more than grandma's phone calls," *Voice On the Net conference (VON)*, Jun. 1999.

H.323 vs SIP

[18] International Telecommunication Union, *H.323 recommendation*, online <URL: http://www.itu.int/itudoc/itu-t/rec/h/h323.html>

[19] I. Dalgic and H. Fang, "Comparison of H.323 and SIP for IP Telephony Signaling," *Photonics East, Proceeding of SPIE'99*, Boston, Massachusetts, Sep.1999.

[20] H. Schulzrinne and J. Rosenberg, "A Comparison of SIP and H.323 for Internet Telephony," *proceedings of the 1998 Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV'98)*, Jul. 1998, Cambridge, England.

[21] L. De Carmo, "Internet Telephony Protocols: H.323 vs. SIP," *Dr. Dobbs Journal*, Jul. 1999.

[22] N. Beijar, "Signaling protocols for Internet Telephony. Architectures based on H.323 and SIP," Department of Electrical and Communications Engineering, Helsinki University of Technology, Espoo, Finland, Oct. 1998.

Other protocols

[23] M. Handley and V. Jacobson, "SDP: Session Description Protocol," *Request for Comments 2327, Internet Engineering Task Force*, Apr. 1998.

[24] H. Schulzrinne, S. Casner, R. Frederick and V. Jacobson, "RTP: A Transport Protocol for Real-Time Applications," *Request for Comments 1889, Internet Engineering Task Force,* Jan. 1996.

[25] H. Schulzrinne, R. Lanphier and A. Rao, "RTSP: Real Time Streaming Protocol," *Request for Comments 2326, Internet Engineering Task Force*, Apr. 1998.

[26] M. Handley, C. Perkins and E. Whelan, "Session Announcement Protocol," online <URL: http://www.ietf.org/internet-drafts/draft-ietf-mmusic-sap-v2-03.txt>, Internet Engineering Task Force, Oct. 1999. *Work in progress.*

[27] J. Rosenberg and H. Schulzrinne, "The IETF Internet Telephony Architecture and Protocols," *IEEE Network*, vol. 13, pp. 18-23, May/Jun. 1999.

[28] The Internet Engineering Task Force, online <URL: http://www.ietf.org/>.

[29] Index of Requests For Comments (RFC's), Internet Engineering Task Force, online <URL: http://www.ietf.org/rfc.html/>.

[30] Multiparty Multimedia Session Control (MMUSIC), online <URL: http://www.ietf.org/html.charters/mmusic-charter.html>.

[31] D. Mills, "Network Time Protocol (version 3) specification and implementation", *Request For Comments 1305*, Mar. 1992.

[32] M. Wahl, T. Howes and S. Kille, "Lightweight Directory Access Protocol (v3)," *Request For Comments 2251,* Dec. 1997.

[33] D. Raggett, A. L. Hors, and I. Jacobs, "HTML 4.0 specification," online <URL: http://www.w3.org/TR/REC-html40/> , Apr. 1998

CPL

[34] J. Lennox and H. Schulzrinne, "Call Processing Language Framework and Requirements," *Internet Draft, Internet Engineering Task Force*, Oct. 1999. *Work in progress.*

[35] J. Lennox and H. Schulzrinne, “CPL: a language for user control of Internet telephony services,” *Internet Draft, Internet Engineering Task Force,* Mar. 1999. *Work in progress.*

XML

[36] M. Leventhal, D. Lewis and M. Fuchs, *Designing XML applications,* The definitive XML series from Charles F. Goldfarb.

[37] “XML-RPC Home Page,” online <URL: http://www.xml-rpc.com/>, UserLand Software Inc., 1999.

[38] “Extensible Markup Language (XML) 1.0,” online <URL: http://www.w3.org/TR/ REC-xml>, W3C Recommendation 10 Feb.1998.

[39] R. Cover, “The SGML / XML Web Page,” online <URL: http://www.oasis-open.org/cover/sgml-xml.html>, Sep. 1999.

[40] XML Resource Guide - XML Parsers, online <URL: http://www.xml.com/xml/pub/Guide/XML_Parsers>

[41] Free XML software, online <URL: http://www.stud.ifi.uio.no/~lmariusg/linker/XMLtools.html#SC_XML>

[42] Working with XML, online <URL: http://java.sun.com/xml/docs/tutorial/overview/index.html>

[43] N. Walsh, “A Technical Introduction to XML,” online <URL: http://xml.com/xml/pub/98/10/guide0.html>

[44] D. Veillard, “The XML library for GNOME,” online <URL: http://rufus.w3.org/veillard/XML/xml.html>

[45] The Open Source Initiative (OSI), online <URL: http://www.opensource.org/>

[46] GNU Network Object Model Environment (GNOME), online <URL: http://www.gnome.org/>

UML

[47] UML 1.1 Standard, online <URL: http://www.rational.com>

[48] M. Fowler and K. Scott, “UML distilled, applying the standard modeling language,” Addison-Wesley, 1997.

Others

[49] T. Berners-Lee, R. Fielding and L. Masinter, "Uniform resource identifiers (URI): generic syntax", RFC 2396, August 1998.

[50] CLC library from Columbia University, online <ftp://ftp.cs.columbia.edu/pub/sos/lib/>

[51] The JAVA tutorial, online <URL: http://java.sun.com/docs/books/tutorial/ >.

[52] J. Kuthan, “Internet Telephony - An Overview,” online <URL: http://www.fokus.gmd.de/research/cc/glone/projects/ipt/abstract.html>.

[53] The Common Gateway Interface, online <URL: http://hoohoo.ncsa.uiuc.edu/cgi/>.

[54] Answers for IT professionals, online <URL: http://www.inquiry.com>

[55] Scripting News, online <URL: http://www.scripting.com/>.

[56] GNU Project web server, online <URL: http://www.gnu.org>.

[57] Helsinki University of Technology, online <URL: http://www.hut.fi/>.

[58] Columbia University, online <URL: http://www.cs.columbia.edu/>.

[59] T. Laakso, “How to write a diploma Thesis,” Department of Electrical and Communications Engineering, Helsinki University of Technology, Espoo, 1999.

Annex A

A complex example

This example shows the sort of sophisticated behavior, which can be achieved by combining CPL nodes. In this case, the user attempts to have his calls reach his desk; if he does not answer within a small amount of time, calls from his boss are forwarded to his celphone, and all other calls are directed to voicemail [35]

```
<?xml version="1.0" ?>
  <!DOCTYPE call SYSTEM "cpl.dtd">

  <call>
   <location url="sip:jones@phone.example.com">
    <proxy timeout="8s">
     <busy>
      <location url="sip:jones@voicemail.example.com" merge="clear">
       <redirect />
      </location>
     </busy>
     <noanswer>
      <string-switch field="from">
       <string matches="boss@*example.com">
        <location url="phone:+19175551212" merge="clear">
         <proxy />
        </location>
       </string>
       <otherwise>
        <location url="sip:jones@voicemail.example.com" merge="clear">
         <redirect />
        </location>
       </otherwise>
      </string-switch>
     </noanswer>
    </proxy>
   </location>
  </call>
```

cpl.dtd file

XML documents may, and should, begin with an XML declaration that specifies the version of XML being used. Next, *Entity declarations* allow you to associate a name with some other fragment of content. There are three kinds of entities: internal entities, external entities and parameter entities. Parameter entities can only occur in the DTD. A parameter entity declaration is identified by placing % (percent-space) in front of its name in the declaration. Parameter entities are not recognized in the body of a document [38].

Element type declarations identify the names of elements and the nature of their content while *attribute list declarations*, identify which elements may have attributes, what attributes they may have, what values the attributes may hold, and what value is the default.

```
<?xml version="1.0" encoding="US-ASCII" ?>

<!-- Define types of nodes -->
<!-- Switch nodes -->

<!ENTITY % Switch 'string-switch|time-switch' >

<!-- Location nodes -->

<!ENTITY % Location 'location|lookup' >

<!-- Signalling action nodes -->

<!ENTITY % SignallingAction 'proxy|redirect|response' >

<!-- Other actions -->

<!ENTITY % OtherAction 'notify|log' >

<!-- Nodes are one of the above four categories, or a link. This entity (macro)
describes the contents of an output. -->

<!ENTITY % Node     '%Location;|%Switch;|%SignallingAction;|
                 %OtherAction;|link' >

<!-- Nodes can have link IDs. Since this is an attribute of every node, we need to
define it early. -->

<!ENTITY % Link-ID 'id  ID    #IMPLIED'>

<!-- Switches: choices a CPL script can make. -->
<!-- All switches contain an 'otherwise' node. -->

<!ELEMENT otherwise ( %Node; ) >

<!-- String-switch makes choices based on strings. -->

<!ELEMENT string-switch ( string+, otherwise? ) >
<!ATTLIST string-switch
  field  CDATA    #REQUIRED
  %Link-ID;
>

<!ELEMENT string ( %Node; ) >
<!ATTLIST string
  is         CDATA    #IMPLIED
  contains   CDATA    #IMPLIED
  matches    CDATA    #IMPLIED
  comparator CDATA    "i;ascii-casemap"
>

<!-- Time-switch makes choices based on the current time. -->

<!ELEMENT time-switch ( time+, otherwise? ) >
```

```
<!ATTLIST time-switch
  %Link-ID;
>

<!ELEMENT time ( %Node; ) >
<!ATTLIST time
  year       CDATA  #IMPLIED
  month      CDATA  #IMPLIED
  date       CDATA  #IMPLIED
  day        CDATA  #IMPLIED
  timeofday  CDATA  #IMPLIED
>

<!-- Locations: ways to specify the location a subsequent action (proxy, redirect)
will attempt to contact. -->

<!ENTITY % Merge  'merge (merge|clear) "merge"' >

<!ELEMENT location ( %Node; ) >
<!ATTLIST location
  url CDATA    #REQUIRED
  %Merge;
  %Link-ID;
>

<!-- Sources of location lookups that aren't URIs. -->

<!ENTITY % Sources '(registration)' >

<!ELEMENT lookup ( success,notfound?,failure? ) >
<!ATTLIST lookup
 url     CDATA     #IMPLIED
 source  %Sources; #IMPLIED
 timeout CDATA     #IMPLIED
 %Merge;
 %Link-ID;
>

<!ELEMENT success  ( %Node; ) >
<!ELEMENT notfound ( %Node; ) >
<!ELEMENT failure ( %Node; ) >

<!-- Signalling Actions: call-signalling actions the script can
    take. -->

<!ELEMENT proxy ( busy?,noanswer?,failure? ) >
<!ATTLIST proxy
  timeout CDATA   #IMPLIED
  %Link-ID;
>

<!ELEMENT busy ( %Node; ) >
<!ELEMENT noanswer ( %Node; ) >
<!-- "failure" repeats from lookup above.  XXX? -->
```

```
<!ELEMENT redirect EMPTY >

<!ATTLIST redirect
   %Link-ID;
>

<!-- Statuses we can return -->

<!ELEMENT response EMPTY >
<!ATTLIST response
   status CDATA    #REQUIRED
   reason CDATA    #IMPLIED
   %Link-ID;
>

<!-- Non-signalling actions: actions that don't affect the call -->

<!ELEMENT notify ( success,failure? ) >
<!ATTLIST notify
   url     CDATA    #REQUIRED
   comment CDATA    #IMPLIED
   %Link-ID;
>

<!ELEMENT log ( success,failure? ) >
<!ATTLIST log
   name    CDATA    #IMPLIED
   comment CDATA    #IMPLIED
   %Link-ID;
>
<!-- Links to other nodes. -->

<!ELEMENT link EMPTY >
<!ATTLIST link
   ref   IDREF    #REQUIRED
>

<!-- The top-level element of the script. -->

<!ELEMENT call  ( %Node; ) >
```

ERICSSON proud host of the...

3rd SIP Bake-Off

TESTING RESULTS
December 6-8, 1999
Richardson, Texas (USA)

Third SIP Bake off

ENVIRONMENT and AGENDA

Regarding the network lab data:
Default router: 63.86.21.1
Netmask: 255.255.255.0
DNS server: 63.86.21.2
DNS domain: ericy.com

The hostnames in DNS were *sip-stat???.ericy.com* and the dynamic (DHCP) address range started at *63.86.21.180*. The hostnames were *sip-dyn???.ericy.com*

SIP Bake off information is CONFIDENTIAL, for that reason we have omitted the name of the companies involved in our testing results.

- Monday, December 6th

Equipment Setup.
Basic Interoperability Testing.
People installed their equipment and started with some basic signaling messages. We spent all day installing the SIP server in the Solaris workstation and the Iptele program plus the JAVA package in the two PCs borrowed from Ericsson.

- Tuesday, December 7th

Basic Interoperability testing.
After a couple of hours we solved a problem with the libraries required by the SIP server and finally, we also started the testing. Next, we present some of the messages registered.

Company1-HUT testing.

Company1 sent us an INVITE:

```
...
INVITE sip:queca@63.86.21.120 SIP/2.0
Via: SIP/2.0/UDP 63.86.21.63:5060
To: sip:queca@63.86.21.120
From: sip:company1@63.86.21.63
Call-ID: 1086034694@63.86.21.63
CSeq: 1 INVITE
```

```
Content-Type: application/sdp
...
```

We sent back a 200 OK response accepting the invitation.

```
200 sip:queca@63.86.21.120 SIP/2.0
Via: SIP/2.0/UDP 63.86.21.63:5060
To: sip:queca@63.86.21.120
From: sip:company1@63.86.21.63
Call-ID: 1086034694@63.86.21.63
CSeq: 1 INVITE
Content-Length: 350
Content-Type: application/sdp
```

The user, *queca@helsinkiu.ericy.com,* **sent a REGISTER request to the SIP server from the IPtele program. The request included the Internet Call Waiting service XML file.**

```
REGISTER sip:ericy.com SIP/2.0
Via: SIP/2.0/UDP 63.86.21.120:5060
From: sip:queca@helsinkiu.ericy.com
To: sip:queca@helsinkiu.ericy.com
Call-ID: 6088@helsinkiu.ericy.com
Cseq: 1 REGISTER
Contact:<sip:queca@helsinkiu.ericy.com:5060;transport=udp
>
Expires: 7200
Content-Length: 350
Content-Type: application/hut-cpl

<?xml version="1.0"?>
<call Type="ICW">
<proxy>
<icw>
      <forward>
      <link ref="voicemail"/>
      </forward>
      <success>
      <location url="queca@pc2.tct.hut.fi"/>
      </success>
      <reject>The user is Busy now</reject>
</icw>
<busy/>
<noanswer/>
<failure/>
</proxy>
<response status="busy"/>
</call>
```

The SIP server accepted the registration and sent back a 200 OK response:

```
SIP/2.0 200 OK
Via: SIP/2.0/UDP 63.86.21.120:5060
From: sip:queca@helsinkiu.ericy.com
To: sip:queca@helsinkiu.ericy.com
Call-ID: 6088@helsinkiu.ericy.com
Cseq: 1 REGISTER
Date: Wed, 07 Dec 1999 16:49:10 GMT
```

```
Server: Columbia-SIP-Server/1.0
Content-Length: 350
Contact:sip:queca@helsinkiu.ericy.com:5060;transport=udp;
expires="Wed,  07  Dec  1999  17:49:10  GMT";  action=proxy;
q=0.00
Expires: Wed, 07 Dec 1999 17:49:10 GMT
Content-Type: application/hut-cpl

<?xml version="1.0"?>
<call Type="ICW">
<proxy>
<icw>
      <forward>
      <link ref="voicemail"/>
      </forward>
      <success>
      <location url="queca@pc2.tct.hut.fi"/>
      </success>
      <reject>The user is Busy now</reject>
</icw>
<busy/>
<noanswer/>
<failure/>
</proxy>
<response status="busy"/>
</call>
```

Company2 - HUT testing.

Company2 sent us an INVITE and we accepted :

```
INVITE sip:queca@63.86.21.120 SIP/2.0
Via: SIP/2.0/UDP sip-stat128
From:Nicolas<sip:nico@sip-stat128>
To:<sip:queca@sip-stat120.ericy.com>
Call-ID: 4e4a6acac4c3404d@sip-stat128
CSeq: 1 INVITE
Content-Length: 0
User-Agent: Company2 User Agent V0.9
Subject:
Priority: non-urgent

SIP/2.0 180 Ringing
Via: SIP/2.0/UDP sip-stat128
From:Nicolas<sip:nico@sip-stat128>
To:  <sip:queca@sip-stat120.ericy.com>
Call-ID: 4e4a6acac4c3404d@sip-stat128
CSeq: 1 INVITE
Content-Length: 0

SIP/2.0 200 OK
Via: SIP/2.0/UDP sip-stat128
From:Nicolas<sip:nico@sip-stat128>
To:  <sip:queca@sip-stat120.ericy.com>
Call-ID: 4e4a6acac4c3404d@sip-stat128
CSeq: 1 INVITE
Contact:
<sip:queca@helsinkiu.ericy.com:5060;transport=udp>
Content-Type:application/sdp
```

```
Content-Length: 213

v=0
o=queca 2890844526 2890842807 IN IP4 63.86.21.120
s=ICW SIP
i=A Session for voice transmission using SIP
e= queca@63.86.21.120
p=+358 9 4514785
c=IN IP4 63.86.21.120
t=0 0
m=audio 50250 RTP/AVP 0
a=rtpmap:0 PCMU/8000
```

Afterwards Company2 sent us another INVITE and we rejected the call.

```
INVITE sip:queca@63.86.21.120 SIP/2.0
Via: SIP/2.0/UDP sip-stat128
From:Nicolas<sip:nico@sip-stat128>
To:  <sip:queca@sip-stat120.ericy.com>
Call-ID: 4e4a6acac4c3404f@sip-stat128
CSeq: 1 INVITE
Content-Length: 0
User-Agent: Company2 User Agent V0.9
Subject:
Priority: non-urgent

SIP/2.0 180 Ringing
Via: SIP/2.0/UDP sip-stat128
From:Nicolas<sip:nico@sip-stat128>
To:  <sip:queca@sip-stat120.ericy.com>
Call-ID: 4e4a6acac4c3404f@sip-stat128
CSeq: 1 INVITE
Content-Length: 0

SIP/2.0 603 Decline
Via: SIP/2.0/UDP sip-stat128
From:Nicolas<sip:nico@sipstat-128>
To:  <sip:queca@sip-stat120.ericy.com>
Call-ID: 4e4a6acac4c3404f@sip-stat128
CSeq: 1 INVITE
Content-Length: 0
```

Company3- HUT testing.

We sent an INVITE to Company3 and they accepted the call. The session was established but we could not talk due to a JAVA exception (new version of JMF 2.0) that we could not solve during the SIP Bake off.

```
INVITE sip:063@63.86.21.63 SIP/2.0
Via: SIP/2.0/UDP 63.86.21.120:5060
From: sip:queca@helsinkiu.ericy.com
To: sip:063@63.86.21.63
Call-ID: 1258@helsinkiu.ericy.com
CSeq: 1 INVITE
Contact:
<sip:queca@helsinkiu.ericy.com:5060;transport=udp>
Content-Type: application/sdp
Content-Length: 218
```

```
v=0
o=queca 2890844526 2890842807 IN IP4 63.86.21.120
s=ICW SIP
i=A Session for voice transmission using SIP
e=063@63.86.21.63
p=+358 9 4514785
c=IN IP4 63.86.21.120
t=0 0
m=audio 50250 RTP/AVP 0
a=rtpmap:0 PCMU/8000

SIP/2.0 180 Ringing
Via: SIP/2.0/UDP 63.86.21.120:5060
To: sip:063@63.86.21.63
From: sip:queca@helsinkiu.ericy.com
Call-ID: 1258@helsinkiu.ericy.com
CSeq: 1 INVITE
Content-Length: 218
Content-Type: application/sdp

v=0
o=RAIDer 9912080204 9912080204 IN IP4 63.86.21.63
s=Company3
p=+1 972 642 1000
c=IN IP4 63.86.21.63
t=0 0
a=ptime:60
m=audio 49152 RTP/AVP 0
a=rtpmap:0 pcmu/8000/1

SIP/2.0 100 Trying
Via: SIP/2.0/UDP 63.86.21.120:5060
To: sip:063@63.86.21.63
From: sip:queca@helsinkiu.ericy.com
Call-ID: 1258@helsinkiu.ericy.com
CSeq: 2 INVITE
Content-Length: 0

SIP/2.0 200 OK
Via: SIP/2.0/UDP 63.86.21.120:5060
To: sip:063@63.86.21.63
From: sip:queca@helsinkiu.ericy.com
Call-ID: 1258@helsinkiu.ericy.com
CSeq: 3 INVITE
Contact: sip:company3@63.86.21.120

ACK sip:063@63.86.21.63 SIP/2.0
Via: SIP/2.0/UDP 63.86.21.120:5060
From: sip:queca@helsinkiu.ericy.com
To: sip:063@63.86.21.63
Call-ID: 1258@helsinkiu.ericy.com
CSeq: 1 ACK
Content-Length: 0
…
BYE sip:queca@helsinkiu.ericy.com:5060 SIP/2.0
Via: SIP/2.0/UDP 63.86.21.63:5060
To: sip:queca@helsinkiu.ericy.com
```

```
From: sip:063@63.86.21.63
Call-ID: 1258@helsinkiu.ericy.com
CSeq: 4 BYE
Content-Length: 0
```

- Wednesday, December 8th

Advanced interoperability testing, complex network setup.

We sent a REGISTER to the SIP server form IPtele but we did not get back a 200 response. When one of the companies sent us an INVITE, the server did not recognize the user and sent back a 404 Not Found response.

```
...
INVITE sip:063@63.86.21.63 SIP/2.0
Via: SIP/2.0/UDP 63.86.21.120:5060
From: sip:pepe@helsinkiu.ericy.com
To: sip:queca@non-esoe.ericy.com
Call-ID: 7130@helsinkiu.ericy.com
Cseq: 2 INVITE
Date: Wed, 08 Dec 1999 18:21:13 GMT
Server: Columbia-SIP-Server/1.0
Content-Length: 0

SIP/2.0 100 Trying
Via: SIP/2.0/UDP 63.86.21.120:5060
From: sip:pepe@helsinkiu.ericy.com
To: sip:queca@non-esoe.ericy.com
Call-ID: 7130@helsinkiu.ericy.com
Cseq: 2 INVITE
Date: Wed, 08 Dec 1999 18:21:13 GMT
Server: Columbia-SIP-Server/1.0
Content-Length: 0

SIP/2.0 404 Not Found
Via: SIP/2.0/UDP 63.86.21.120:5060
From: sip:pepe@helsinkiu.ericy.com
To: sip:queca@non-esoe.ericy.com
Call-ID: 7130@helsinkiu.ericy.com
Cseq: 2 INVITE
Date: Wed, 08 Dec 1999 18:21:13 GMT
Server: Columbia-SIP-Server/1.0
Content-Length: 0
```