# Service provisioning in two open-source SIP implementations, CINEMA and VOCAL

A thesis submitted in fulfillment of the
requirements for the degree of
MASTER OF SCIENCE
of
RHODES UNIVERSITY

by

**Ming Chih Hsieh**

December 2003

## Acknowledgments

I would like to thank the Internet Telephony Group at the Department of Computer Science, Rhodes University, for working as a group towards shared goals. Thank you especially to my supervisors Joshua Okuthe and Alfredo Terzoli for a fulfilling two years of study and a lot of learning. Thank you to my friend Danny for listening to some outrageous ideas and for correcting me when those ideas seemed out of range.

I am also grateful to my family for their personal encouragement via continuous emails and phone calls. This communication helped me to remain in focus and determined to finish my project.

Finally, I would like to acknowledge the importance of the financial support provided by the National Research Foundation, South Africa, as well as sponsorship for conference trips provided by the Department of Computer Science.

## Abstract

The distribution of real-time multimedia streams is seen nowadays as the next step forward for the Internet. One of the most obvious uses of such streams is to support telephony over the Internet, replacing and improving traditional telephony.

This thesis investigates the development and deployment of services in two Internet telephony environments, namely CINEMA (Columbia InterNet Extensible Multimedia Architecture) and VOCAL (Vovida Open Communication Application Library), both based on the Session Initiation Protocol (SIP) and open-sourced.

A classification of services is proposed, which divides services into two large groups: basic and advanced services. Basic services are services such as making point-to-point calls, registering with the server and making calls via the server. Any other service is considered an advanced service. Advanced services are defined by four categories: *Call Related*, *Interactive*, *Internetworking* and *Hybrid*.

New services were implemented for the *Call Related*, *Interactive* and *Internetworking* categories. First, features involving call blocking, call screening and missed calls were implemented in the two environments in order to investigate *Call-related* services. Next, a notification feature was implemented in both environments in order to investigate *Interactive* services. Finally, a translator between MGCP and SIP was developed to investigate an *Internetworking* service in the VOCAL environment.

The practical implementation of the new features just described was used to answer questions about the location of the services, as well as the level of required expertise and the ease or difficulty experienced in creating services in each of the two environments.

## Table of Contents

## List of Figures

## List of Tables

# Chapter 1 General Introduction

## 1.1 Internet telephony and SIP

> "Data has overtaken voice as the primary traffic on many networks built for voice. Soon voice networks will run on top of networks built with a data-centric approach." [Davidson and Peters, 2000]

*IP telephony*, or *voice-over-IP* (*VoIP*), is the use of IP data connections to exchange real-time voice that have been traditionally carried by public, or private, switched telephone networks. A closely related phrase, *Internet telephony,* used to refer more specifically to the use of the Internet for carrying telephone traffic, as opposed to a private LAN within an organization. Over time, however, IP telephony, VoIP and Internet telephony have become equivalent and have been extended to embrace the transport, in real-time, not just of audio or fax data but video and shared applications over IP networks [Rosenberg and Schulzrinne, 1999].

Since the deregulation of the telecommunications industry, new operators regularly enter the market. Often these operators use VoIP in order to save on costs and be competitive with already established operators.

The work reported in this thesis focuses on service provisioning for Internet telephony using a protocol developed by the Internet Engineering Task Force (IETF) known as Session Initiation Protocol (SIP). SIP is defined in RFC2543 of the MMUSIC (Multiparty Multimedia Session Control) working group of the IETF [Gurle et al., 1999]. Since so much interest was generated for this protocol a separate working group, the SIP charter working group, was later formed. The SIP charter working group provides an open forum where developers and other people interested in the protocol can discuss modifications or extensions to it [SIPCharter, 2003].

SIP began as part of a set of utilities and protocols developed for the MBONE (Multicast Backbone) network. MBONE was established as an experimental multicast network used to distribute multimedia streams in multimedia sessions. SIP was one of the components used to invite users to multimedia sessions. Subsequently, SIP began to be used as a signaling protocol to provide telephony over the Internet. SIP was built from established protocols such as the Hypertext Transfer Protocol (HTTP) and the Simple Mail Transfer Protocol (SMTP). This makes SIP easily compatible with today's Internet operations [Rosenberg and Shockey, 2000].

Indicating your willingness to communicate with another person is more complicated than it seems [Rosenberg and Schulzrinne, 1998a]. This indication is called signaling. In computer networks, signaling includes name resolution and user location. Name resolution involves resolving the name utilized to call a friend into a resource that can be contacted. For example, an alias or nickname used to refer to a friend must be resolved into a unique identifier used to locate that person. 'User location' determines the exact location of the friend you are trying to communicate with and this can be quite complex. Other factors also affect signaling, including media capabilities and the user's preferences for communication. For example, if a user does not have the particular media capability that you want to use in your communication then he or she might want to redirect you to someone that has that capability. The user's preferences can also affect signaling; for example, the recipient might want to accept calls only during office hours.

SIP seems able to effortlessly provide all the functionality required for a signaling protocol in Internet telephony; thus, it was chosen to investigate service creation during this project primarily for this reason.

## 1.2 Internet telephony and legacy telephony

Telephony started out as two telephones connected to each other with a pair of copper wires. When other telephones were added, efficient ways to connect to each of the

telephones had to be found. At first, a human operator was introduced, to control a central connector (switch board) to all phones in a group and to other groups.

Figure 1.1 illustrates what a traditional telephone network looked like before a human operator was introduced.

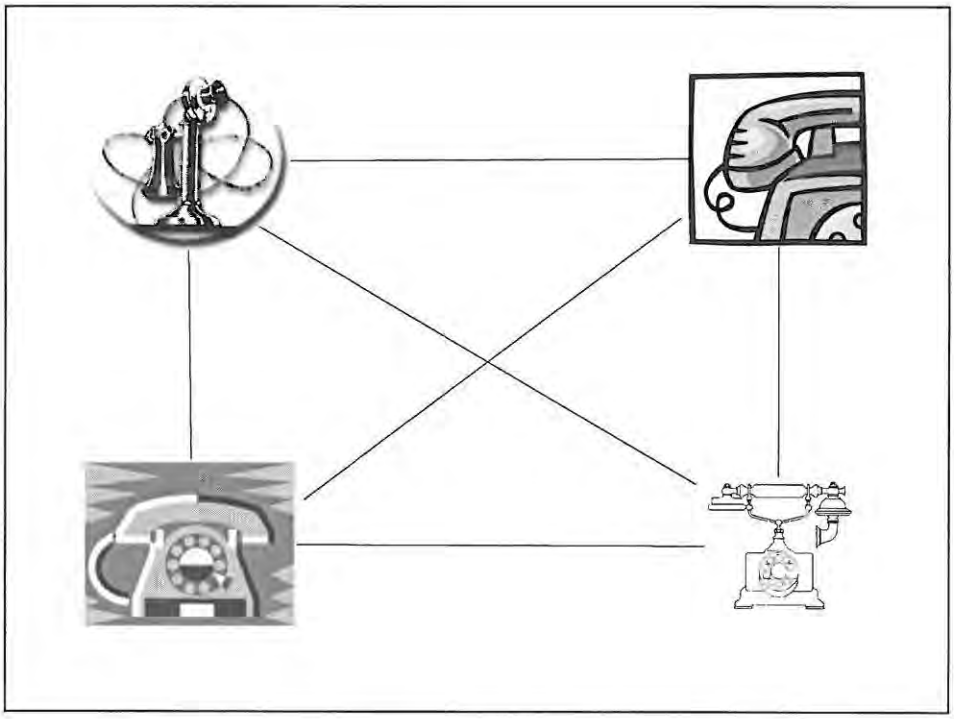

**Figure 1.1 A telephone network without switching**

Figure 1.2 shows the traditional telephone network utilizing an operator.

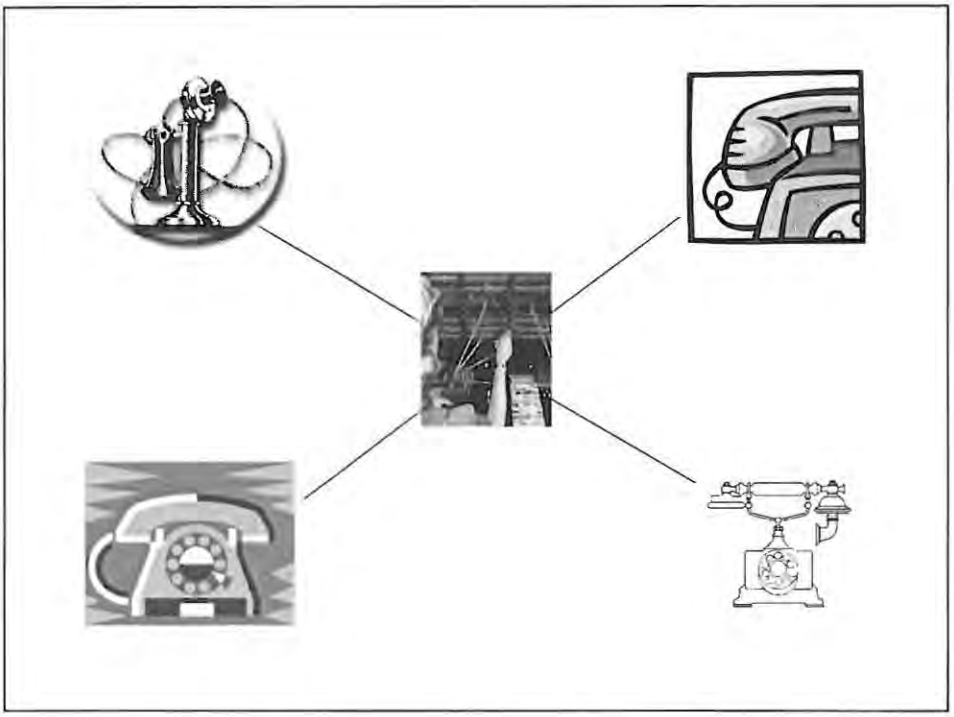

**Figure 1.2 A telephone network with a human operator**

The human operator was later replaced by a mechanical switch and further improved upon with an electronic switch. This type of network became known as the Public Switch Telephone Network (PSTN). PSTN telephony is referred to throughout this thesis as legacy or traditional telephony.

Introducing services into this type of network was difficult, partly because it used specialized and centralized switches. The switch provided the network with a centralized point of control where services could be created and controlled, but at the time the switch was introduced third parties other than the carriers manufactured it. This meant that as new services were created the switch manufacturer had to modify or replace them, since the carriers themselves typically could not modify the switches. Service provisioning in traditional telephony improved substantially with the introduction of the Intelligent Network (IN).

This thesis is not intended to compare Internet and traditional telephony, and so IN is not further discussed, beyond providing a few references relating to IN-Internet hybrid

systems later on. For the interested reader, an introduction to IN networks is available through the International Engineering Consortium [IEC, 2003a and IEC, 2003b]. For a review on existing techniques for creating services in IN see [Lennox et al., 1999b].

In contrast to traditional networks, IP networks are packet-oriented and distributed. Connections are set up virtually in IP networks and this allows for multiple connections to be set up on a single physical line, thus freeing bandwidth for other usages.

Figure 1.3 represents what Internet telephony typically looks like today.

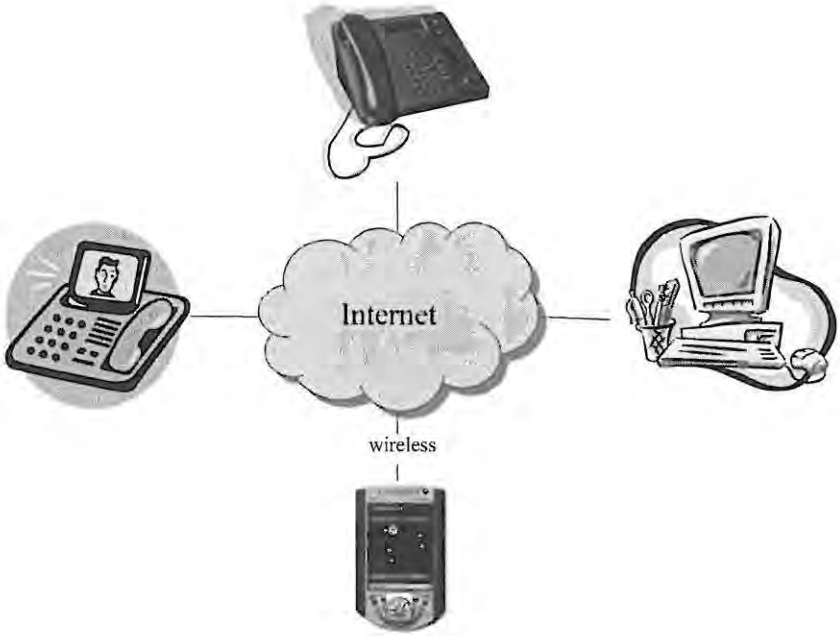


**Figure 1.3 Internet telephony**

In general, endpoints in Internet telephony are much more intelligent than the endpoints in traditional telephony and can handle signaling as well as the media component in a sophisticated manner. For example, media data are usually compressed, to save on bandwidth, a capability that is lacking in the endpoints of traditional telephony. One advantage, in the context of service creation, to having endpoints that are more intelligent is that many services that traditionally resided at the center of the network, because of the

availability of computing resources there, can be moved to the edge of the network [Lennox et al., 1999a].

Research has been done on the integration between Intelligent Networks (IN) and Internet telephony. Services in IN can be reproduced in Internet telephony as shown in [Lennox et al., 1999a]. IN services can naturally be accessed from packet networks, as shown in [Chapron and Chatras, 2001]. Research has also been done whereby the architecture of IN is applied to Internet telephony resulting in IN-Internet hybrid systems. For an in depth analysis on how the IN architectural framework can be applied to Internet telephony see [Glitho, 2001]. [Andreetto et al., 2001] provides an architecture on which services can be developed for IN-Internet hybrid systems using an API-based approach.

Another approach towards IN-Internet hybrid systems is to use Mobile Agent Technology, based on Distributed Object Technology, to create a unified service framework for the Internet and PSTN as shown in [Chatzipapadopoulos, 2000]. [Anjum et al., 2001] provides an innovative way of creating services whereby third-party software components can be dynamically downloaded and installed from the network, as needed for IN-Internet hybrid systems. This allows advanced services to be deployed and delivered to users rapidly.

## 1.3 SIP in relation to other protocols

SIP is not the only signaling protocol for Internet telephony. H.323 is another major suite of specifications for Internet telephony (in the latter's wider meaning, as defined in section 1.1). H.323 was developed by the International Telecommunications Union (ITU), a few years before SIP, and thus is more established than SIP, with support from many vendors. A tutorial description of H.323 can be found at [IEC, 2003c].

There are numerous differences between SIP and H.323.

H.323 is a complete specification of the signaling, media capabilities and all other components of the telephony system. This approach, whereby the protocol specifies everything, gives the programmer the advantage of focusing just on service creation. SIP, however, uses a modular approach towards Internet telephony. In this sense, it is directly concerned only with establishing sessions among multiple participants and leaves issues such media description to the Session Description Protocol (SDP) and Quality of Service (QoS) issues to the Resource ReSerVation Protocol (RSVP). This modular approach also relates to the ability of SIP to internetwork with other networks such as the PSTN. For example, in order to internetwork, SIP can use another protocol to control media gateways, namely the Media Gateway Control Protocol (MGCP). Media gateways are gateways that convert media from one network to another, such as a gateway that converts audio from the PSTN to audio in RTP packets for IP.

H.323 accommodates little room for change, flexibility or different architectures. For example, the H.323 standard specifies that the Real Time Protocol (RTP) must be used to transport audio and video media, while the SIP specification does not mandate it. Detailed comparisons by [Wind River, 2002] and [Nortel Networks, 2000a] show the advantages and disadvantages of SIP vs. H.323.

## 1.4 Aim of the project

Creating new services will make Internet telephony better. The aim of this project is to investigate various ways to create services for Internet telephony in two specific and well known SIP environments, namely CINEMA and VOCAL.

CINEMA (Columbia InterNet Extensible Multimedia Architecture) is the result of work done at Columbia University (New York), and VOCAL (Vovida Open Communication Application Library) is a SIP implementation from Vovida. Vovida is an organization based on the Internet aimed at providing open-source communication software.

More specifically, an investigation into service creation should lead to answers to the following questions:

*"Where in the architecture can services be deployed?"*
Asking this question is essential to the operation and management of the service and relates to other questions such as "Should the service be at the edge or at the center of the network?"

*"What level of expertise is required to create services?"*
The answer to this question determines what level of skill is required from the programmer.

*"How easy it is to create a service?"*
The answer to this question is very significant for developers.

The work reported in this thesis is part of the work done by the Internet Telephony Group (ITG) at the Department of Computer Science at Rhodes University in South Africa. The presence of this group was essential for practical experimentation with internetworking, as one will see later in the thesis.

## 1.5 Structure of the thesis

The structure of the thesis is as follows:

- **Protocol Overview (Chapter 2)** gives a basic introduction to the structure of the Session Initiation Protocol (SIP), including call flow diagrams detailing how SIP messages are exchanged between different entities.
- **Two SIP Architectures (Chapter 3)** describes the two environments used to deploy Internet telephony in order to investigate service creation. The environments used were CINEMA and VOCAL. A section that contrasts the two environments is also presented.

- **Service Categories (Chapter 4)** introduces the service categories used in this thesis. Services were divided into four categories: *Call-related*, *Interactive*, *Internetworking* and *Hybrid*. The subsequent three chapters focus on *Call-related*, *Interactive* and *Internetworking* services, respectively.
- **Call-related Services (Chapter 5)** explains the standard mechanisms available to create *Call-related* services. The mechanisms used in the project are SIP-CGI (SIP-Common Gateway Interface) and CPL (Call Processing Language), and the practical development of some services is reported to demonstrate the capabilities of those mechanisms. A discussion that contrasts SIP-CGI and CPL is also provided.
- **Interactive Services (Chapter 6)** begins with an investigation into how the voicemail service, catagorized as an *interactive* service, is implemented in the two environments. The investigation into the voicemail service is used as a springboard to implement a reminder service in the two environments. The different implementation processes in the two environments are discussed.
- **Internetworking Services (Chapter 7)** reports first the investigation into the mechanisms in each environment that are available to enable internetworking between H.323 and SIP networks. A SIPMGCP translator is then described. This translator was developed to provide SIP users in VOCAL access to an MGCP network. Finally, the internetwork mechanisms in the two environments are compared.
- **Conclusions and Extensions (Chapter 8)** provide a summary of the research and answers the questions introduced as part of the objectives of the work. The chapter also discusses possible extensions to the research.

## 1.6 Summary

This chapter introduced IP telephony, a phrase now used to mean communication using any media over IP (Internet Protocol) networks. SIP (Service Initiation Protocol) and its origins were described. We also briefly compared Internet telephony to legacy telephony,

and considered SIP relative to other protocols. Finally, the aim of the project and the structure of the thesis were presented.

# Chapter 2 Protocol Overview

## 2.1 Overview of SIP

The Session Initiation Protocol (SIP) is an application-layer control protocol that resides in the 7th layer of the Open Systems Interconnect (OSI) model. The protocol can be used for creating, modifying and terminating multimedia sessions. Examples of multimedia sessions are:

- Internet multimedia conferences
- Internet telephone calls
- Any form of audio, video and data communications.

SIP supports five steps towards 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**: establishment of call parameters for both the called and calling party;
- **Call handling**: transfer and termination of calls.

SIP is designed as part of the overall Internet Engineering Task Force (IETF) multimedia data and control architecture. The IETF multimedia data and control architecture incorporates other protocols, such as Resource reSerVation Protocol (RSVP), Real Time Protocol (RTP), Real Time Streaming Protocol (RTSP), Session Announcement Protocol (SAP), for advertising multimedia sessions via multicast, as well as Session Description Protocol (SDP) for media description [Handley and Jacobson, 1998]. SIP is defined in RFC2543 (Request For Comments 2543) [Handley et al., 1999].

SIP uses terms and header fields from HyperText Transfer Protocol (HTTP) [Berners-Lee et al., 1997]. Terms such as client, server and proxy are common in HTTP and SIP. Also, the basic operation of SIP is similar to HTTP.

SIP uses an email-like address scheme, such as user@domain or user@host. This allows for easy location of a particular user by doing lookups on the domain or the host machine. The actual addressing scheme is based on Uniform Resource Identifiers (URI) and Uniform Resource Locators (URL) [Berners-Lee et al., 1998]. In this work SIP addresses are referred to as SIP URLs. A particular SIP address or SIP URL would be sip:user@domain.com.

## 2.2 Terminology

The following simplified definitions are taken from RFC2543:

*Call*: A call consists of all participants in a conference, invited by a common source. A SIP call is identified by a globally unique Call-ID. Thus, if several people invite a user to the same multicast session, for example, each of these invitations will have a unique Call-ID. A point-to-point Internet telephony conversation maps into a single SIP call. In a Multiparty Conference Unit (MCU) call-in conference, each participant uses a separate call to invite himself to the MCU.

*Client*: A client is an application program that sends SIP requests. Clients may or may not interact directly with a human user. User agents and proxies contain clients.

*Conference*: A conference is a multimedia *session*, identified by a common session description. A conference can have zero or more members and includes the cases of a multicast conference and a two-party "telephone call", as well as combinations of these. Any number of calls can be used to create a conference. A member is a participant in a call.

*Initiator, calling party, caller*: These terms indicate the party initiating a conference invitation. Note that the calling party need not be the same as the one creating the conference.

*Invitation*: An invitation is a request sent to a user (or service) requesting participation in a session. A successful SIP invitation consists of two transactions: an INVITE request followed by an ACK request.

*Invitee, invited user, called party, callee*: These terms indicate the person or service that the calling party tries to invite to a conference.

*Location service*: Location service is what a SIP redirect or proxy server uses to obtain information about a callee's possible location(s). Location servers offer location services. Location servers may be co-located with a SIP server.

*Proxy, proxy server*: This is an intermediary program that acts as both a server and a client for the purpose of making requests on behalf of other clients. Requests are serviced internally or by passing them on, possibly after translation, to other servers. A proxy interprets, and, if necessary, rewrites a request message before forwarding it.

*Redirect server*: A redirect server accepts a SIP request, maps the address into zero or more new addresses and returns these addresses to the client. Unlike a proxy server, it does not initiate its own SIP request. Unlike a user agent server, it does not accept calls.

*Registrar*: A registrar is a server that accepts REGISTER requests. A registrar is typically co-located with a proxy or redirect server and may offer location services.

*Server*: A server is an application program that accepts requests and services them, sending responses back to the requesting entity. There are proxy, redirect and registrar servers.

*Session*: The SDP specification states: "A multimedia session is a set of multimedia senders and receivers and the data streams flowing from senders to receivers. A multimedia conference is an example of a multimedia session (RFC 2327) [Handley and Jacobson, 1998]." (A session as defined for SDP can comprise one or more RTP sessions.) As defined, a callee can be invited several times, by different calls, to the same session. If SDP is used, a session is defined by the concatenation of the username, session id, network type, address type and address elements in the origin field.

*SIP transaction*: A SIP transaction occurs between a client and a server and comprises all messages from the first request sent from the client to the server up to a final (non-1xx) response sent from the server to the client. The command sequence (CSeq) number identifies a SIP transaction. The ACK request has the same CSeq number as the corresponding INVITE request, but is a transaction of its own.

*User agent*: A user agent consists of a User Agent Server (UAS) and a User Agent Client (UAC).

## 2.3 Overview of SIP operations

SIP signaling is based on the Client-Server protocol. The client issues request messages while the server responds by issuing response messages. The user agent in SIP usually contains a user-agent client and a user-agent server so that it is able to handle both requests and responses.

Figure 2.1 diagrams call flow between two users, namely Ming and Cspw. Ming is the initiator of the call, i.e., the caller for the session. Cspw is the recipient of the call, i.e., the callee for the session.

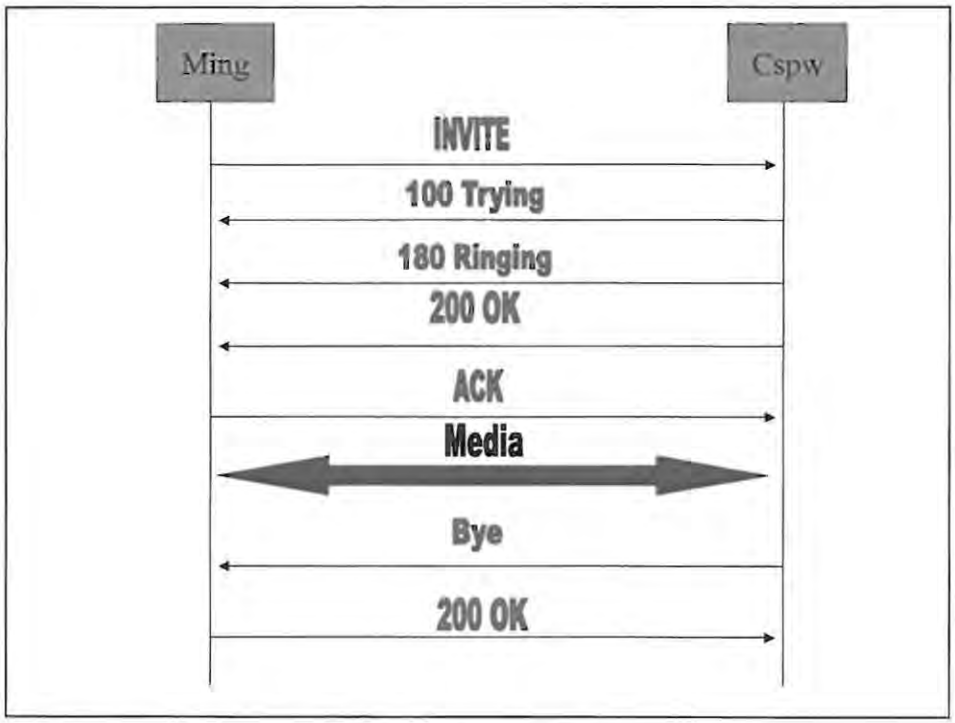


**Figure 2.1 Call flow between Ming and Cspw**

The actual messages indicated in Figure 2.1 are described next:

**The invitation sequence—**

**1. The INVITE Message**

Ming, the initiator of the call, sends an INVITE request message to cspw@cspwnb.cs.ru.ac.za from ming@csmsc01.cs.ru.ac.za. The IP address for csmsc01.cs.ru.ac.za is 146.231.26.153 and for cspwnb.cs.ru.ac.za it is 146.231.29.63.

```
INVITE sip:cspw@cspwnb.cs.ru.ac.za SIP/2.0
Via: SIP/2.0/UDP 146.231.26.153:5060
From: sip:ming@csmsc01.cs.ru.ac.za
To: sip:cspw@cspwnb.cs.ru.ac.za
Contact: sip:ming@146.231.26.153:5060
Call-ID: 442388787@146.231.26.153
CSeq: 1 INVITE
```

Subject: testing
Expires: 3600
Content-Type: application/sdp
Content-Length: 133
v=0
o=ming 679139736425 1008247652 IN IP4 146.231.26.153
s=testing
c=IN IP4 146.231.26.153
t=0 0
m=audio 10000 RTP/AVP 0

This exemplifies the structure for an INVITE message. The message starts off with the request message type, in this case INVITE, followed by the SIP Uniform Resource Locator (URL) of the person to be called, followed by the SIP version number. There are five other possible message request types: REGISTER, OPTIONS, BYE, CANCEL and ACK.

The "Via" field indicates where the message originates from. It contains the IP address and port number of the machine from which the message was sent.

The "From" field contains the SIP URL of the person calling.

The "To" field contains the SIP URL of the callee.

The "Contact" address field contains the various locations where the caller can be contacted.

The "Call-ID" general-header field uniquely identifies a particular invitation or all registrations of a particular client. It contains the Call-ID and the hostname (Call-ID@host). Call-IDs are cryptographically generated. The host part of the Call-ID can be a fully qualified domain name or a globally routable IP address.

The "CSeq" (Command Sequence) general-header field contains a decimal number followed by the request type. The decimal number is unique within the generated Call-ID number.

The "Subject" header contains the subject of the session; it can be any alphanumeric string.

The "Expires" entity-header field gives the amount of time for the validity of the invitation. The amount can be indicated in seconds or in date/time format. In this case 3600 indicates the amount of seconds that the caller is willing to wait before the invitation expires. The field can be used both in requests and responses.

Embedded inside the SIP message is the SIP message body. The SIP message body is usually used to describe the type of session to take place. SIP uses the Session Description Protocol (SDP) for this function, but it does not mandate it. The "Content-Length" field is used to indicate the length of the SIP message body. The caller inputs the codecs that it is able to handle, for the type of session to be undertaken, inside the SDP. The content of the SIP message body will not be explained here.

**2. The 100 Trying Message**

The 100 Trying Message is sent by the callee's user agent to indicate to the caller that it has received the INVITE message and is busy processing it. The 100 Trying Message is a response message.

```
SIP/2.0 100 Trying
Via: SIP/2.0/UDP 146.231.26.153:5060
From: sip:ming@csmsc01.cs.ru.ac.za
To: sip:cspw@cspwnb.cs.ru.ac.za
Call-ID: 442388787@146.231.26.153
CSeq: 1 INVITE
Content-Length: 0
```

Notice that there is no SDP inside this message.

**3. The 180 Ringing Message**

The 180 Ringing Message is sent by the callee's user agent to indicate to the caller that it is in a ringing state and waiting for the callee's response.

SIP/2.0 180 Ringing
Via: SIP/2.0/UDP 146.231.26.153:5060
From: sip:ming@csmsc01.cs.ru.ac.za
To: sip:cspw@cspwnb.cs.ru.ac.za;tag=8321234356
Call-ID: 442388787@146.231.26.153
CSeq: 1 INVITE
Content-Length: 0

Notice that there is no SDP inside this message.
The "tag" parameter serves as a general mechanism to distinguish multiple instances of a user who is identified by a single SIP URL.

**4. The 200 OK Message**

This is the response from cspw@cspwnb.cs.ru.ac.za when he decides to accept the call. If he sends a 200 OK response message to the caller; this is how the message will look:

SIP/2.0 200 OK
Via: SIP/2.0/UDP 146.231.26.153:5060
From: sip:ming@csmsc01.cs.ru.ac.za
To: sip:cspw@cspwnb.cs.ru.ac.za;tag=8321234356
Contact: sip:cspw@146.231.29.63:5060
Call-ID: 442388787@146.231.26.153
CSeq: 1 INVITE
Content-Type: application/sdp
Content-Length: 137

v=0
o=g9610645 877469378001 1008248918 IN IP4 146.231.29.63
s=testing proxy
c=IN IP4 146.231.29.63
t=0 0
m=audio 32770 RTP/AVP 0

Since proxies can fork requests, the same request can reach multiple instances of a user (mobile and home phones, for example). As each device responds, there needs to be a means to distinguish the responses from each device at the caller's end. The tag in the "To" header field serves to distinguish responses at the UAC.

The SDP in this message contains the codecs that the callee is willing to use for the session. These codecs are based on the SDP sent by the caller, meaning that the callee must choose from the codecs that the caller is able to handle. If there are no codecs that the callee is able to handle, the callee responds with a 400 Bad Request response message with a "Warning: 304" header field.

**5. The ACK Message**

This is the ACK response message from the caller, to indicate to the callee that it has received the 200 OK message.

ACK sip:cspw@cspwnb.cs.ru.ac.za SIP/2.0
Via: SIP/2.0/UDP 146.231.26.153:5060
From: sip:ming@csmsc01.cs.ru.ac.za
To: sip:cspw@cspwnb.cs.ru.ac.za;tag=8321234356
Call-ID: 442388787@146.231.26.153
CSeq: 1 ACK
Content-Length: 0

**6. Media flow**

The media can now be flowing between the two participants for the session because the session is now set up between the caller and the callee.

**The termination sequence—**

**7. The BYE Message**

In the example, callee cspw wants to terminate the session so he sends a BYE message.

BYE sip:ming@csmsc01.cs.ru.ac.za SIP/2.0
Via: SIP/2.0/UDP 146.231.29.63:5060
From: sip:cspw@cspwnb.cs.ru.ac.za;tag=8321234356
To: sip:ming@csmsc01.cs.ru.ac.za
Call-ID: 442388787@146.231.26.153
CSeq: 1 BYE
Content-Length: 0

**8. The second 200 OK Message**

The caller must respond to the BYE message with a 200 OK message.

SIP/2.0 200 OK
Via: SIP/2.0/UDP 146.231.29.63:5060
From: sip:cspw@cspwnb.cs.ru.ac.za;tag=8321234356
To: sip:ming@csmsc01.cs.ru.ac.za
Call-ID: 442388787@146.231.26.153
CSeq: 1 BYE
Content-Length: 0

The above scenario involved a simple call flow between two endpoints that knew the locations and IP addresses of each other. In instances where endpoints do not know each other's IP addresses, and also require the use of a proxy server, more messages are naturally required.

## 2.4 SIP message overview

SIP is a text-based protocol and uses the ISO 10646 character set in UTF-8 encoding. SIP is described in RFC2543 in Augmented Backus-Naur Form (ABNF). Figure 2.2 emphasizes the fact that a SIP message is either a Request or Response message.

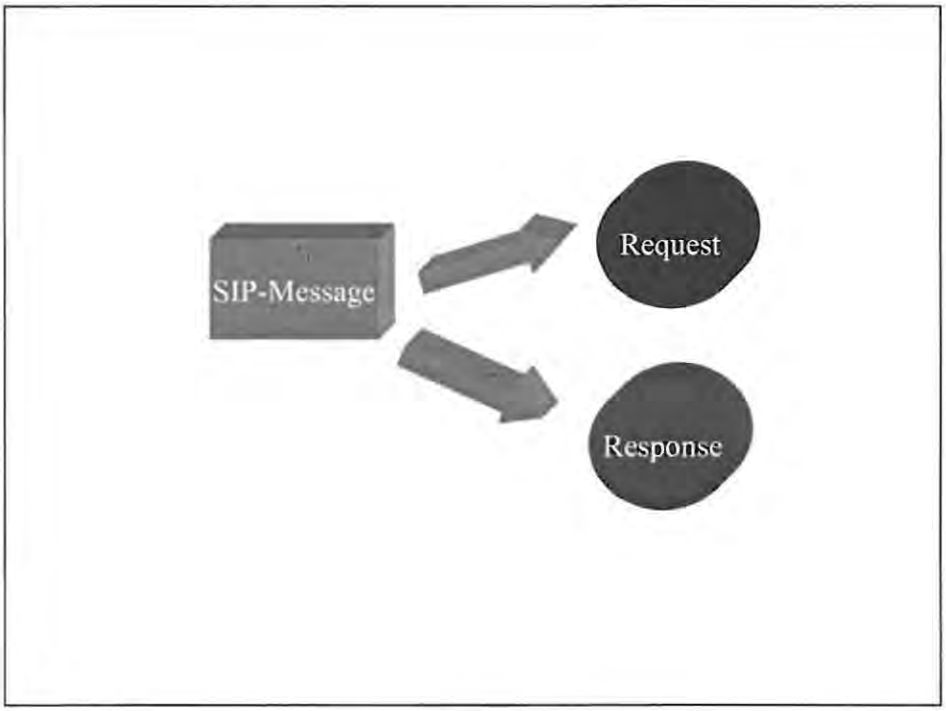


**Figure 2.2 SIP message**

Figure 2.3 shows that the Request and Response messages can be further decomposed. The Carriage Return and Line Feed (CRLF) characters are line terminators used to indicate the end of a line and to separate Request line and Response line information from the message body. The message body is used to indicate the kind of the session that the sender wants to have. It usually contains SDP information.

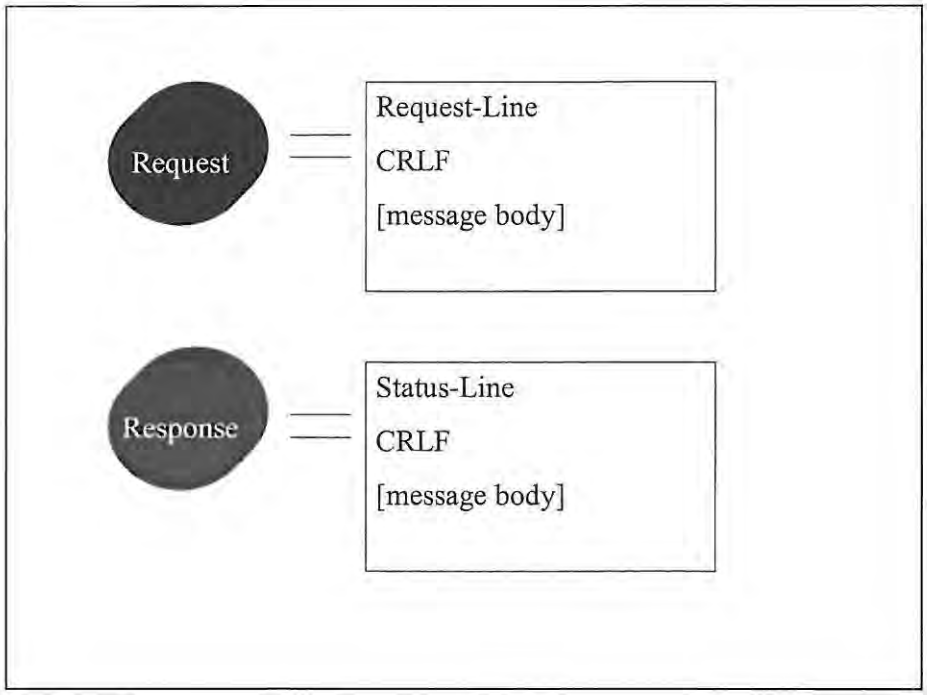


**Figure 2.3 Request and Response messages**

## 2.5 Request messages

In this section Request messages are described in more detail. In Request messages, the Request-Line statement can be further decomposed:

```
Request-Line  =  Method SP Request-URI SP SIP-Version CRLF
```

where Method can be one of the following methods:

```
Method  =  "INVITE" | "ACK" | "OPTIONS" | "BYE"
                    | "CANCEL" | "REGISTER"
```

(SP characters are spacing characters.) The Request-URI is a SIP URL that indicates to which user the request is addressed; it is different from the "To" field in that a Request-URI can be overwritten by proxy servers.

The SIP-Version indicates the version of SIP being used (the current version is "SIP/2.0").

The various methods available in SIP and their functionality are described in the next section.

### 2.5.1 INVITE

To invite a callee to join a session the caller uses the INVITE method. The message body here contains a description of the session that the caller wants to have. The type of media to be used in this session is included in the message body. SIP proxy, redirect and user-agent servers as well as clients must support this method. The following is an example of an INVITE message:

```
INVITE sip:cspw@cspwnb.cs.ru.ac.za SIP/2.0
Via: SIP/2.0/UDP 146.231.26.153:5060
From: sip:ming@csmsc01.cs.ru.ac.za
To: sip:cspw@cspwnb.cs.ru.ac.za
Contact: sip:ming@146.231.26.153:5060
Call-ID: 442388787@146.231.26.153
CSeq: 1 INVITE
Subject: testing
Expires: 3600
Content-Type: application/sdp
Content-Length: 126
"Below contains the SDP"
```

### 2.5.2 ACK

The ACK method is used to indicate acknowledgment to the callee that the "200 OK" response message has been received. The ACK method is used only in INVITE requests.

SIP proxy, redirect and user-agent servers as well as clients must support this method. Below is an example of an ACK message:

```
ACK sip:cspw@cspwnb.cs.ru.ac.za SIP/2.0
Via: SIP/2.0/UDP 146.231.26.153:5060
From: sip:ming@csmsc01.cs.ru.ac.za
To: sip:cspw@cspwnb.cs.ru.ac.za;tag=8321234356
Call-ID: 442388787@146.231.26.153
CSeq: 1 ACK
Content-Length: 0
```

### 2.5.3 OPTIONS

The OPTIONS method is used to query the server of its capabilities. A potential caller can use this method to determine the capabilities of the callee before sending the actual INVITE message. A callee can respond with a 200 OK containing the SDP of the capabilities that it can manage, so that the caller can predetermine the INVITE with the correct SDP to send to the callee [Collins, 2000]. Proxy and redirect servers simply forward the request without indicating their capabilities. SIP proxy, redirect and user-agent servers, registrars and clients must support this method.

Below is one example of an OPTIONS message:

```
OPTIONS sip:cspw@cspwnb.cs.ru.ac.za SIP/2.0
Via: SIP/2.0/UDP csmsc01.cs.ru.ac.za
From: sip:ming@csmsc01.cs.ru.ac.za
To: sip:cspw@cspwnb.cs.ru.ac.za
Call-ID: 442388787@146.231.26.153
CSeq: 1 OPTIONS
Accept: application/sdp
Content-Length: 0
```

### 2.5.4 BYE

The BYE method is used to terminate a session. The person receiving the BYE message should cease transmitting media to the person sending the BYE message. This method must be supported by proxy servers and should be supported by redirect and user-agent SIP servers.

Below is an example of a BYE message:

```
BYE sip:ming@csmsc01.cs.ru.ac.za SIP/2.0
Via: SIP/2.0/UDP 146.231.29.63:5060
From: sip:cspw@cspwnb.cs.ru.ac.za;tag=8321234356
To: sip:ming@csmsc01.cs.ru.ac.za
Call-ID: 442388787@146.231.26.153
CSeq: 1 BYE
Content-Length: 0
```

### 2.5.5 CANCEL

The CANCEL method can be used to cancel pending requests. For example, SIP proxy servers can fork different INVITE requests for the various locations at which a callee can be located. Once the callee has accepted the call from a particular location, the proxy server can cancel the requests to the other locations. This method must be supported by proxy servers and by all other SIP server types.

The following is an example of a CANCEL message:

```
CANCEL sip:cspw@cspwnb.cs.ru.ac.za SIP/2.0
Via: SIP/2.0/UDP csmsc01.cs.ru.ac.za:5060
From: sip:ming@csmsc01.cs.ru.ac.za
To: sip:cspw@cspwnb.cs.ru.ac.za
Call-ID: 442388787@146.231.26.153
CSeq: 1 CANCEL
```

Content-Length: 0

### 2.5.6 REGISTER

This method is used to register a person at a particular location. The "To" header field (SIP URL) is the SIP address used to register the registrant. The SIP URL shown in the message below shows another form that SIP URLs can take. It is possible for a visitor to register for a different network using the Request-URI (Uniform Resource Indicator) field. The Request-URI field is the field following the REGISTER characters. In the following REGISTER message the Request-URI field is sip:cssip.cs.ru.ac.za. The RFC recommends that all SIP servers support this method.

Here is an example of a REGISTER message:

REGISTER sip:cssip.cs.ru.ac.za SIP/2.0
Via: SIP/2.0/UDP csmsc01.cs.ru.ac.za:5060
From: Ming Hsieh <sip:ming@csmsc01.cs.ru.ac.za>
To: Ming Hsieh <sip:ming@csmsc01.cs.ru.ac.za>
Call-ID: 442388787@146.231.26.153
CSeq: 1 REGISTER
Contact: <sip:ming@146.231.26.153>
Content-Length: 0

## 2.6 Response messages

In this section we describe Response messages in more detail.
The Status-Line statement in Response messages can be further decomposed:

```
Status-Line  =  SIP-version SP Status-Code SP Reason-Phrase
                  CRLF
```

The Reason-Phrase can be any humanly readable phrase that describes the status-code. The Reason-Phrase is intended to give a short textual description of the Status-Code. This

is used to provide information to the caller that concerns his request, whether it was successful or unsuccessful, and the reason for failure in the case of the latter.

The Status-Code can be further divided into:

```
Status-Code     =  Informational
                   |  Success
                   |  Redirection
                   |  Client-Error
                   |  Server-Error
                   |  Global-Failure
                   |  Extension-code
```

The Status-Code is a 3-digit integer result code that indicates the outcome of the attempt to understand and satisfy the request. The Status-Code works in ranges; for example, the 1xx series means values from 100-199.

Below is an explanation of the different ranges and their usages:

1xx: Informational -- request received, continuing to process the request.
2xx: Success -- the action was successfully received, understood, and accepted.
3xx: Redirection -- further action needs 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 fulfill an apparently valid request.
6xx: Global Failure -- the request cannot be fulfilled at any server.

## 2.7 Header field definitions

Table 2.1 shows the different categories of header fields that are available for use in the message structure of SIP.

**Table 2.1 Header fields**

| 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 | | | |

The "General-header" fields apply to both request and response messages. The Call-ID and the CSeq fields are closely related and are important to the signaling. Call-IDs are used to identify calls while CSeqs are used to identify transactions within calls. For example, a caller who might want to change the media being used in a call, rather than terminate the call and make a new call, can "re-INVITE" the callee with the same Call-ID but with a different CSeq and a different SDP.

The "Entity-header" field defines meta-information about the message-body or, if no body is present, about the resource identified by the request.

The "Request-header" field allows the client to pass additional information about the request, and about the client itself, to the server.

The "Response-header" field allows the server to pass additional information about the response, which otherwise cannot be placed in the Status-Line. The Warning field is used to carry additional information about the status of a response. It consists of a 3-digit code followed by some warning text. The digits start with "3" to indicate that the warning is SIP related. Table 2.2 shows some warning codes that can be used in the Warning field.

**Table 2.2 Warning codes**

| |
|---|
| 300 Incompatible network protocol:<br>One or more network protocols contained in the session description are not available. |
| 301 Incompatible network address formats:<br>One or more network address formats contained in the session description are not available. |
| 302 Incompatible transport protocol:<br>One or more transport protocols described in the session description are not available. |
| 303 Incompatible bandwidth units:<br>One or more bandwidth measurement units contained in the session description were not understood. |
| 304 Media type not available:<br>One or more media types contained in the session description are not available. |
| 305 Incompatible media format:<br>One or more media formats contained in the session description are not available. |
| 306 Attribute not understood:<br>One or more of the media attributes in the session description are not supported. |
| 307 Session description parameter not understood:<br>A parameter other than those listed above was not understood. |
| 330 Multicast not available:<br>The site where the user is located does not support multicast. |
| 331 Unicast not available:<br>The site where the user is located does not support unicast communication (usually due to the presence of a firewall). |
| 370 Insufficient bandwidth:<br>The bandwidth specified in the session description or defined by the media exceeds that known to be available. |
| 399 Miscellaneous warning:<br>The warning text can include arbitrary information to be presented to a human user, or logged.<br>A system receiving this warning MUST NOT take any automated action. |

## 2.8 Status-code definitions

Table 2.3 lists some status codes that can be used in SIP Response messages.

**Table 2.3 Status codes**

| INFORMATIONAL | SUCCESS | REDIRECTION | CLIENT ERROR | SERVER ERROR | GLOBAL FAILURE |
|---|---|---|---|---|---|
| "100" Trying<br>"180" Ringing<br>"181" Call Being Forwarded<br>"182" Queued | "200" OK | "300" Multiple Choices<br>"301" Moved Permanently<br>"302" Moved Temporarily<br>"303" See Other<br>"305" Use Proxy<br>"380" Alternative Service | "400" Bad Request<br>"401" Unauthorized<br>"402" Payment Required<br>"403" Forbidden<br>"404" Not Found<br>"405" Method Not Allowed<br>"406" Not Acceptable<br>"407" Proxy Authentication Required<br>"408" Request Timeout<br>"409" Conflict<br>"410" Gone<br>"411" Length Required<br>"413" Request Message Body Too Large<br>"414" Request-URI Too Large<br>"415" Unsupported Media Type<br>"420" Bad Extension<br>"480" Temporarily Not Available | "500" Internal Server Error<br>"501" Not Implemented<br>"502" Bad Gateway<br>"503" Service Unavailable<br>"504" Gateway Timeout<br>"505" SIP Version Not Supported | "600" Busy Everywhere<br>"603" Decline<br>"604" Does Not Exist Anywhere<br>"606" Not Acceptable |

Status codes can be used with header fields to convey important session information to the caller. For example, if the callee is unable to accept the call due to the media types wanted by the caller, the callee can respond with a "400" Bad Request message (see Table 2.3), and with the 304 Warning header field (see Table 2.2), which means the requested media type is not available.

The Status-Code "400" Bad Request message can be used to indicate a malformed header field in the request message, but it can also be used indicate that the media type in the message body is not available by using the Warning header field.

The Status-Code "415" Unsupported Media Type does not indicate the same as above, but rather indicates that the message body is in a format that the user-agent server does not understand. The UAS returns this Status-Code along with the Accept, Accept-Encoding and Accept-Language header fields to indicate to the UAC what it can accept. The message body of a request can be encoded in a format that the UAS does not understand; thus, the UAS must indicate to the UAC what encoding it can accept via the Accept-Encoding header field.

## 2.9 Summary

This chapter has presented a brief overview of how the Session Initiation Protocol (SIP) operates, summarizing the basic structure of the protocol and explaining the various header fields and status codes.

# Chapter 3 Two SIP Architectures

## 3.1 Introduction

During the search for suitable SIP architectures to employ for the research, two main architectures appeared: the CINEMA architecture from Columbia University and the VOCAL architecture from Vovida. Later, the SIPTREX™ system based on the JAIN™ SIP Specifications from the Java™ Community Process appeared appropriate, but that system was not investigated due to its newness. Below, a brief introduction to the two chosen environments is given followed by a discussion of their various components.

## 3.2 The CINEMA environment

The SIP protocol was co-authored by Prof H. Schulzrinne at Columbia University in New York City. Columbia first developed a SIP stack, server and user agent. This was developed as part of Columbia InterNet Extensible Multimedia Architecture (CINEMA), which includes a voicemail server, a conferencing server and a SIP-H.323 translation server. The stack and server were written in C language and the user agent was written in Tcl/Tk. The SIP server has a combination of functions: it is a Proxy, Redirect and Registrar server. Figure 3.1 is a diagram of the CINEMA environment.

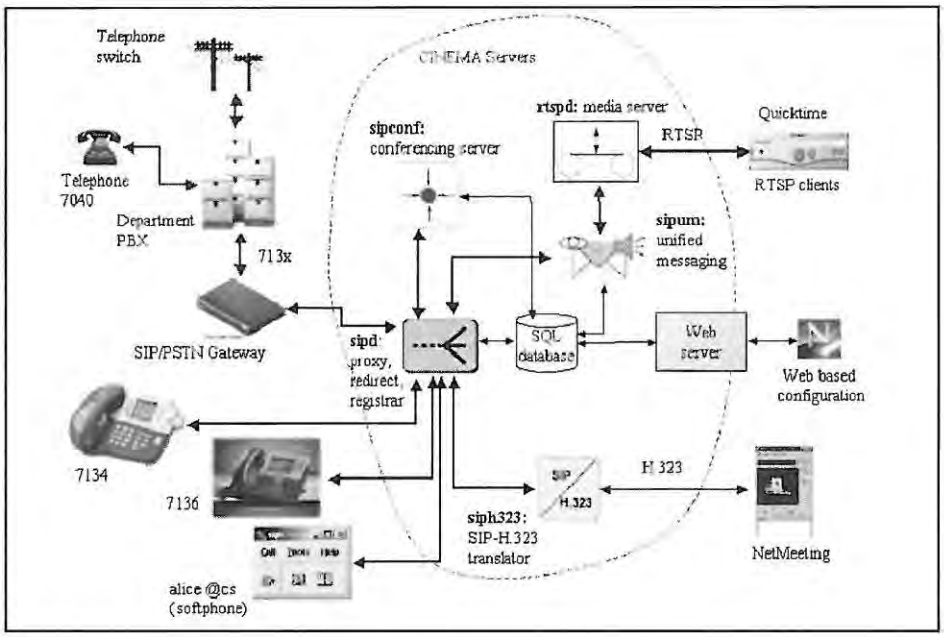


**Figure 3.1 CINEMA environment [Jiang et al., 2002]**

### 3.2.1 Installation process

Since CINEMA's stack and server were written in C language they could be compiled under Windows and Linux. The server was installed on an Intel Pentium II 450Mhz machine running Windows 2000 server edition. The standard name of the server is `sipd` (SIP Daemon server). Columbia University also provided a web interface to this system, in order to administer users and the server remotely. User agents are available for both Windows and Linux. The installation process for the user agents was quite simple, as it was automated. The standard name of the user agent is `sipc`. The interface for the user agent is shown in Figure 3.2.

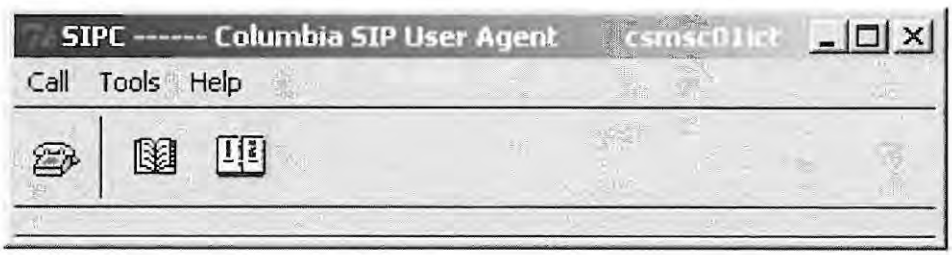


**Figure 3.2 User interface for the CINEMA SIP user agent**

The setup process for the server involved three phases: compilation, installation and testing. The compilation phase required knowledge of Microsoft Visual C++ 6.0. The installation phase required knowledge of the MySql database and Apache web servers. A database was created to store the various locations at which a user can be located. Users were added to the system using the program "addsipuser". Once a user is added to the system, the user can use his `sipc` user agent to register his location with the server. The Apache web server was set up to give users the ability to modify their contact information and to give the administrator access to modify the configurations of the SIP server. The testing phase consisted of running the system and making sure that all components worked properly.

Figure 3.3 shows the web interface provided by CINEMA for administering the users and also the configuring of the SIP server.


**Figure 3.3 Web interface for the CINEMA SIP server**

The sipd server is run from the command prompt, with different options such as "-v" for verbose output. sipd server is a proxy, redirect and registration server all in one and no extra options are required to set the mode in which it runs.

### 3.2.2 User agent and Register server interaction

This section demonstrates successful registration of sipc with the Register server. The Register server in this case is sipd. Figure 3.4 shows the messages required in order for a user to register his location with the sipd server. In this diagram Ming is located at csmsc01ict.cs.ru.ac.za and the Registrar is at cssipict.cs.ru.ac.za.

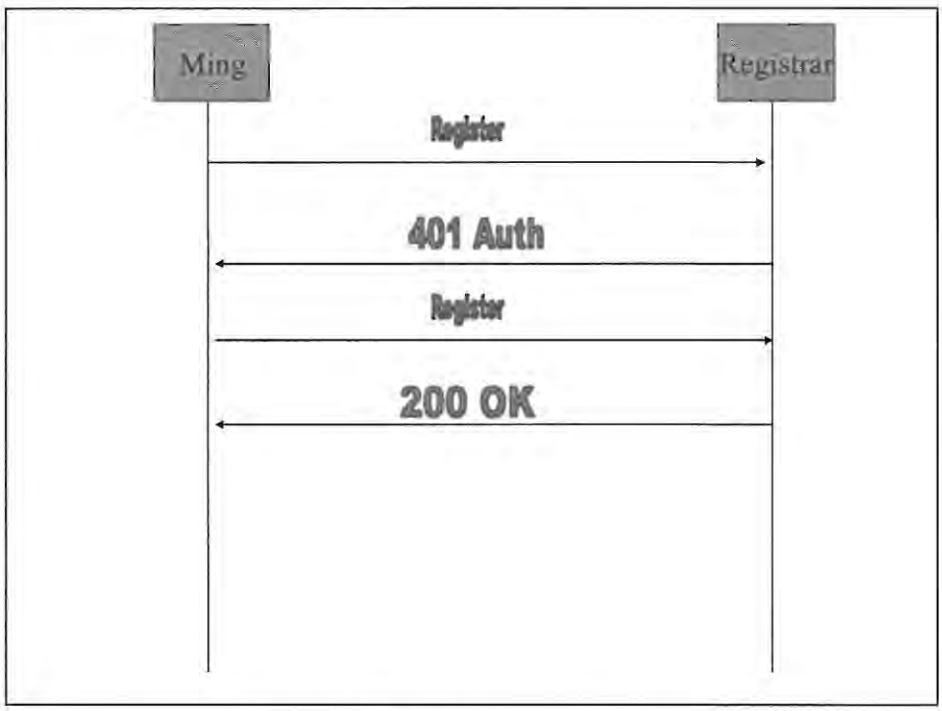


**Figure 3.4 Register sequence diagram**

SIP provides authentication mechanisms similar to that of HTTP. The sipd server provides Basic authentication via the Authentication header field in the SIP REGISTER message. In the example, Ming, the user, tries to register with the server cssipict.cs.ru.ac.za; the server asks for a username and password by sending a "401

Authentication required" response back to the user Ming. This is shown below in the log files for the user agent and the register server.

**User Agent Log File**

10/28/2002 12:13:27.353999
Sent to: cssipict.cs.ru.ac.za:5060

REGISTER sip:csmsc01ict.cs.ru.ac.za SIP/2.0
Via: SIP/2.0/UDP 146.231.123.15:5060
CSeq: 1 REGISTER
Expires: 3600
Contact: sip:ming@146.231.123.15:5060;q=0.1;action=proxy
From: sip:ming@csmsc01ict.cs.ru.ac.za
Authorization: Basic bWluZ0Bjc21zYzAxaWN0LmNzLnJ1LmFjLnphOg==
Date: Mon, 28 Oct 2002 10:13:27 GMT
Call-ID: 770215716@146.231.123.15
To: sip:ming@csmsc01ict.cs.ru.ac.za
Content-Length: 0

The Basic authentication scheme used here is similar to the Basic authentication scheme for HTTP. The Basic authentication scheme is based on the idea that the user must provide a username and password for each realm that he wants to be authenticated with [Berners-Lee et al., 1997]. The realm is the domain with which the user wants to be registered, and in this case Ming wants to register with the realm cs.ru.ac.za. The Authorization header field has the form:

Authorization: Basic SP basic-cookie

The basic-cookie is a base64 encoded string of the username and password.

10/28/2002 12:13:27.554000
Recv from: 146.231.121.142:1336

SIP/2.0 401 Must authenticate with username ming@cssipict.cs.ru.ac.za
Via: SIP/2.0/UDP 146.231.123.15:5060

From: sip:ming@csmsc01ict.cs.ru.ac.za
To: sip:ming@csmsc01ict.cs.ru.ac.za
Call-ID: 770215716@146.231.123.15
CSeq: 1 REGISTER
Date: Mon, 28 Oct 2002 10:15:07 GMT
Server: Columbia-SIP-Server/1.0
Content-Length: 0
WWW-Authenticate: Basic realm="cssipict.cs.ru.ac.za"

In the previous REGISTER message, Ming tried to register with the realm csmsc01ict.cs.ru.ac.za. Because the server does not recognize this realm, it sends a "401" response with a challenge via the WWW-Authenticate header field, together with the correct realm that the user must register with.

10/28/2002 12:13:38.559999
Sent to: cssipict.cs.ru.ac.za:5060

REGISTER sip:csmsc01ict.cs.ru.ac.za SIP/2.0
Expires: 3600
Authorization: Basic bWluZ0Bjc3NpcGljdC5jcy5ydS5hYy56YTptaW5nAA==
To: sip:ming@csmsc01ict.cs.ru.ac.za
Call-ID: 770215717@146.231.123.15
Via: SIP/2.0/UDP 146.231.123.15:5060
From: sip:ming@csmsc01ict.cs.ru.ac.za
Contact: sip:ming@146.231.123.15:5060;q=0.1;action=proxy
CSeq: 1 REGISTER
Date: Mon, 28 Oct 2002 10:13:38 GMT
Content-Length: 0

In this REGISTER message the user Ming registers with the username ming@cssipict.cs.ru.za and password ming. Notice that this has changed the Authorization field from:

"Authorization: Basic bWluZ0Bjc21zYzAxaWN0LmNzLnJ1LmFjLnphOg=="

to:

"Authorization: Basic bWluZ0Bjc3NpcGljdC5jcy5ydS5hYy56YTptaW5nAA=="

The Authorization field changed due to the base64 encoding.

10/28/2002 12:13:38.75
Recv from: 146.231.121.142:1336

SIP/2.0 200 OK
Via: SIP/2.0/UDP 146.231.123.15:5060
From: sip:ming@csmsc01ict.cs.ru.ac.za
To: sip:ming@csmsc01ict.cs.ru.ac.za
Call-ID: 770215717@146.231.123.15
CSeq: 1 REGISTER
Date: Mon, 28 Oct 2002 10:15:18 GMT
Server: Columbia-SIP-Server/1.0
Content-Length: 0
Contact: <sip:ming@csmsc01ict.cs.ru.ac.za>; expires="Mon, 28 Oct 2002 11:26:37 GMT"; action=proxy; q=1.00
Contact: <sip:ming@146.231.123.15:5060>; expires="Mon, 28 Oct 2002 11:15:18 GMT"; action=proxy; q=0.10
Expires: Mon, 28 Oct 2002 11:15:18 GMT

Once the user is authenticated, the server sends a "200 OK" message back to the user. It also lists the current available contacts at which the user has registered. The Contact header field also contains the date and time when this contact will expire. The "q=0.10" indicates the preference for where the user would like to be contacted. The "q" field is a decimal number between 0 and 1. Higher values indicate a higher preference for locations where the user would like to be contacted.

The log file for the Registrar server side is the same in terms of the SIP messages received and sent. The log file is given in Appendix A.

### 3.2.3 User agent and Proxy server interaction

This section explains the interaction between user agents and proxy servers. The example depicts a successful call setup between two user agents, with the Proxy server as an intermediate server needed to locate a user and for sending the INVITE on behalf of the user. Figure 3.5 shows messages exchanged between those three entities.

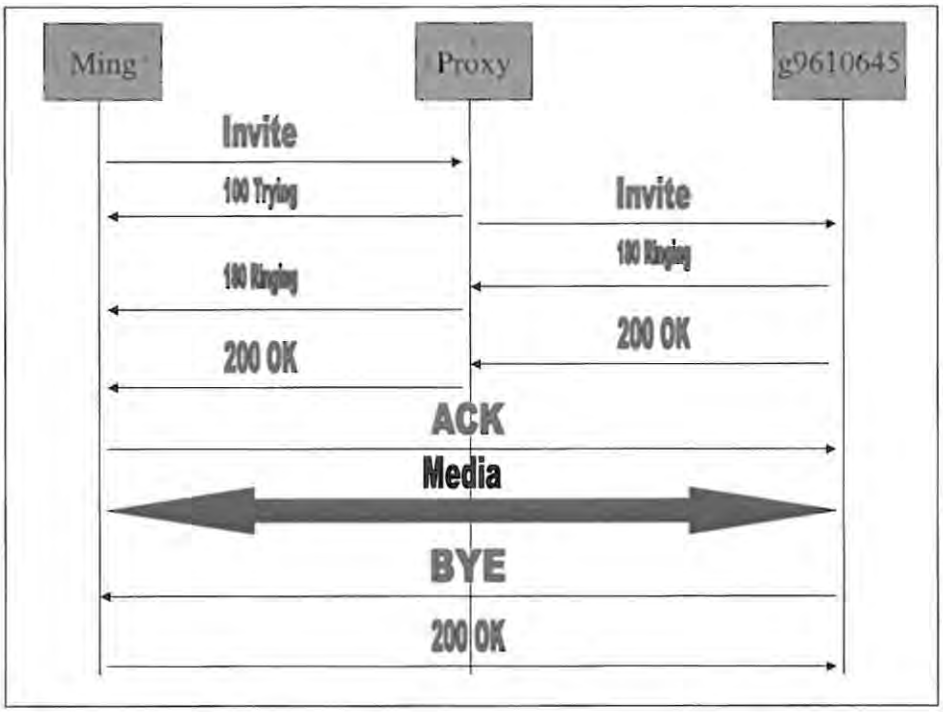


**Figure 3.5 User agents and Proxy server call setup**

The actual SIP messages are given in Appendix B. A notable feature here is that the ACK message sent from Ming to g9610645 does not go through the Proxy server. One reason is because, at this moment, Ming knows the location of g9610645, so he does not need the Proxy server to send the ACK. Another reason is that the Proxy server is operating in the stateless mode; that is, once it has set up the call, the rest of the transaction mechanism is up to the user agents. Alternatively, stateful proxies keep track of the entire transaction. Although this can be resource intensive when scaling the system to a large base of users, it can be useful in services such as billing.

## 3.3 The VOCAL environment

VOCAL (Vovida Open Communication Application Library) is an open-source application suite created by Vovida Networks and eventually acquired by Cisco Systems in November 2000. Cisco concurs with the open-source philosophy of Vovida and continues to support the Vovida Open-Source Initiative.

The philosophy behind open protocols and open-source applications is the idea that customers are not bound by the constraints of proprietary software as characteristic of closed protocols. Open protocol and open source also means flexibility, which is especially needed in times of rapid change [Vovida, 2003].

VOCAL consists of three layers, namely *Call Control and Switching*, *Operation System Support* and *Services and Feature Creation*:

- *Call Control and Switching* handles user registration, call initiation, call modification and call termination.
- *Operation System Support* handles provisioning, network monitoring and billing information.
- *Services and Feature Creation* contains features such as call forwarding, call blocking, call transfer and call waiting.

The basic components needed to set up a VOCAL system are the *Marshal server*, *Redirect server*, *Feature server* and *User Agents*:

- The *Marshal server* is an entry point into VOCAL. Marshal servers provide the logical functions of the SIP proxy server and the SIP registration server.
- The *Redirect server* redirects the SIP requests depending on the features enabled by the Feature servers. The Redirect server also provides Registration, Redirection and Location services and functions. While the Marshal server provides the logical functions for the registration process, the Redirect server

provides the physical functions, i.e., the Redirect server stores the actual location of the user in its database.

- The *Feature server* is the service logic center for determining what services are available to each user. There is a Feature server for each feature or service available on the network. The Feature server provides features such as call forwarding, call blocking and call return.
- *User Agents*: The initial functionality of the user agent was to test the SIP stack that was developed in VOCAL. Slowly this progressed to a functional user agent with limited capabilities. At the moment the user agent is able to make and receive calls with limited audio support. The user agent is also able to register with a registration server and use a proxy server to make calls.

Figure 3.6 is a diagram of the VOCAL environment.

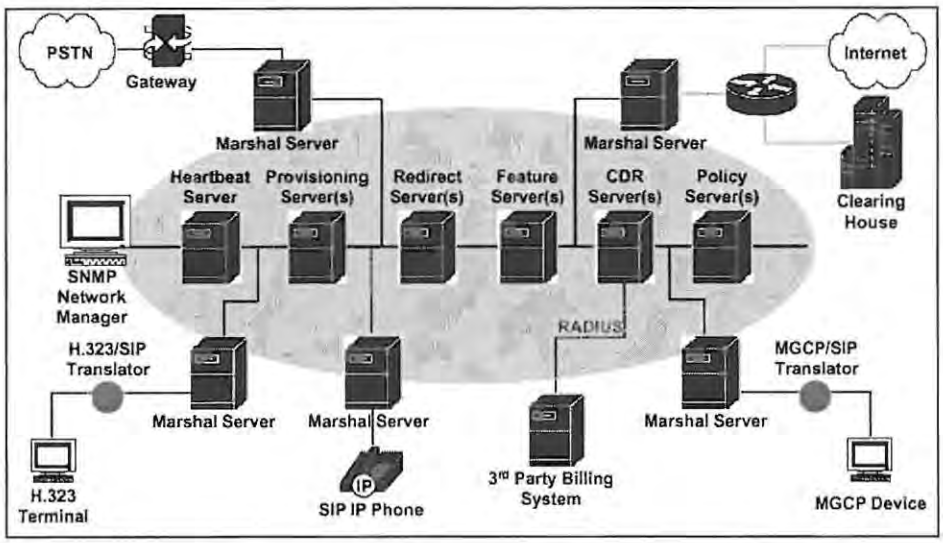


**Figure 3.6 VOCAL environment [Vovida, 2001a]**

Although the VOCAL architecture may seem complex, not all of the usual components are required in order to set up a basic working SIP network. The components that were used to install a basic SIP network at the Department of Computer Science are a Feature server (`fs`), Redirect server (`rs`), Provisioning server (`ps`), and Marshal server (`ms`); User Agents (`ua`) were also made available to the users. Redundant Marshal servers were

installed to provide scalability to the system. This required the installation of a Heartbeat server to monitor the availability of each Marshal server.

### 3.3.1 Installation process

The version of VOCAL used for this project is VOCAL-1.3.0. The hardware requirements for this version are:

- 700 Mhz, Pentium III Intel-based PC
- 512 MB RAM
- 1 GB hard disk space.

The software requirements for this version are:

- Red Hat Linux 6.2 or later version
- Apache web server
- JDK 1.2
- Web browser with Java 2 Runtime Environment Plug-in enabled.

The actual installation was on:

- 333 Mhz, Pentium II
- 256 MB RAM
- 2 GB hard disk space,

with the following software installed:

- Red Hat Linux 7.1
- Apache web server 1.3.19
- JDK 1.3.1
- Netscape browser with Java Runtime Environment 1.3.1.

The VOCAL software suite is provided open source, which requires the GNU Compiler Consortium (gcc) to compile and install the software. The compiled version of the VOCAL source directory came to 1.5 GB. The whole VOCAL server suite could be installed as a system service and can be started and stopped via the command prompt. Log files for each server are also provided for diagnostics of the system.

### 3.3.2 User agent registration diagram

The registration sequence for registering a user with the system requires the interaction of the Marshal and Redirect servers. The Marshal server is the entry point into VOCAL and it forwards the requests from users to the Redirect server. Figure 3.7 depicts a simple registration sequence with no authentication.

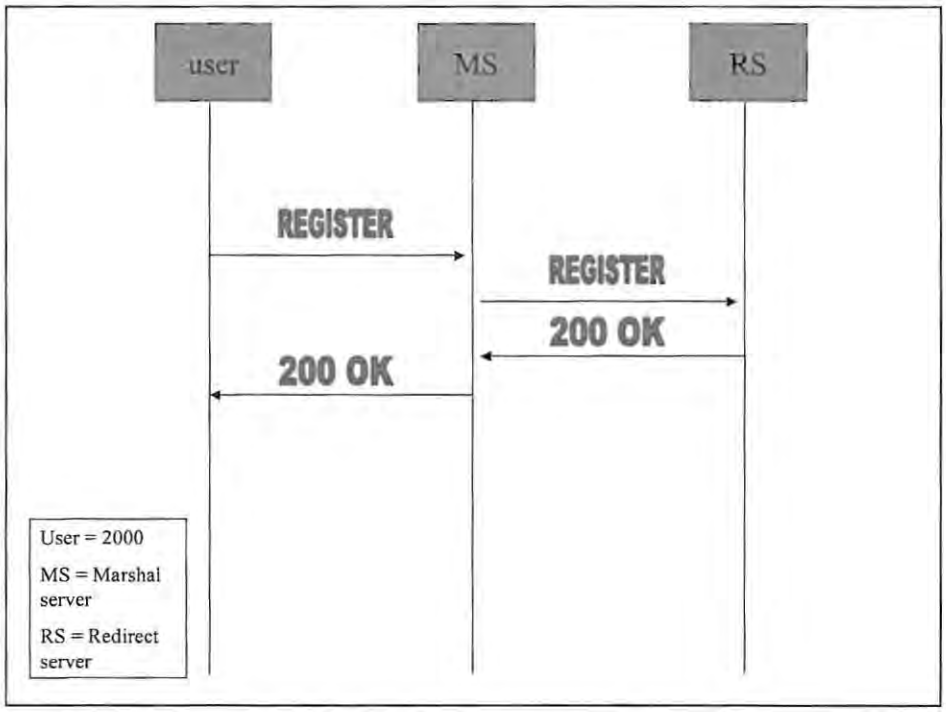


**Figure 3.7 VOCAL registration diagram**

When authentication is involved the Marshal server contacts the provisioning server to get the correct username and password (which is not shown in Figure 3.7). The diagram shows the REGISTER message received by the Marshal server. The Marshal server forwards this message to the Redirect server, which stores the registration information in its database. The Redirect server returns a "200 OK" message to indicate that the registration was successful.

This registration sequence is different from that of CINEMA. That is, in VOCAL more messages are exchanged in order to register a user, because of the separation of the registration and redirect server.

### 3.3.3 User agent simple call diagram

Once the user is registered with the system he or she is able to make calls to other users. Figure 3.8 shows a basic call between two users using VOCAL. The user agents are able to communicate with each other without the Marshal server (point-to-point communication) if the users know each other's location beforehand.

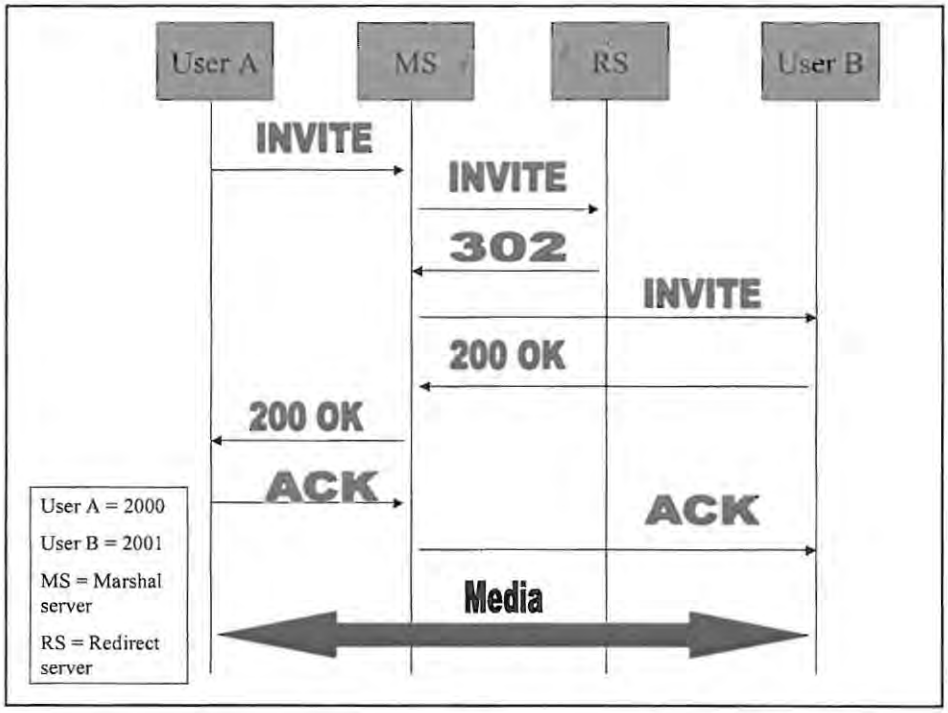


**Figure 3.8 VOCAL simple call flow**

Users in VOCAL are assigned telephone numbers. For example, user Alice gets the number 2000 and user Bob gets the number 2001. The Marshal server forwards the INVITE message from Alice to the Redirect server. The Redirect server checks its database for possible locations of Bob and forwards this information to the Marshal

server via a contact header field in a "302 Moved Temporarily" SIP message. The Marshal server uses that information to forward the INVITE to Bob. The "100 Trying" and the "180 Ringing" messages are not shown in the diagram, although, for the purpose of the example, they are being sent by the Marshal server and user Bob, respectively.

Note that in VOCAL the "200 OK" and the "ACK" messages still pass through the Marshal server. This is different from what happens in CINEMA, wherein users can directly send this information to each other, bypassing the proxy server. The same process occurs for the "BYE" and the "200 OK" messages that are used for termination of the session. In CINEMA these messages are sent directly to each other, while in VOCAL these messages still pass through the Marshal server. More examples of call flows with different call scenarios are available in [Vovida, 2001a] and [Johnston, 2001].

## 3.4 Contrasting the architectures

As one should expect, the CINEMA architecture and the VOCAL architecture have strong similarities. Below, we compare their components.

**Registrar, Redirect and Proxy servers**:

The two architectures contain a server, or servers, for registration, redirection and proxy functionalities. The central SIP server (`sipd`) in CINEMA contains the functionalities of the Registrar, Redirect and Proxy servers and is similar to the combination of the Marshal server and Redirect server in VOCAL. In VOCAL the registrar and proxy functionalities are contained in the Marshal server, while the redirect functionality is naturally placed in the Redirect server. Putting the functionalities of registration and proxy into the Marshal server, and that of redirecting into the Redirect server, increases the distributed nature of VOCAL. This means, for example, that the Marshal server and the Redirect server can actually be put on separate machines. In that case, however, more SIP messages will be exchanged between the two servers and so reduce the speed of a call setup. A more detailed discussion of this difference between the two environments is given in Chapter 5.

**Backend databases and web servers**:

The following aspect provides for the storage of usernames and the maintenance of the various servers for the two architectures, which is essential in the operation of a SIP network. In CINEMA the MySql database and the Apache web server are used as the backend storage and web interface configuration mechanism, respectively, while in VOCAL a custom-made database is used for backend storage and Apache is used for the web interface. The databases for both environments are used to store the contact locations of users and also the configurations of the various servers. The web servers of these environments provide an interface to these databases so that administrators of the system can operate remotely and users can modify their settings or profiles.

**User agents**:

As expected, both these architectures provide user agents for users to communicate with. However, the capabilities of their user agents differ. In CINEMA, a user agent called `sipc` has the capability to execute scripts, which provide services. In VOCAL, the user agent called `ua` is unable to execute scripts and at the moment is mostly used as a 'dumb' terminal to make and receive calls. Both systems can interact with any SIP-standard compliant user agent.

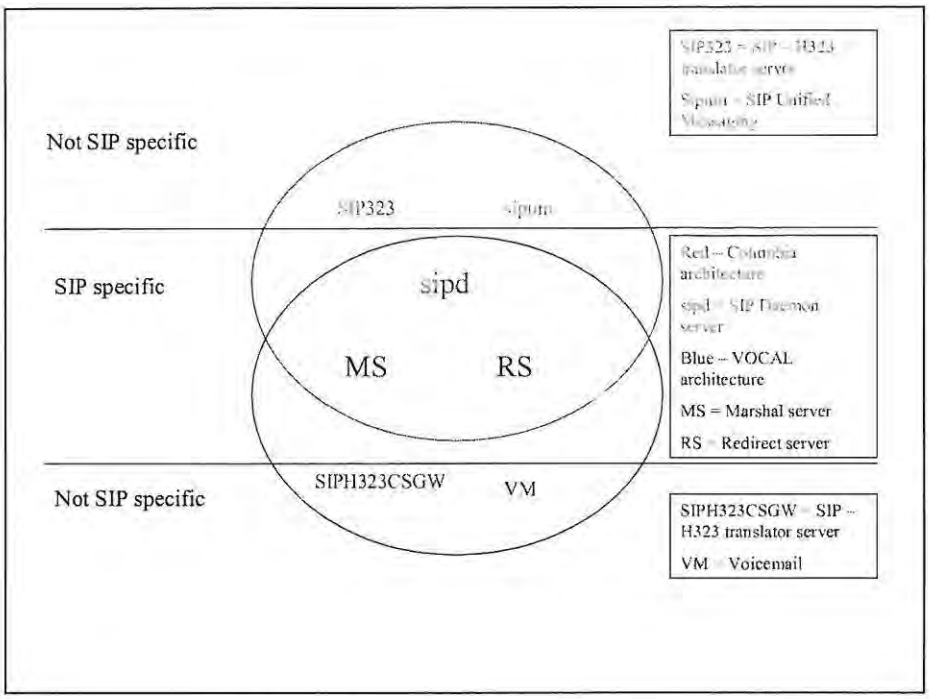


**Figure 3.9 Comparison of architectures**

Figure 3.9 shows the essential SIP-specific similarities for both environments (grouped in the middle). Each environment has various other servers with similar functionalities, such as the H.323 to SIP translators and the voicemail servers; these are not classified as SIP-specific. The essential elements required to set up an operational Internet telephony environment are the SIP-specific elements.

## 3.5 Summary

This chapter introduced the two environments that were used to deploy SIP, namely CINEMA and VOCAL. A detailed explanation was given on how the user agents for the architectures interact with their systems and servers, and a discussion that contrasted the two environments was provided.

# Chapter 4 Service Categories

## 4.1 Introduction

The services chosen for the investigation into service creation in CINEMA and VOCAL are those most commonly experienced by an average PSTN/PABX user, such as call forwarding or voicemail. Services that allow internetworking, and that may not seem to fall into this category, are included because they allow a user to experience the wide range, easy reaching of other users typical of traditional telephony.

The choice of these services is in keeping with the general theme of this thesis. While we do not aim to establish if and why Internet telephony is better than traditional telephony, we are interested in exploring the degree of difficulty in creating services that are seen nowadays as common in traditional telephony in two different, relatively mature SIP environments, CINEMA and VOCAL.

Before investigating service creation it is natural to attempt to categorize the services themselves and this chapter is devoted to this task. A classification is never an easy task and often has some arbitrary aspects. The classification proposed here has helped us to organize our investigative work.

The actual operation of any telephony environment is permitted by mechanisms that can be seen as services, such as the ability to make point-to-point voice calls. We will call these *basic services* and list them in section 4.2. Any other service will be called an *advanced service*. Our classification of advanced services is presented in section 4.3.

## 4.2 Basic Services

Below is a list of CINEMA's basic services:

- *Make calls point-to-point.* Calls can be made using a `sipc` user agent connected to any other SIP user agent. A definite SIP URL is required to make the call (point-to-point). A SIP URL consists of a username plus the machine name of the IP address where the callee can be contacted. A definite SIP URL consists of the username plus the exact machine name where the callee is currently located.
- *Register with the proxy server.* Registration is considered a basic service because it provides a central place whereby parties can inform each other about their location. This is useful, for example, in terms of mobility: each time a person registers from another location the call can be directed to him because the server knows where he can be located. This suggests that each registration must contain an expiry time and allow each user to re-register and de-register from a particular location. The Expires header field in the REGISTER message enables the user to specify the valid duration of his registration. For example:

  Expires: Sun, 01 Dec 2002 16:00:00 GMT

  means the registration is valid until 1 December 2002, 4 p.m. If the user does not specify this field, the server rather will provide and display a default expiry date.
- *Make calls via the proxy server.* The information required to make a call under this scenario is a normal SIP URL where the machine name can be the machine name of the proxy server. Making calls using the `sipd` server provides basic services such as user lookup and user location. For example, it looks up any aliases for the concerned callee and also provides all the possible locations of the callee.

Below is a list of VOCAL's basic services:

- *Make calls point-to-point.* The user needs to know how to modify a configuration file for the user agent in order to make a call from point-to-point in VOCAL.
- *Register with the system via the Marshal server.* This service is essentially the same as the register service in CINEMA described above.
- *Make calls using central resources of the VOCAL system.* To use VOCAL the user needs to register with the system and use the proxy server in the system to make calls. This operation is more difficult than in CINEMA because of the lack

of GUI support, forcing one to use the configuration file for the user agent. Other changes that are required in the configuration file are the dial patterns. The user must know how to modify the dial patterns for the user agent. The user agent in VOCAL is modeled after a normal PSTN phone, whereby specific dial patterns are required in order for the user to dial out. Dial patterns are like the quick dial pads on Plain Old Telephone Service (POTS) phones. For example, one could configure "1#" to a frequently used number. The VOCAL calling system is closely modeled after the traditional telephony calling system wherein a number representing an alias identifies each user. This eases the transition for users who have no prior knowledge of SIP URLs. (VOCAL still uses SIP URLs, and the numbers used in the dial patterns are treated as aliases.) CINEMA does provide dial plans on the server side but those are used only to make calls to the PSTN network.

The services listed in this section are assumed to be available at all times. Figure 3.9 and section 3.4 provide an illustration and contrasts the architectures namely CINEMA and VOCAL.

The user agent from CINEMA has audio, video and data capabilities. The user agent from VOCAL is only audio capable at the moment. To make the comparison between the environments significant, we assume that only audio will be available to the user agents in both environments.

## 4.3 Advanced Services

### 4.3.1 Call-related services

*Call-related services* are services executed during the establishment of a session and do not require the participation of the user during their execution, although user interaction is required to set up the service prior to its execution. Call forwarding, call blocking and

call redirect are examples of call-related services. Services in this category require the signaling capabilities of SIP.

Call forwarding and call blocking are implemented in both CINEMA and VOCAL using scripting languages: SIP-CGI in the case of CINEMA, and CPL in the case of VOCAL, as described in sections 5.3 and 5.4, respectively.

Using a scripting language as the service creation mechanism makes it easy to create call-related services in Internet telephony. In VOCAL the process is further simplified by a GUI interface that allows the user to customize the service according to his needs. This naturally limits flexibility, but this is generally acceptable as long as the limitations are not too extensive. VOCAL's approach also helps deal with the security issues concerning scripting languages (see section 5.6).

### 4.3.2 Interactive services

*Interactive services* are services that require user interaction during the execution of the service. A voicemail service that requires input from the user, such as "Press 1 if you are satisfied with this greeting", is an example of an interactive service. Generally, this type of service has a multimedia nature as well; for example, the voicemail service has to handle G.723 or G.711, the typical audio formats used in telephony, to record voice. However, whether or not the service has a multimedia nature is irrelevant to its categorization. The services in this category also require signaling, but their signaling capability is less important, in this context, than interactivity with the user.

The voicemail services for CINEMA and VOCAL are presented in sections 6.2 and 6.3, respectively. Later in this work the voicemail services are used as a template to develop a reminder service for both environments.

Creating services in this category usually requires intensive modifications of existing components in the system (as demonstrated in sections 6.4 and 6.5 in regard to creating a

reminder service); thus, programming techniques that are more advanced than basic scripting are required to develop interactive services. The service creation process is further complicated by the need to anticipate user behavior and the need for catering for endpoints with different capabilities. Users are generally unpredictable and this unpredictability has to be taken into account by the programmer. Endpoints with different capabilities need to be catered for by redirecting the user either to another user agent with the appropriate capability or by providing an informative message.

### 4.3.3 Internetworking services

*Internetworking services* allow two or more different networks to interoperate and communicate with each other, such as an H.323 network and a SIP network.

Bridging of networks can be done on a services-type level, wherein a gateway is dedicated to each service to be internetworked. For example, where a call-forwarding service is resident on a H.323 network, a gateway could be developed so that SIP users who want to access the service must use that specific gateway. This type of bridging has the advantage of security because it forces SIP user agents to use a specific service-level protocol to access the services on the H.323 network; thus, SIP user agents are not indiscriminately allowed to access other parts of the H.323 network. The disadvantage in this case is that the bridging will not scale well, as more and more services are exposed to the SIP network more and more gateways have to be developed, deployed and maintained. This, combined with the fact that the service must be correctly provisioned to the correct users (so that only designated users are allowed to use the service), makes this solution unviable.

Bridging can also be done on a protocol level whereby a translator is placed in between the two networks and handles each of the calls directly, either from a H.323 or a SIP standpoint. The translator cannot merely convert one H.323 message into a SIP message, but must also remember which messages it has sent, which messages it must receive and to which call the message belongs. This is due to the session nature of both protocols.

This more common approach to internetworking is the approach followed in this thesis. While this approach allows for any session to be established between H.323 and SIP, security is however reduced.

Once an internetworking service was established between H.323 and SIP, we began to investigate whether or not a service that resides on a H.323 network can be accessed from a SIP network. For example, at the Department of Computer Science at Rhodes University, a SIP to H.323 Call Signaling Gateway (`SIPH323CSGW`) was set up so that SIP users could access services on the H.323 network. Hence, services such as the Email reader and the ISDN gateway (`OpenISDNGw`) that were already available on the H.323 network became readily available to the SIP network [Penton et al., 2001b]. For more information about the setup of the H.323 environment see [Penton et al., 2001a] and [Penton and Terzoli, 2002]. Overall, `SIPH323CSGW` acted as a translator that allowed sessions to be established between SIP and H.323. This service allowed users on the SIP network to access the services on the H.323 network and vice versa.

In sections 7.2 and 7.3 we will explore the mechanisms available for internetworking between H.323 and SIP in CINEMA and VOCAL. A MGCP network was also set up using VOCAL [Jacobs and Clayton, 2002]. Next, we used the available MGCP and SIP stack in VOCAL to develop an internetworking service between MGCP and SIP.

Developing, deploying and maintaining internetworking services is generally more difficult than what is required for interactive services. To start with, the developer needs to be knowledgeable of two protocols, and working with two APIs doubles the workload.

### 4.3.4 Hybrid services

*Hybrid services* are services that contain elements from more than one of the categories defined above. For example, a voicemail service that contains a call-forwarding service fits both the interactive and the call-related categories. (As an example: Perhaps one would like phone calls from his boss to be forwarded to his voicemail address and calls

from his friends delivered directly to him.) Hybrid services are not discussed in this thesis but their study is suggested as an extension to the work.

### 4.3.5 Composite services

*Composite services* are services that result from the concatenation of services in one of the categories defined above. A hypothetical example of a composite service may be named CallCatcher, comprising call-blocking and call-barring features: call blocking would stop incoming calls and call barring would prevent outgoing calls. We consider composite services a subcategory of each category of service defined above.

## 4.4 Discussion

Figure 4.1 depicts the classification of advanced services as proposed in this chapter. The size of each region in the diagram does not reflect the number of services in each category, as these can differ substantially. This choice of categorization does not encompass all services, since there may be a service that does not fit in any of the above-mentioned categories. As stated above, the main function of our choice of categories is to organize the particular services created during the course of the research into a conceptually satisfactory manner.

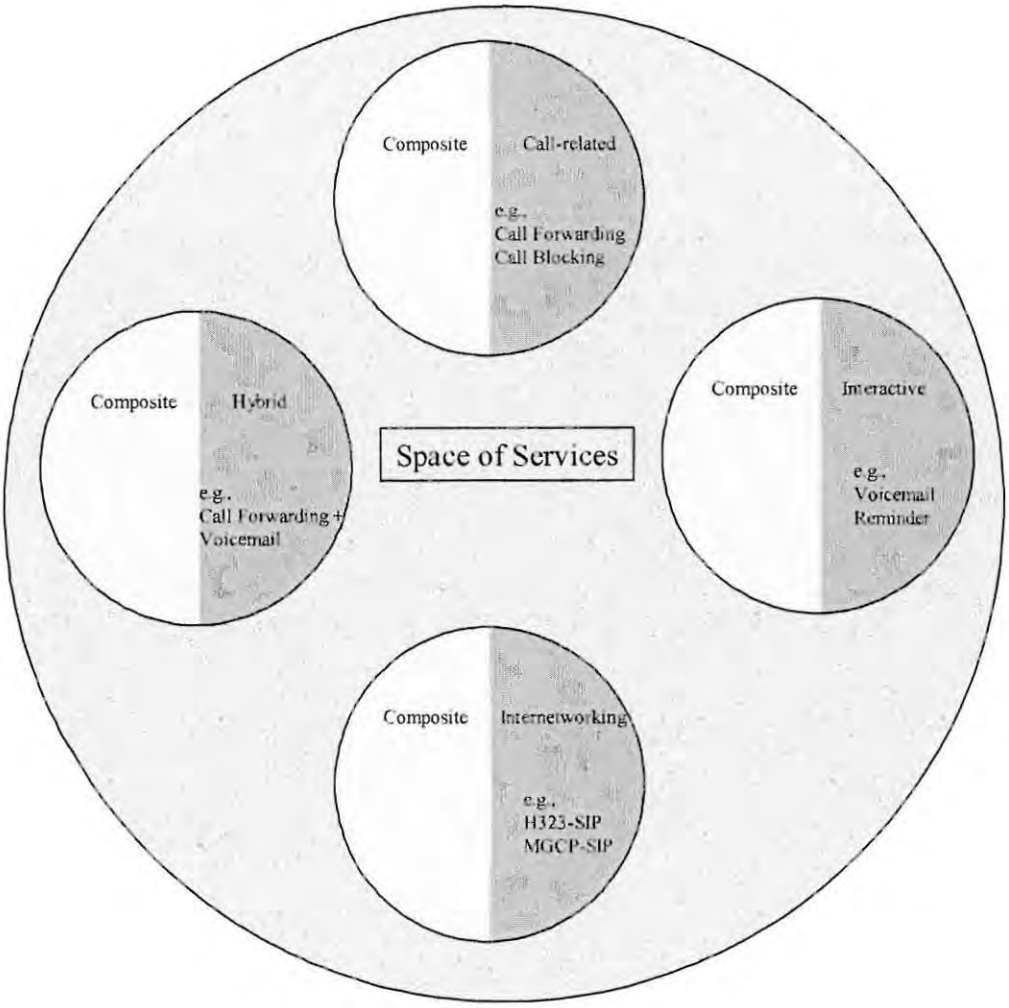


**Figure 4.1 Advanced services chart**

There are other approaches to classifying services. For example, since some services are harder to create than others, one could consider the dimension of complexity as a foundation for classification. Interestingly, that would likely result in essentially the same categories proposed and summarized above (see Figure 4.1): Call-related services are the easiest to structure, requiring only basic knowledge of scripting languages; Interactive services are more difficult to create, due to the unpredictability of user behavior and the need to cater to different end devices; Internetworking services are the most difficult to put together, because a developer must be proficient with two protocols. Even so, we felt

that using complexity as a basis for categorizations was unsuitable, because complexity is transient or relative: once a standard framework is in place for any category of service, service creation complexity will be immediately reduced.

However, we did consider another dimension, *multimedia*, whereby services would be classified according to the amount and type of media support they required. We felt, however, that this did not provide the necessary characterists to separate the different services.

VOCAL categorizes services (called *features*) into *calling* or *called* [Vovida, 2001a]. For example, call forwarding is considered a *called* feature, while call blocking is considered a *calling* feature. Again, we felt that this categorization did not provide the necessary characterists for the organization of the service space explored in this thesis.

CINEMA's categorization of services is, in a sense, founded on a consideration of trust. This is due to the fact that with SIP-CGI one can run malicious scripts on the proxy server, which can disrupt services. The amount of trust given to the service creator varies. Service creators can be the owner of the server, a third-party service provider, or an end-user [Lennox et al., 1999b]. Administrators using SIP-CGI scripts necessarily run mission critical services; services from third-party providers may run some SIP-CGI and CPL scripts, while end-users will only use CPL scripts. Since the same approach to classification did not cater for interactive services, it was not used in the present work.

[Anjum et al., 2001] report another approach to categorization: determining whether the service is public or user specific. For example, the 800-number (toll-free number) is a public service provided through a central server. On the other hand, a speed-dialing service that maps frequently dialed numbers to single digits is a user-specific service and so is provided from the user's terminal. However, a voicemail service, which is both public and user specific, is not accommodated by this categorization.

## 4.5 Summary

There are many services available in telephony today. The categorization of services is regarded as a useful prerequisite to studying service creation in two SIP environments, CINEMA and VOCAL.

We proposed an initial division into two groups, *basic* and *advanced* services. VOCAL similarly splits up services, whereby basic services are known as "SIP-Based Call Control and Switching," and advanced services are known as "Feature and Application Creation" [Vovida, 2001a]. Although this type of partitioning is not evident in CINEMA, the basic services discussed here are assumed to be fundamental to the CINEMA architecture. We view basic services as ones that provide minimal functionality for any telephony environment and are assumed to be available at all times, and advanced services as those that need basic services in order to operate. Consequently, advanced services are the focus of this thesis.

Advanced services have been assigned to various categories: *call-related*, *interactive*, *internetworking* and *hybrid* (see Figure 4.1). Each category includes the subcategory *composite*, where services that are the result of the concatenation of services in that category are grouped.

Finally, a brief discussion explains the choice of categories, including other possible approaches to categorization.

The next three chapters will investigate the three main advanced service categories, namely *call-related*, *interactive* and *internetworking*.

# Chapter 5 Call-related services

## 5.1 Introduction

In this chapter we discuss call-related services as they were defined in Chapter 4. The methods for creating this type of service are standard and widely used in the industry today. The most popular methods for creating call-related services are SIP-CGI, CPL, SIP Servlet API and JAIN™ API. The SIP Servlet API and JAIN API are being developed within the JAVA™ Community Process. Here, we discuss only SIP-CGI and CPL, since these are directly supported by CINEMA and VOCAL, respectively.

## 5.2 SIP-CGI

SIP-CGI is based on the Common Gateway Interface (CGI) for HTTP. CGI has been used widely on the Internet to create services for the World Wide Web. CGI allows users to input data onto a form and send the form to a web server. The web server invokes a program to pass the data acquired from the user. The program will process the data and send a response back to the user via the web server. For example, a form may be used to get the list of fields that a user would like to view, and then pass this information to the CGI script, which uses it to query a backend database. The script will retrieve the information from the database and display it to the user. Web services were once created using CGI. Since SIP is similar to HTTP, CGI was the likely candidate for creating SIP services. SIP-CGI is defined in RFC3050 [Lennox et al., 2001].

Since HTTP is based on a client-server protocol, a HTTP transaction usually consists of a request and a response, while in a SIP transaction many responses can be generated from one request. For instance, a SIP proxy server can proxy an INVITE message to the various locations where a callee can be located. Each of these locations may respond with either 'not here' or 'available and ringing'. The distinction between SIP-CGI and HTTP-CGI is made at the level of the web server and SIP server. Notably, with HTTP-CGI the

spawned script is essentially running another program on the web server and can generate more requests if it wants, but the requests that the spawned script generates need to be handled by the script itself. This is not true for SIP-CGI. In SIP-CGI the script can generate additional requests to various destinations, but the SIP server handles these requests. This distinction is important to note because it confers restrictions on the type of services we can create with SIP-CGI. The particular restriction pertains to the number of requests, for the performance sake of the SIP server.

### 5.2.1 Basic model

SIP-CGI is modeled on the functionality of the SIP server. It tries to model what the SIP server does by using CGI action lines.

The four basic functions of the SIP server are:

1. *Proxying requests*: When the server receives a request from a client, it must decide whether or not to add new headers to the request and provide a list of servers to which it must forward the request, and then do the actual forwarding of the request.

2. *Returning responses*: When the server receives a response from a request that it has proxied, it must decide whether or not to add new headers to the response and then forward the response back to the client where the request originated.

3. *Generating requests*: In this case the request originates from the server. The server must compose the request in its entirety, including the headers and the message body, and then forward the request to another server.

4. *Generating responses*: When the server receives a request from a client it must generate an appropriate response that includes the right headers and message body. Then, the response must be sent back to the client.

### 5.2.2 SIP-CGI actions

The following are brief descriptions of typical SIP-CGI actions. These actions model the functions of the SIP server described above. A SIP server supporting these actions is SIP-CGI compliant.

1. *Proxying requests*: The action line for proxying a request is supported by CGI-PROXY-REQUEST. An example of such an action is:

```
CGI-PROXY-REQUEST sip:ming@csmsc01ict.cs.ru.ac.za SIP/2.0
Contact: sip:g9610645@cspg34.ict.ru.ac.za
```

The example shows a typical output from a SIP-CGI script requesting to proxy the initial request to a different SIP URL. The SIP server will receive this output and process it to determine if it is a valid SIP-CGI output. Once it determines that it is a valid output, the server will proxy this request to ming@csmsc01ict.cs.ru.ac.za, and adds the contact header field with the contact g9610645@cspg34.ict.ru.ac.za. The general form of a proxy request is:

```
Proxy-Request = "CGI-PROXY-REQUEST" SIP-URL SIP-Version
```

The action line is similar to a SIP request line, which simplifies parsing.

2. *Returning responses*: Returning a response is more complex than generating a response because a server can potentially receive more than one response as the result of proxying a request. Some sort of state is needed to maintain control over the SIP transactions. SIP-CGI uses a method similar to the HTTP-CGI cookies system. A unique cookie or token is passed to the script as an environment variable for each response that the server has received. The script uses this environment variable as a means to determine which response is to be sent back to the client. The client is the user at the location where the request originated. For example:

```
CGI-FORWARD-RESPONSE abcdefghij SIP/2.0
```

will return the response with the cookie labeled "abcdefghij" back to the client. Cookies are thus stored on the server.

3. *Generating requests*: A SIP server can also be responsible for the process of generating requests that originate at the server. For example in the case of multimedia conferencing the server is responsible for inviting users to a particular conference. The server creates requests on its own, including the message headers and the message body. It is also responsible for processing the entire transaction. This is not discussed in the RFC since it is outside the scope of SIP-CGI.

4. *Generating responses*: Simply using the status line of a SIP response will generate responses. As a result, that status line will be considered an action line for a SIP-CGI output, and the SIP server will generate the rest of the body of the SIP response. For example,

```
SIP/2.0 200 OK
```

will create a "200 OK" response to the original request.

Other available SIP-CGI actions are CGI-SET-COOKIE and CGI-AGAIN. CGI-SET-COOKIE sets the cookie or token described in CGI-FORWARD-RESPONSE. All subsequent script invocations must use the cookie set here. CGI-AGAIN enables the script to be executed again, after receiving subsequent requests and responses belonging to this transaction.

Besides using the available SIP-CGI actions, SIP-CGI scripts can also modify header fields. In the example of the CGI-PROXY-REQUEST (item #1 above), the "Contact" header field was modified to contain the SIP URL of the person to be contacted. Other available SIP-CGI header fields are CGI-REQUEST-TOKEN and CGI-REMOVE. CGI-REQUEST-TOKEN assists in matching responses to a proxy request by passing a token as a header field in a CGI-PROXY-REQUEST. Subsequent responses to the proxy request will pass this token in a meta-header. The CGI-REMOVE header field allows the script to remove certain header fields from the outgoing request or response.

SIP-CGI scripts can also be used to invoke other programs similar to that capability of CGI for HTTP. This will be demonstrated in context of the call-related services implemented for CINEMA (see section 5.3).

## 5.3 CINEMA and SIP-CGI

The entire CINEMA SIP server was written in C language, including the different libraries it uses, such as the libraries *libcine* and *libsip*. The *libcine* library is responsible for creating the network sockets, resolving host names to IP addresses, handling requests, logging files and handling errors. The *libsip* library is responsible for authenticating SIP requests, handling SIP requests and responses, parsing SIP messages and performing basic database queries for user lookups.

Appendix C contains file listings from the libraries *libcine* and *libsip* and describes their functionalities.

Figure 5.1 illustrates the internal components of the CINEMA SIP server. It is a basic flow diagram showing how the server will handle a request. In this example, the server simply proxies the request to another client. Udp.c contains functions such as `ReceiveUDP` that can handle the network connections. Request.c handles the different SIP requests. Request.c uses the *libdb++* library to check if the user who is issuing the request is registered and what kind of request he is allowed to use. Method.c then determines what kind of policy to use and calls `execute_policy()` which executes the policy, while `execute_policy()` is contained in policy_core.c. There are four types of policies that the policy_core.c can execute, namely the cgi, proxy, redirect and route policies. Each of these policies can spawn a thread to handle a SIP client; for example, in the diagram, proxy.c spawns the client.c. The main job of client.c is to proxy the request. The client.c is written as an explicit state machine that changes its state depending on the response it receives after proxying the request. Each of the policies can control the behavior of the state machine.

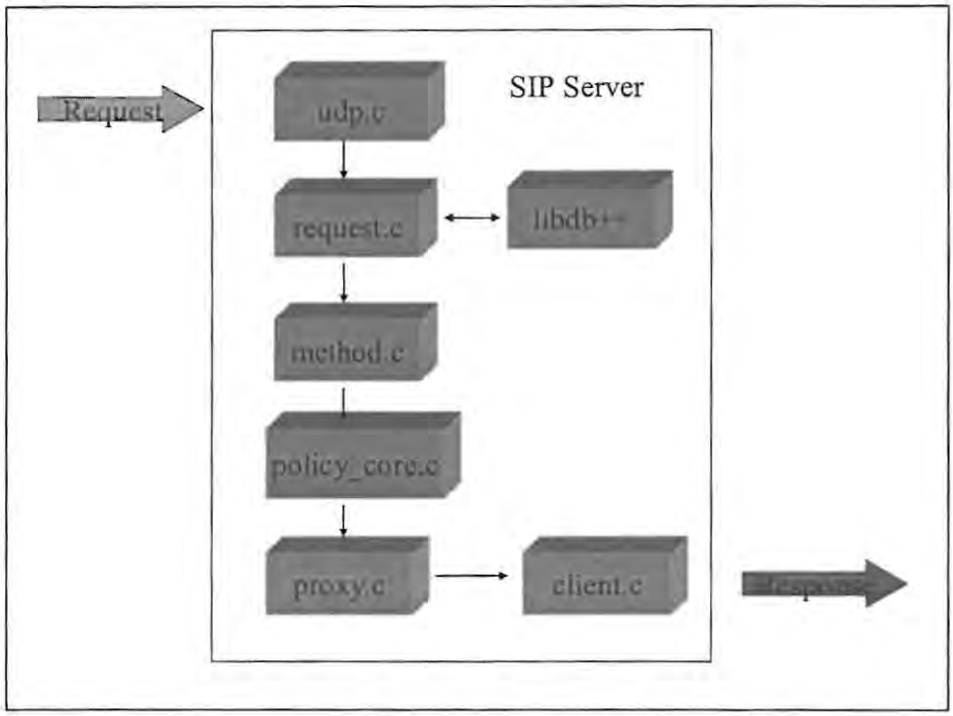


**Figure 5.1 CINEMA SIP-CGI call flow**

The SIP-CGI scripting capability is built into cgi.c. Conforming to the RFC, SIP-CGI scripts are executed when a message arrives. To do this, cgi.c must:

1. Load the environment variables that the script needs in order to execute.
2. Load the script, passing it the environment variables.
3. Execute the script.
4. Get the output from the script.
5. Parse the output from the script and validate it as proper SIP-CGI output. In order for it to be recognized as proper SIP-CGI output, it must generate one of the outputs defined in SIP-CGI actions.
6. Perform any header field changes required from the output. For example, the script might require that the contact headers be changed in the SIP message (as described above in the example of the CGI-PROXY-REQUEST, section 5.2.2). The SIP server will then make the necessary changes on the outgoing SIP message.

7. Finally, cgi.c will perform the action of the script output, and either proxy, forward or send the message to the required SIP URI.

### 5.3.1 Simple call-blocking script

One advantage of SIP-CGI scripts is that they can be written in any scripting language. Below is an example of a SIP-CGI script written in Perl:

```
#! C:/Perl/bin/perl
# Reject messages whose 'From:' matches 'sip:ming@' by responding with
# 603 reject message.
print "SIP/2.0 100 Wait\n\n";
if (defined $ENV{SIP_FROM} && $ENV{SIP_FROM} =~ "sip:ming@") {
    print "SIP/2.0 603 I don't want to talk to you\n\n";
}
else {
    print "SIP/2.0 200 OK lets talk\n\n";
}
```

This is a simple script that has responded to the SIP request firstly with a 100 Wait response message, then, once it had determined that the person calling was ming, it responded with a 600 I don't want to talk to you rejection message. If the person calling were not `sip:ming`, then the default action would have accepted the call. Notice that the SIP-CGI action lines terminate with "\n\n", to enable the server to determine their end.

The server executes the script. The output in this case is:

SIP/2.0 100 Wait

SIP/2.0 603 I don't want to talk to you

The server performs the required actions by generating these SIP messages:

SIP/2.0 100 Wait

Via: SIP/2.0/UDP 146.231.121.130:5060

From: sip:g9610645@146.231.121.130

To: sip:ming@cssipict.cs.ru.ac.za

Call-ID: 751008035@146.231.121.130

CSeq: 1 INVITE

Date: Thu, 27 Dec 2001 10:25:46 GMT
Server: Columbia-SIP-Server/1.0
Content-Length: 0

SIP/2.0 603 I don't want to talk to you
Via: SIP/2.0/UDP 146.231.121.130:5060
From: sip:g9610645@146.231.121.130
To: sip:ming@cssipict.cs.ru.ac.za
Call-ID: 751008035@146.231.121.130
CSeq: 1 INVITE
Date: Thu, 27 Dec 2001 10:25:46 GMT
Server: Columbia-SIP-Server/1.0
Content-Length: 0

It is the server that generates these responses and sends them back to the caller. The caller was ming@cssipict.cs.ru.ac.za and he was trying to call g9610645@cssipict.cs.ru.ac.za. The server is cssipict.cs.ru.ac.za. CINEMA has provided some interfaces for users to upload scripts onto the server. Users can either upload their scripts via the web interface or via the `sipc` user agent. If users decide to use `sipc` they then upload the scripts by sending a REGISTER message, containing the script, to the server. The REGISTER sequence ensures that the uploaded script is from the authenticated user, but it does not determine that the uploaded script is a valid script. The server will execute the script; if the script times-out or produces an invalid output the server produces a 500 Server Error message to the caller.

### 5.3.2 SIP-CGI HTML script

To further demonstrate the capabilities of SIP-CGI, other scripts are provided below. The first script allows the programmer to check the environment variables that are passed to the script, and so are available to be used; for example, the programmer can use this script if he or she wants to know what SIP fields can be accessed. This script also demonstrates how one can add different content-types to the message body of the SIP

message and it shows the ability of SIP-CGI to generate dynamic content. This is similar to CGI for HTTP.

```
#!c:/perl/bin/perl -w
#print out some content-type stuff
$myenv = "ming\n";
foreach $env (sort keys %ENV) {
   $myenv .= "$env=\"$ENV{$env}\"\n";
}
$body = "<html>\n";
$body .= "<p> MCHSIEH perl cgi\n";
$body .= "</p>\n";
$body .= $myenv;
$body .= "\n";
$body .= "</html>\n\n";
print "SIP/2.0 600 Web stuff\n";
print "Content-type: text/html\n";
print "Content-Length: ", length($body), "\n\n";
print $body;
```

In the first part of this code there is a foreach loop that goes through the environment variables and stores them in the array `$myenv`. `$body` contains some html code along with the environment variables. This `$body` is printed as the body for the SIP message response.

The response received on the client side is shown below:

SIP/2.0 600 Web stuff
Via: SIP/2.0/UDP 146.231.123.15:5060
From: ming@csmsc01ict.cs.ru.ac.za
To: sip:g9610645@cssipict.cs.ru.ac.za
Call-ID: 900645918@146.231.123.15
CSeq: 1 INVITE
Date: Wed, 13 Nov 2002 14:05:59 GMT
Server: Columbia-SIP-Server/1.0
Content-Length: 779
Content-Type: text/html

The body of the SIP message contains html code. With an html-enabled `sipc` user agent, one that is able to view and display html, the body can be displayed as a web page. This is shown in Figure 5.2.

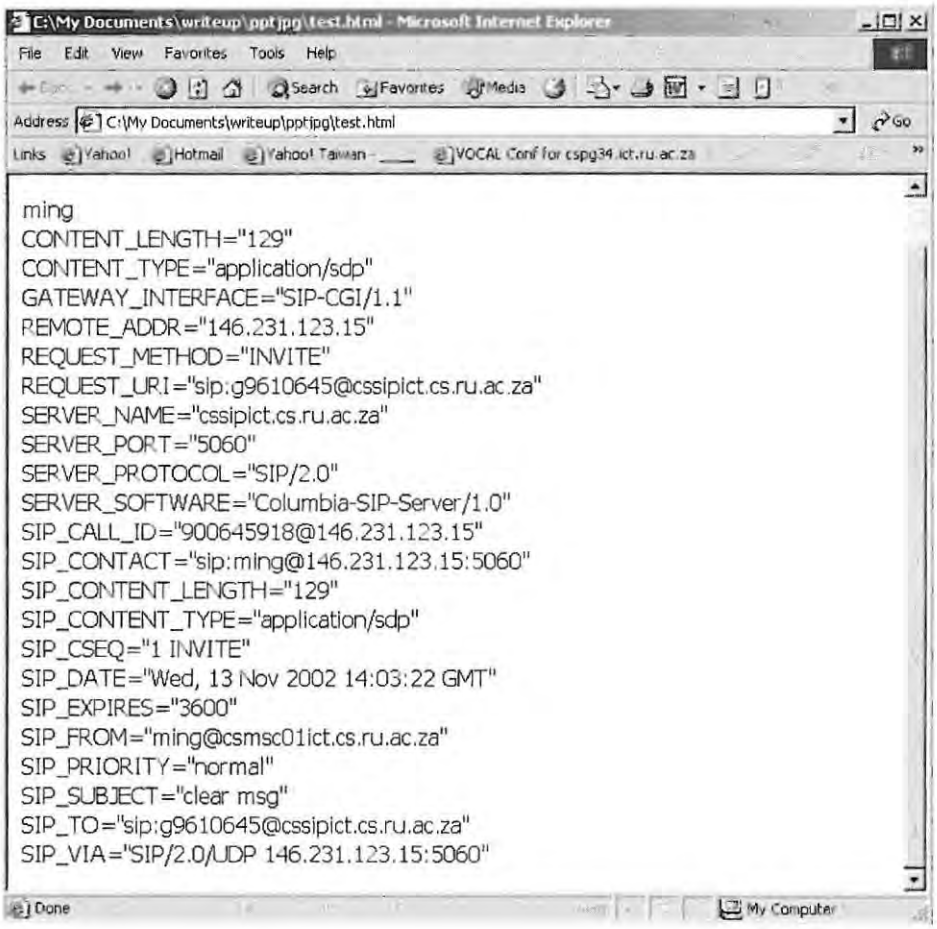


**Figure 5.2 CINEMA HTML code output**

### 5.3.3 Username lookup service

The next script demonstrates how SIP-CGI scripts can be used to do certain backend operations, for example database queries.

```
#!c:/perl/bin/perl -w
#my SIP-CGI script to test db access
use DBI;
#a subroutine to print out errors
sub fail {
```

```
    my($status, $reason) = @_;
    print "SIP/2.0 $status $reason\n\n";
    $sth->finish;
    $dbh->disconnect;
    exit 0;
}
print "SIP/2.0 100 Wait\n\n";
```

To access databases in Perl the DBI Perl module must be installed on the server. To access MySql databases in particular, the DBI::MySql driver needs to be installed as well. Once the module is installed, the statement "`use DBI;`" will load the right drivers when executing the script.

```
#try to connect to database and make a query
$database = "sip";
$hostname = "cssipict.cs.ru.ac.za";
$user = "ming";
$password = "mysql2000";
#use "500" response message to print out errors from the server side
$dbh = DBI->connect("DBI:mysql:$database:$hostname",
$user, $password)
or fail("500", "Can't connect");
$sth = $dbh->prepare("select * from contacts")
or fail("500", "Can't prepare");
$sth->execute()
or fail("500", "Can't execute");
#if it gets past all these statements just print out a 100 message
#to tell the user that he is connected to the database
print "SIP/2.0 100 Passed all the sql statements\n\n";
```

The name of the database is stored in the variable `$database`. The host name of this database server is cssipict.cs.ru.ac.za. A user name and password are required to connect to the database. The statements `connect()`, `prepare()` and `execute()` are SQL statements employed to access the database. The `print` statement prints out a SIP message, notifying the user that he has successfully connected to the database.

```
#check if the person calling is in the contacts list
$myusername = $ENV{SIP_CONTACT};
$checkflag = 0;
#get a row from the result and check it
while( $resptr = $sth->fetchrow_hashref() )
{
        if(($resptr->{"contact"} eq $myusername))
        {
           #call accepted
           $checkflag = 1;
```

```
        }
}
if($checkflag == 0)
{
        fail("600", "You are not on the contacts list");
}
#close the connection to the database
$sth->finish;
$dbh->disconnect;
```

`$myusername` is storing an environment variable, the SIP_CONTACT field. The username is checked by a `while` statement that goes through the output from the SQL query to see if the user is in the database, printing a SIP-CGI error statement if the user is not in the database. If the person calling is not on the list he or she is rejected and sent the response (600 You are not on the contacts list) or else the call is accepted.

This SIP-CGI database query raises an issue that should be clarified. The database query will require extra time to perform if the user database is large (in this scenario there were only five users in the database). The time delay for the SQL statement to take place could adversely affect the performance of the SIP server. The CINEMA SIP server is responsible for establishing real-time communications, as the time it takes to establish a session is important. Therefore, a time limit is placed on the execution of the script. If the script times-out the server sends a 500 server timeout error to the client.

### 5.3.4 Missed call service

The next script demonstrates how SIP-CGI can be used to interoperate with other types of services. This script allows users to use the Short Messaging Service (SMS) gateway, set up by Guy Halse at the Department of Computer Science, Rhodes University, to notify themselves of missed calls. A user modifies this script to match his or her cellphone number and uploads the script onto the SIP server.

```
#! C:/Perl/bin/perl
#load the correct modules
use Socket;
use MIME::Base64;
#a subroutine to print out errors
sub fail {
```

```
    my($status, $reason) = @_;
    print "SIP/2.0 $status $reason\n\n";
    exit 0;
}
#compose the soap message
$username = 'sms';
$password = 'sms';
$ARGV[0] = '0836816045';
#the person who is trying to call you
$ARGV[1] = $ENV{SIP_CONTACT};
$AUTH='YXNobGV5OmFzaGxleQ==';
$msg = "<?xml version=\"1.0\" encoding=\"iso-8859-1\" ?>\n";
$msg .= "<soap:Envelope xmlns:soap=\"http://www.w3.org/2001/12/soap-
envelope\">\n";
$msg .= " <soap:Body>\n";
$msg .= " <sms:sms
xmlns:sms=\"http://omniscient.ict.ru.ac.za/sms/sms.xsd\">\n";
if(defined($ARGV[0]) && defined($ARGV[1])) {
    $msg .= "  <sms:sendsms>\n";
    $msg .= "  <sms:phone>$ARGV[0]</sms:phone>\n";
    $msg .= "  <sms:message>$ARGV[1]</sms:message>\n";
    $msg .= "  </sms:sendsms>\n";
}
$msg .= "  </sms:sms>\n";
$msg .= " </soap:Body>\n";
$msg .= "</soap:Envelope>\n";
#setup a socket connection and send the soap message
socket(T, PF_INET, SOCK_STREAM, getprotobyname('tcp')) or fail("500",
"socket: $!");
connect(T, sockaddr_in(80, inet_aton('omniscient.ict.ru.ac.za'))) or
fail("500", "connect: $!");
select((select(T), $|=1)[0]);
print T "POST http://omniscient.ict.ru.ac.za/sms/ HTTP/1.0\n";
print T "Authorization: Basic " . $AUTH;
print T "Host: omniscient.ict.ru.ac.za\n";
print T "User-Agent: test-sms/0.1\n";
print T "Content-Length: " . length($msg) . "\n\n";
print T $msg, "\n";
```

The SMS gateway is set up using the interface Simple Object Access Protocol (SOAP). SOAP is used to establish a common interface for other services to access the SMS gateway. More information about SOAP can be found at the [SOAP, 2000] website; information about how the SMS gateway was built is given in [Halse and Wells, 2002]. A cellphone number and the message to be sent are contained within a SOAP message. The SOAP message is then sent within a HTTP POST message.

```
#compose the message to be sent back to the client
$body = "<html>\n";
$body .= "MCHSIEH sending a sms to the user\n";
$body .= $msg, "  \n";
```

```
$body .= "--------------------------\n";
#get a line from the socket
$body .= <T>;
$body .= "</html>\n";
$body .= "\n";
print "SIP/2.0 600 I can't talk right now\n";
print "Content-type: text/html\n";
print "Content-Length: ", length($body), "\n\n";
print $body;
```

When the script sends the SOAP message it is actually interacting with a HTTP server, making a request to it and waiting for a response. The response from the HTTP server is captured inside a html message and is sent inside a SIP response message. Executions of this script can potentially time-out due to interaction with the HTTP server.

## 5.4 Call Processing Language (CPL)

Due to a number of problems associated with SIP-CGI, mainly involving how a call is established and security issues, SIP-CGI is not suitable for end-users in telephony services; therefore, a more restrictive language was needed. Call Processing Language (CPL) was developed by the IETF as a solution to this problem [Lennox and Schulzrinne, 2000a and Lennox and Schulzrinne, 2000b]. CPL is used to describe and control Internet telephony services. Although it is not a complete language (i.e., it does not have loops or variables to make it Turing-complete), it may be easily read and edited by users. The eXtensible Markup Language (XML) was chosen to describe CPL. XML is a meta-language used to describe other languages; it is extensible and allows developers to define what data should be in a CPL document. That data is stored in nested structures allowing other applications to use the data as they see fit. CPL is also protocol-independent and can be used with SIP or H.323.

### 5.4.1 CPL model

The basic structure of CPL is that of a tree consisting of nodes and subnodes. The nodes denote the actions to be taken when an event arrives; the subnodes can denote subactions to be executed. The actions to be taken are divided into 4 groups: switch, location, signaling and non-signaling. Subactions focus each action; for example, within a switch

action other subactions are available, such as address, string, time or priority switch. Figure 5.3 shows the basic structure of CPL. Incoming and Outgoing are top-level switches. Incoming denotes the actions to be taken when a call arrives for a particular callee. Outgoing denotes the actions taken to handle a caller's outgoing calls.

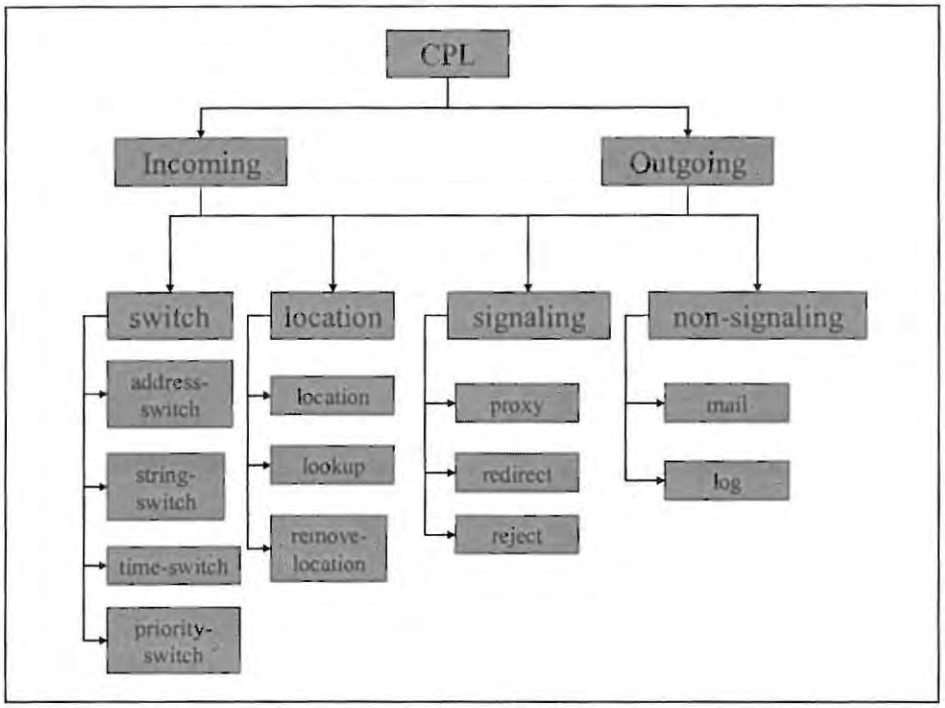


**Figure 5.3 CPL model**

Below is an example of CPL script showing some nodes and tags:

```
<?xml version="1.0" ?>
<!DOCTYPE cpl PUBLIC "-//IETF//DTD RFCxxxx CPL 1.0//EN" "cpl.dtd">
<cpl>
  <subaction id="voicemail">
    <location url="sip:jones@voicemail.example.com">
      <redirect />
    </location>
  </subaction>
  <incoming>
    <address-switch field="origin" subfield="host">
      <address subdomain-of="example.com">
        <location url="sip:jones@jonespc.example.com">
          <proxy timeout="10">
            <busy> <sub ref="voicemail" /> </busy>
```

```
            <noanswer> <sub ref="voicemail" /> </noanswer>
          </proxy>
        </location>
      </address>
    </address-switch>
  </incoming>
</cpl>
```

This sample CPL script is for Mr. Jones who works at example.com. He has written a CPL script that will filter his incoming calls. Mr. Jones would like calls from his company, example.com, to be immediately delivered to his PC. If he is busy with another customer or does not answer the phone, he would like his calls to be redirected to his voicemail box. If none of the actions apply, the call is forwarded to Mr. Jones's usual location.

The script that he has written will filter depending on the origin address of the calls (field="origin") and the host name portion of the address (subfield="host"). The subdomain-of field will match those host names whose domain names contain "example.com"; in this manner, calls from "sales.example.com" will match and so will be processed. The location url adds the SIP URL to the location set and this will be used for proxying the request. The proxy action tag contains a timeout value to indicate the amount of seconds to wait before the call is considered not answered. The subaction node does the redirecting of the call to Mr. Jones's voicemail box. If none of the actions have taken place, a call is processed in the normal manner by being forwarded to Mr. Jones's usual location.

## 5.5 VOCAL and CPL

The feature servers in VOCAL use CPL scripts to describe the features of the system. Features are the same as services. Features in VOCAL are classified into two categories, namely Calling features and Called features. Calling features are those activated in an outgoing call, such as Caller ID Blocking and Call Blocking. Called features are features that are activated in an incoming call; examples are Call Forwarding and Call Screening.

The operation of a service or feature is as follows:

1. The user configures, via the web interface, a service.
2. This service is stored as a CPL script on the provisioning server. When a call comes in or goes out the redirect server is contacted.
3. The redirect server queries the provisioning server to see if a feature is enabled for that user, then determines to which feature server it must redirect the request.
4. The request gets redirected to a particular feature server.
5. A particular feature server queries the provisioning server to get the CPL script enabled for that user, compiles it to an executable one and stores it in the cache so that at the next request it need not query the provisioning server again. A call-blocking feature server will store the call-blocking feature for the user while the call-screening feature server will store the call-screening feature for the user. This differentiates the various feature servers.

All of the operations of the services described in the following sections follow the sequence of steps just described.

### 5.5.1 Call blocking

Call blocking is a calling feature that is activated on an outgoing call. This service allows users to restrict calls. For example, the service can be used to block long-distance calls.

Figure 5.4 shows the sequence of SIP messages for the call-blocking service. The following explains the sequence of events depicted in Figure 5.4:

1. The user agent sends an INVITE message to the Marshal server, because the user is requesting a call to an external number.
2. The Marshal server proxies the INVITE message to the Redirect server.
3. The Redirect server responds with the message "302 Moved Temporarily", which contains the URL of the Feature server that the Marshal server must try next.
4. The Marshal server acknowledges the message by sending an ACK message.

5. The Marshal server forwards the INVITE message to the Feature server. The Feature server receives this INVITE message and contacts the Provisioning server to get the CPL scripts for the particular caller.
6. The Feature server executes the CPL script and determines that the number that the user is trying to call is out of bounds and sends a "403 Forbidden" message.
7. The Marshal server acknowledges the message by sending an ACK message.
8. The Marshal server proxies the message "403 Forbidden" back to the user agent. The user agent will then respond by acknowledging this message.

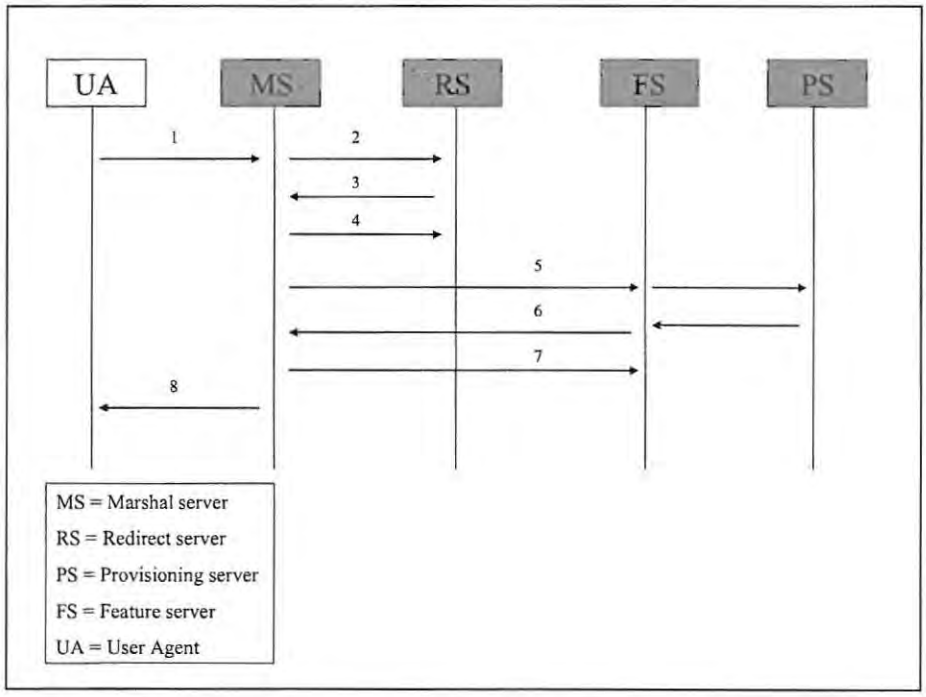


**Figure 5.4 VOCAL call blocking**

Below is the CPL script generated by the Provisioning server. The output has been formatted for easier reading:

```
<?xml version="1.0" ?>
<!DOCTYPE cpl SYSTEM "cpl.dtd">
<cpl>
<subaction id="rejectcall">
  <reject reason="feature activated" status="reject">
  </reject>
</subaction>
```

```
<outgoing>
  <address-switch field="original-destination" subfield="user">
   <address subdomain-of="1900">
      <sub ref="rejectcall">
      </sub>
   </address>
   <address subdomain-of="976">
      <sub ref="rejectcall">
      </sub>
   </address>
   <address subdomain-of=";">
      <sub ref="rejectcall">
      </sub>
   </address>
   <otherwise>
     <lookup clear="yes" source="registration" timeout="2">
      <success>
        <proxy ordering="first-only">
        </proxy>
      </success>
      <notfound>
        <sub ref="rejectcall">
        </sub>
      </notfound>
     </lookup>
   </otherwise>
  </address-switch>
</outgoing>
</cpl>
```

The CPL script generated by the Provisioning server uses address-switches in order to filter outgoing calls. (Address-switches will switch depending on the address that is received.) If the script sees numbers such as "1900286" and "97612345" then the call will be rejected.

### 5.5.2 Call screening

Call screening is a user feature that screens incoming calls and allows the user to filter out unwanted ones.

A SIP messages diagram for call screening would be similar to Figure 5.4 depicting call blocking, except that the INVITE message could come from a user agent or from another Marshal server. Also, the Redirect server may contact the Provisioning server to get the contact list for the callee. The number of messages would remain the same, from the

callee's point of view, but if the caller had additional CPL scripts, for example the call-blocking script, the total number of SIP messages for this transaction could double.

The CPL script for the call-screening feature is similar to the CPL script for call blocking, except for the incoming node;

```
<incoming>
  <address-switch field="origin" subfield="user">
   <address subdomain-of="2000">
<sub ref="rejectcall">
</sub>
   </address>
```

This code segment will filter out calls coming from users whose identifier starts with "2000".

### 5.5.3 SMS missed call service

This service is similar to the "Missed Call Service" implemented in CINEMA. It allows a user to be notified, via SMS, of any missed calls. To do this, a new extension had to be given to the CPL. The chosen extension was:

```
<sms destination="0836816045" msg="Missed Call">
</sms>
```

The chosen extension denotes the destination of the SMS via the destination attribute, and the message to be sent is denoted via the msg attribute.

In order to make this extension to CPL, modifications were made on the CPL parser and the feature server. Vovida CPL documents [Vovida, 2001b] explains how CPL scripts are transformed into finite state machines programmed in C++. It also explains what files must be modified in order to extend CPL.

First, the SMS CPL extension was assigned to the category "Other actions" since no SIP signaling messages are required for sending the SMS. Next, the extensions to CPL were

modified in CPLFeatureBuilder.cxx, which creates the Document Type Definition (DTD) for CPL. These two modifications are shown below:

```
oFile << "<!-- Other actions -->" << endl;
//mchsieh added sms
oFile << "<!ENTITY % OtherAction 'mail|log|sms' >" << endl;
oFile << endl;
...
oFile << "<!-- Simple Messaging: Simple Messaging -->" << endl;
//mchsieh required here to describe the node
oFile << "<!ELEMENT sms ( %Node; ) >" << endl;
oFile << "<!ATTLIST sms" << endl;
oFile << "   destination   CDATA    #IMPLIED" << endl;
oFile << "   msg           CDATA    #IMPLIED" << endl;
oFile << ">" << endl;
oFile << endl;
oFile << endl;
```

Next, actual headers must be added for the CPL parser to parse the new CPL script. This was done in CallProcessingLaguage.cxx. The attributes of the SMS tag (destination and msg) are added in this file.

```
const char* CPLNodeStr [] =
    {
      ...
      "sms",//mchsieh sms tag
      ...
    };

//mchsieh number of attributes for the SMS tag
const unsigned int smsAttrTableSize = 2;
const char* smsAttributesTable [] =
    {
      "destination", "msg"
    };
```

Next, the CPL parser must be told what to do when it sees a SMS tag. This was done in CPLInterperter.cxx:

```
bool
CPLInterpreter::nodeStart( int nodeFound, xmlNodePtr ptr )
{
        ...
        case CPLNodeSMS:
        returnValue = processSMS( ptr );//mchsieh
        break;
        ...
}
```

When the interpreter sees the `CPLNodeSMS` it calls the `processSMS` function.

```
//mchsieh the processSMS function
bool
CPLInterpreter::processSMS( xmlNodePtr ptr )
{
    Sptr < CPLOpSMS > aNode = new CPLOpSMS;
    processNodeAttributes(      aNode,      ptr,      smsAttributesTable,
smsAttrTableSize );
    //Add log operator to current state machine
    bool result = myCplBuildStatus->nodeEntry( aNode, CPLBuildStatus::
SimpleOp );
    //Add default action if no node follows
    if ( (result != false) && (ptr->children == NULL) )
      return processDefaultAction( ptr );
    else
      return result;
}
```

After that, `processSMS` creates a new node, the `CPLOpSMS` node, passing it the attributes `smsAttributesTable` and `smsAttrTableSize`.

Appendix D contains the code for the `CPLOpSMS` node. The functionality of the node is to establish a connection to the SMS gateway, similar to what was done for CINEMA; it will compose the SOAP message to be sent using the values from `destination` and `msg` in the CPL script and then send the message via TCP.

VOCAL provides a useful test utility called "`cplTest`" to test the validity of the CPL script. `cplTest` was used to test the following SMS CPL script:

```
<?xml version="1.0" ?>
<!DOCTYPE cpl PUBLIC "-//IETF//DTD RFCxxxx CPL 1.0//EN" "cpl.dtd">

<cpl>
  <subaction id="missed">
    <sms destination="0836816045" msg="Missed Call">
    </sms>
  </subaction>

  <incoming>
   <location url="sip:2000@146.231.121.142:5060;user=phone">
     <proxy timeout="5">
       <busy>
```

```
        <sub ref="missed"></sub>
      </busy>
      <noanswer>
        <sub ref="missed"></sub>
      </noanswer>
      <failure>
        <sub ref="missed"></sub>
      </failure>
    </proxy>
  </location>
 </incoming>

</cpl>
```

This specific script will first proxy the request to

"sip:2000@146.231.121.142:5060;user=phone".

Depending on the result of the proxy, if the call is not answered within 5 seconds it will be considered not answered and an SMS will be sent to "0836810645". This belongs to the switch <noanswer>. If the user is busy or the call can not be completed an SMS will also be sent. This belongs to the <busy> and <failure> switches. Figure 5.5 shows the SMS that notifies the callee of a missed call.

**Figure 5.5 Cellphone missed call SMS**

## 5.6 Call-related services discussion

Both CINEMA and VOCAL provide excellent facilities to create call-related services. CINEMA provides SIP-CGI so that users may write their own services; programmers can create new services in VOCAL by making extensions to the CPL language and the GUI allows users to personally customize the service. However, several significant similarities and differences between the two environments exist.

**Similarities—**

Considering the architecture designs of CINEMA and VOCAL, we recommend that each SIP endpoint be registered with the system and that all calls go through the `sipd` or Marshal server, respectively. (Registering with CINEMA or VOCAL provides the user agents with basic services such as directory services, whereby users can call each other using aliases instead of IP addresses.) Accordingly, if all endpoints are registered with the system and all endpoints have to use the `sipd` or the Marshal server in order to make calls, then only one SIP-CGI or one CPL script would be needed in order to service the call-blocking or call-screening services, respectively.

However, the SIP RFC does not enforce the rule that all endpoints must use their servers to make calls. In fact, users can bypass the servers and call each other directly, thus avoiding the call-blocking or call-screening service enabled by callees.

The authors of the SIP RFC designed SIP to focus mainly on session establishment and so use the Internet approach, in which is there is no central point of control; therefore, it depends on the administrator of the network to enforce users to use a proxy in order to make calls. This suggests the need to put a policy in place whereby all external calls coming in or going out have to pass a Gateway Marshal Server (GMS). The GMS would act like a firewall. The gateway would filter unwanted incoming calls from external sources and it could also act as a proxy for outgoing calls. Then, it would depend on the user to choose whether or not local calls are put through the GMS.

**Differences—**

Due to a lack of central control of the calling process, CINEMA has provided SIP user agents with a mechanism whereby a user can load scripts directly to the user agents and have them activated upon receiving a call (known as local scripting capability). This ensures that the service is activated when the call is coming from an external source or from the server. VOCAL does not provide such a mechanism.

The local scripting capability in CINEMA introduces the problem of mobility. For example, if a user decides to move to another user agent, how would he or she move the settings and script to the new user agent? Introducing a centralized script provisioning and profile storage server for CINEMA could solve the problem. This would allow one to log onto the server from any user agent and download one's own profile plus the script that has been stored on the server. This would ensure that one script is available for the client and the server. Of course, only user agents that are SIP-CGI capable would be able to access this feature or service. This server could be introduced as another service.

In CINEMA's SIP server (`sipd`) there is currently no support for outbound SIP-CGI. This means that users cannot upload a script onto the server that would block outgoing calls. Various scripts were tried to see if scripts could be executed on outgoing calls, including:

```
#!c:/perl/bin/perl -w
# Don't allow calls to 'sip:g9610645@' by responding with
# 600 Call is not allowed.
if (defined $ENV{SIP_TO} && $ENV{SIP_TO} =~ "sip:g9610645@") {
    print "SIP/2.0 600 Call is not allowed\n\n";
}
```

This script, if executed, would match the `SIP_TO` field in the SIP message in order to block outgoing calls. Although the script was uploaded onto a `sipd` server and `sipc` user agent, neither of them were executed on outgoing calls, but only when an incoming call arrived. Through correspondence with the researchers at Columbia University, we established that the outgoing SIP-CGI capabilities had not been implemented but the call blocking (outgoing) capability will be available when the CPL feature is incorporated

into CINEMA. In VOCAL, because CPL is supported, call blocking of outgoing calls is already enabled.

In VOCAL approximately eight messages are required to complete the call-screening service, as compared to four messages in CINEMA (see sections 5.5.2 and 5.3.1, respectively). This is a substantial difference. The eight messages are viewed from the callee's perspective, and the number of messages could increase if we included the caller's features. For example, if the caller has enabled call blocking in regard to outgoing calls, his call has to go through the call-blocking feature server before the Marshal server can forward the call to the callee. This increases the number of messages by seven; each of these messages must be processed on the server, potentially creating significant performance impact on the server and unwelcome delays for users. The distributed nature of VOCAL is the factor responsible for introducing these extra messages. Yet, the system's distributed nature provides scalability and these delays are acceptable in terms of the extra number of users that VOCAL may provide.

In terms of the security of the system, SIP-CGI allows users to run basically anything on the server and thus undermine the system's security. CPL provides a much more secure service deployment model by restricting the user to a smaller, better-controlled set of services. On the other hand, CINEMA provides a more flexible environment to create services, because the user is not restricted to what is made available by CPL.

In terms of services control, VOCAL presents a centralized control model, as all the CPL scripts are stored on a centralized server and the administrator can manage them from there; in CINEMA the administrator has no control over which scripts are executed since SIP-CGI scripts can be located on the user agents.

It seems that service creation in CINEMA is modeled closely on the Internet model, by virtue of providing a facility for users to run scripts on the server and by virtue of maintaining no centralized position of control. In VOCAL, an approach that more

resembles traditional telecommunications is used, as it centralizes the services and provides control of these services to the administrator.

## 5.7 Summary

In this chapter we investigated two main call-related service-creation mechanisms, namely SIP-CGI and CPL; specifically they are the implementation of SIP-CGI for CINEMA and CPL for VOCAL. Creating actual services has provided explicit examples. Several issues regarding call-related service-creation were also discussed.

# Chapter 6 Interactive Services

## 6.1 Introduction

Unlike call-related services, interactive services involve user interaction during the execution of a service. Services in both categories require SIP signaling. Voicemail is one example of an interactive service, since during the service the user is requested to record a voice message and input any number of digits to confirm the message.

A voicemail service is vital to any telephony system. In this portion of the work we investigate how the voicemail service is variously implemented in CINEMA and VOCAL, in an effort to show how new interactive services may be provided. We also describe in detail the creation of a notification service in both environments.

## 6.2 CINEMA's voicemail service

The CINEMA voicemail system is based on a SIP and Real-Time-Streaming Protocol (RTSP) combination. RTSP is used to record, send and control recorded voice streams. The basic architecture of the voicemail system consists of the SIP proxy server (`sipd`), Unified Messaging server (`sipum`) and RTSP server (`rtspd`). The Unified Messaging server is the voicemail server that is able to accept calls from users and record their messages (voice messages in particular). The RTSP server is the multimedia server that can be used to stream multimedia data over the network.

Users on the SIP network can register for the voicemail service by using a web interface. The proxy server needs to know the location of `sipum` and `rtspd`, and this information is given to the proxy via the administrator's web interface.

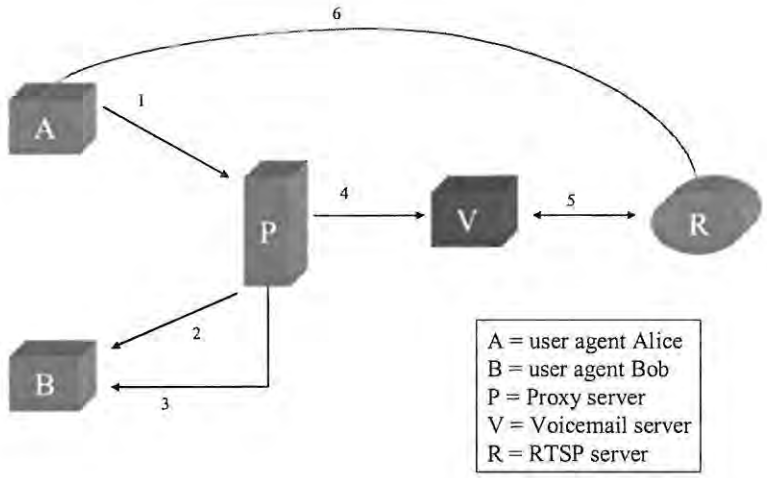


**Figure 6.1 CINEMA voicemail system**

Figure 6.1 illustrates the flow of SIP messages in a typical session using the voicemail service in CINEMA. In this example:

1. Alice sends an INVITE message to Bob. The proxy server (P) handles this message and finds that Bob has been registered at two locations, namely Bob@machineB.com and Bob@voicemail.com. Bob has also configured P to redirect his calls to voicemail if he cannot be reached at his first location.
2. P forwards the call first to Bob@machineB.com and the phone rings for 5 seconds before it is terminated because no one has picked up.
3. P sends a CANCEL message to Bob@machineB.com.
4. P then forwards the call to Bob@voicemail.com and the call is established between Alice and Bob@voicemail.com, essentially the same INVITE message that was originally sent to Bob@voicemail.com.
5. Bob@voicemail.com is an automated response agent that sets up a connection to `rtspd` (R) using the RTSP protocol.
6. Alice can now record her message with the R server; `rtspd` will record the message.

7. Once Alice terminates her call with Bob@voicemail.com by sending a BYE message, `rtspd` stores the message on the web server and sends an email message to Bob to notify him of the voicemail message. Bob can use a web browser or a suitable SIP user agent to retrieve this message.

Notably, in Figure 6.1 the media flow path (in blue) is between the RTSP server and the user agent Alice. The difference in the media flow path in the voicemail services of CINEMA and VOCAL is explained in section 6.3.

Figure 6.2 is a screenshot of the user interface for retrieving voicemail messages in CINEMA. More information about the CINEMA Voicemail System can be found in [Singh and Schulzrinne, 2000].

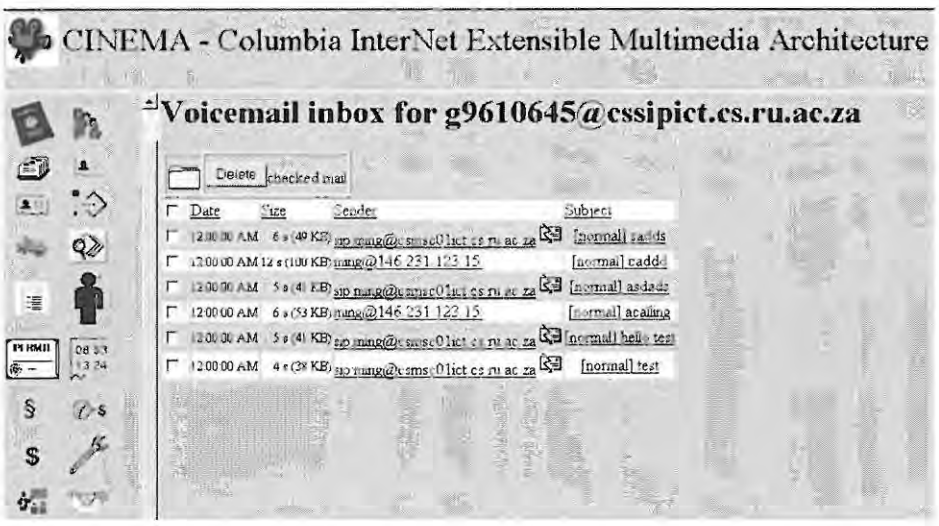


**Figure 6.2 Graphical user interface for CINEMA voicemail system**

## 6.3 VOCAL's voicemail service

Unlike the voicemail service in CINEMA, the voicemail feature in VOCAL is separated into two entities: the voicemail client (`vmclient`) and the voicemail server (`vmserver`). `vmclient` is responsible for the signaling capabilities of the feature (i.e., it handles the call between the caller and the voicemail system). `vmserver` instructs

vmclient about the actions it must take, based on actions taken by the caller. For example, if the user presses "1" to confirm the message she has recorded, vmserver will instruct vmclient to store the message and terminate the session with the caller. The signaling capabilities of the feature are stored in vmclient and the logic of the feature is stored with vmserver. vmclient and vmserver communicate with each other using a proprietary protocol developed by Vovida, known as VoiceMail Control Protocol (VMCP). VMCP is a simple protocol with messages such as STARTPLAY, STOPPLAY, STARTRECORD and STOPRECORD. Figure 6.3 illustrates the basic architecture of the VOCAL voicemail system.

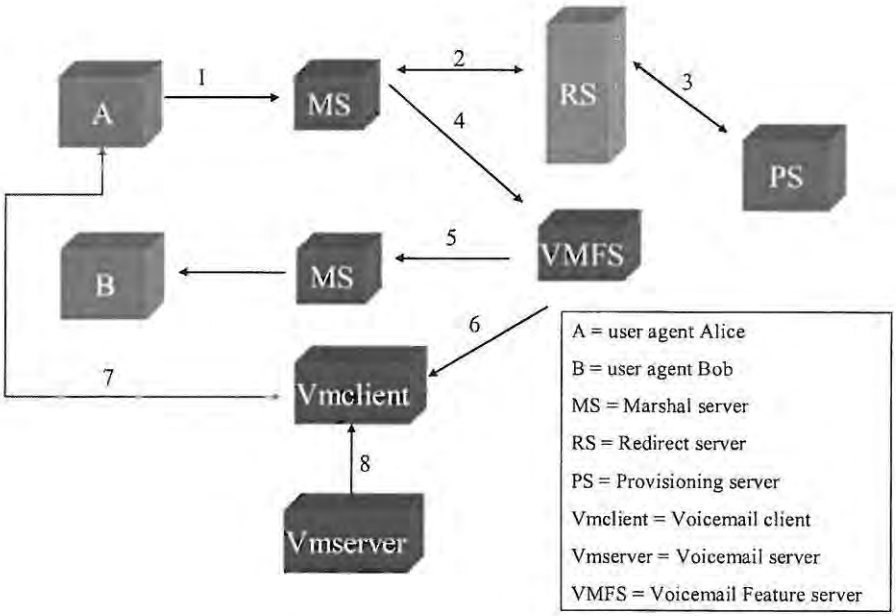


**Figure 6.3 VOCAL voicemail ystem**

The following describes the path of SIP messages using VOCAL's voicemail service in a typical session (Figure 6.3):

1. Alice sends an INVITE to Bob. The Marshal server handles this message.
2. The Marshal server forwards the request to the Redirect server.

3. The Redirect server checks with the Provisioning server concerning what services are enabled for Bob. The Provisioning server sends the information that Bob has voicemail enabled.
4. The Marshal server forwards the request to the Voicemail Feature server, which controls the feature.
5. The Voicemail Feature server will try to call Bob. If there is no response within 10 seconds, the Voicemail Feature server will send a CANCEL message to terminate the request. (The Marshal server must handle all these messages.)
6. The Voicemail Feature server forwards the request to `vmclient`. A session is set up between `vmclient` and Alice.
7. The RTP flow path is between Alice and `vmclient`.
8. Once a session is established between Alice and `vmclient`, `vmclient` sets up a connection to `vmserver`. `vmserver` and `vmclient` will communicate using VMCP. `vmserver` will instruct `vmclient` what to do: play and record a message for example.
9. After a greeting message is played to Alice, Alice can record her message. The RTP packets used for the media flows exist between `vmclient` and Alice.
10. Once the message is recorded, `vmserver` sends the message as a .wav file attachment to an email message that is addressed to Bob.

Figure 6.4 shows a screenshot of the email message containing the attachment (see item #10, above) that is eventually sent in the scenario laid out in Figure 6.3. More information about VOCAL's voicemail feature can be found at [Vovida, 2000].

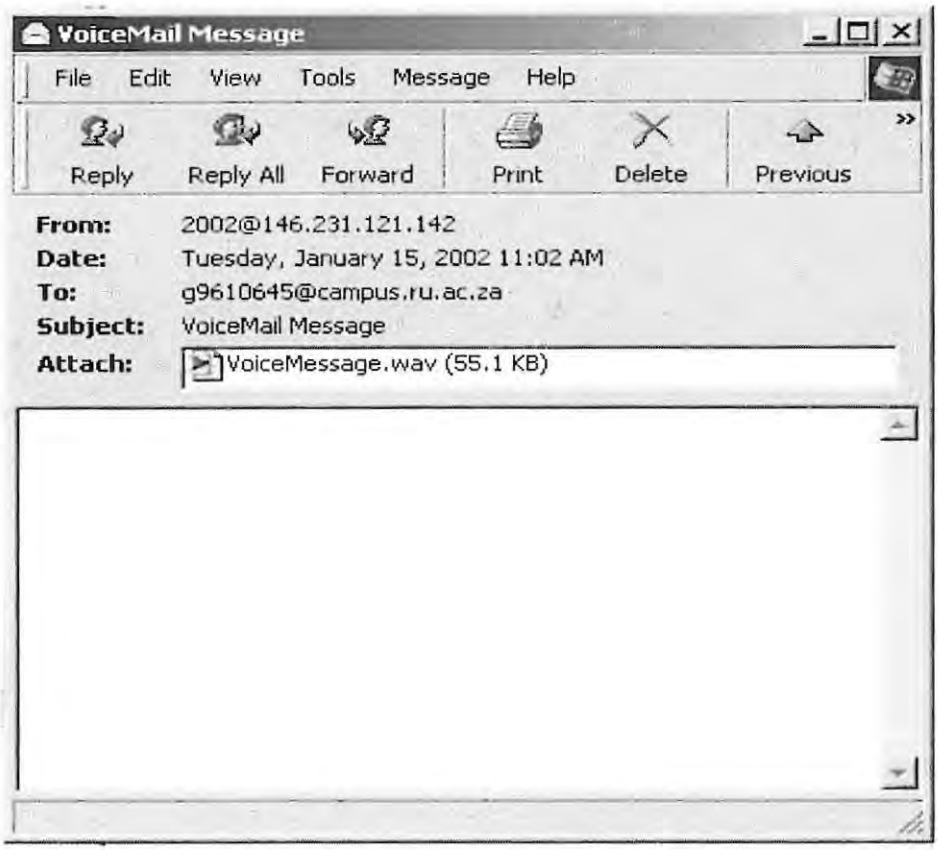


**Figure 6.4 VOCAL voicemail attachment**

## 6.4 CINEMA's reminder service

The work in this section investigates the possible implementation of new interactive services within CINEMA. A reminder service is created and used as a case study. The reminder service is in fact an alarm clock service that reminds the user of a specific event that is about to occur. Thus, the keyword alarm is used in various parts of the text and code. This reminder (also alarm or notification) service may be extended to include other services. For example, the service could be extended so that household appliances connected to the Internet could notify one of other particular events. The reminder service was implemented in CINEMA using essentially the same architecture as the voicemail service (see Figure 6.1). However, an Alarm server (`sipam`) replaced the Voicemail server (`sipum`). Also, the RTSP server (`rtspd`) was removed since the reminder service

would be text-based and no audio support was needed from that media server. The required SIP message flow for the reminder service is shown in Figure 6.5.

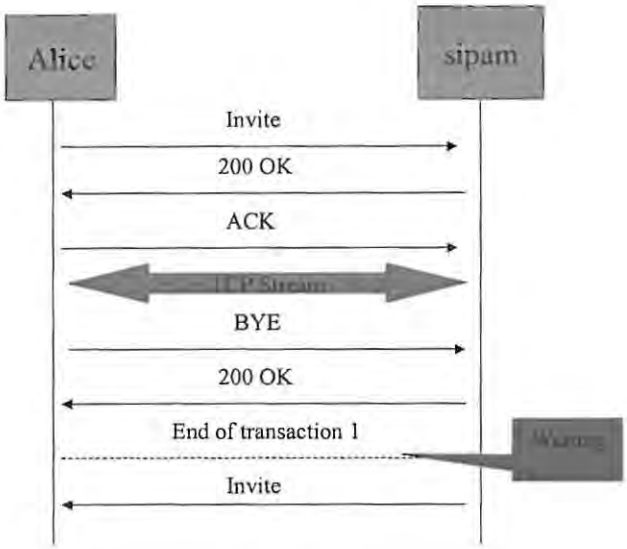


**Figure 6.5 CINEMA reminder service messages**

The TCP stream shown in Figure 6.5 represents the communication medium that user Alice and the Alarm server will use to transport text characters. `sipam` required the development of a server from the sample user agent that came with the CINEMA SIP package, using the SIP C++ libraries in the package. This reminder service was first presented at the South African Telecommunication Networks and Applications Conference (SATNAC) [Hsieh et al., 2001].

The C++ SIP library from CINEMA (libsip++) is built from the C SIP library (libsip) and consists mostly of wrapper classes containing the original C functions. The basic generic classes are the `IptelEndpoint` and `IptelCall`. `IptelEndpoint` is an abstraction of a terminal or IP portion of a gateway or any other IP telephony entity that has a fixed set of capabilities and local addresses for all calls. `IptelCall` is an abstraction of a call: it contains methods to initiate a call, alert a remote party to a call,

accept an incoming call, and more. Both `IptelEndpoint` and `IptelCall` are abstract classes and can be used independently of the protocol. Both classes were used by CINEMA to build `SIP323`, a signaling gateway. `SIP323` translates SIP messages to H.323, and vice versa. Figure 6.6 shows a partial list of methods for each class.

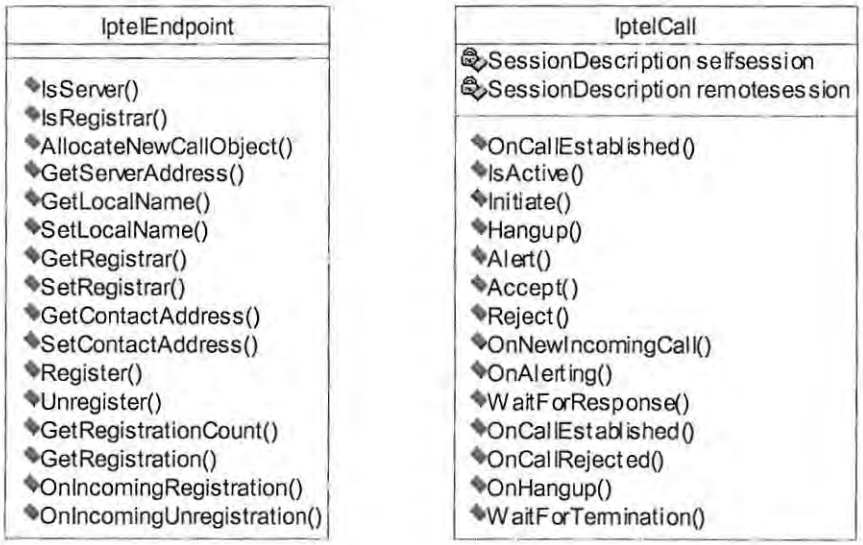


**Figure 6.6 Iptel class diagrams**

The protocol-dependent subclasses `SIPEndpoint` and `SIPCall` are derived from `IptelEndpoint` and `IptelCall`, respectively. `SIPEndpoint` represents a SIP end system, such as a SIP user agent; `SIPCall` represents a call between the local and remote entity. Figure 6.7 lists the methods associated with each of the two subclasses.

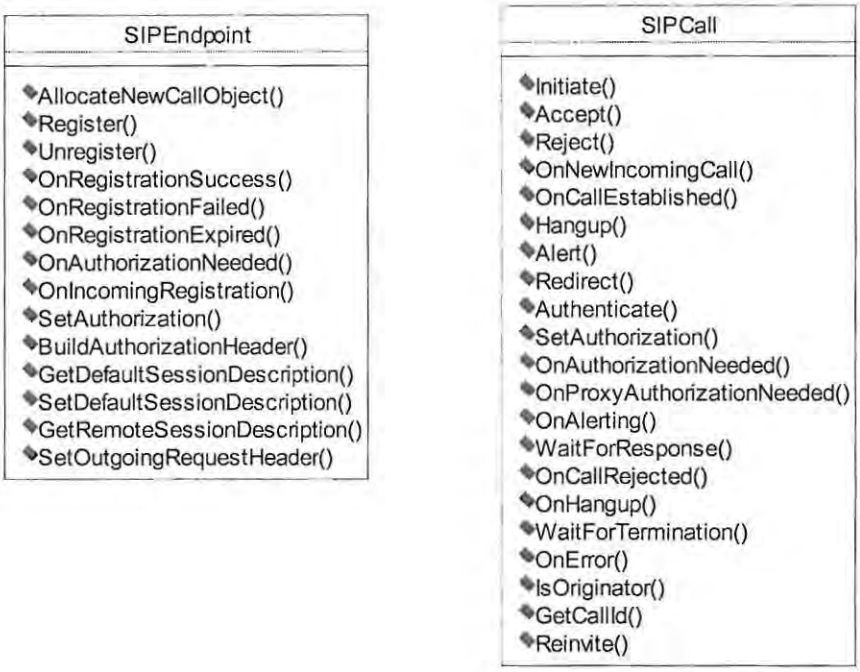


**Figure 6.7 SIPEndpoint and SIPCall methods**

Applications must derive from these classes and so implement the methods in order to access the libraries. For example, in `SIPCall` the method `OnNewIncomingCall()` is activated when `SIPCall` receives an INVITE message; the derived class must determine what to do with this message.

### 6.4.1 Alarm server (`sipam`)

The Alarm server (`sipam`) starts by instantiating the classes `MySIPCall` and `MySIPEndpoint`. `MySIPCall` and `MySIPEndpoint` are derived from `SIPCall` and `SIPEndpoint`, respectively. `MySIPCall` is responsible for handling a new call via the method `OnNewIncomingCall()`. This method is called when `SIPCall` receives an INVITE message. The method will prompt the user when a new incoming call is being received and then it will wait for the user's input. The user can accept or reject the call. In the case of the Alarm server, this method will automatically accept the call.

The program will determine from the SDP information contained inside the SIP message whether the session required is a text-based session. (The Alarm server is only capable of receiving and sending text messages from and to the caller.) The following segment of code from the Alarm server contains the method `OnNewIncomingCall()`:

```
void MySIPCall::OnNewIncomingCall(IptelEndpoint & ep)
{
  ...
  //mchsieh getting the remote description
  SessionDescription *session;
  /* Get remote IP and port */
  session = GetRemoteSessionDescription();
  /* Get remote sip url (who to call back) */
  myaddress = src;

  if ( session != NULL ) {
        cout << "Remote Session is " << *session << endl;
        MediaInfo rem_media;
/*------------------------- Audio ------------------------------*/
//Determine if the session is audio, if it is start the audiotool
...
/*------------------------ Application -----------------------------*/
//If the session is text based then accept the call and start the
//tcpserver.
//The alarm server starts the tcpserver on port 40000.
//We also need to find the ipaddress and port number of the client to
//sent to.
        if ( session->FindSession(rem_media,
            MediaInfo::Receive, MediaInfo::Application,
            MediaInfo::UDP_Transport) ) {

          //rem_addr stores the remote media ip address and port
          //number
          IptelIPAddress rem_addr;
          rem_addr = rem_media.GetRxAddress();

          //mchsieh display remote media ip address and port number
          cout << "remote media ip address and port number " <<
          rem_addr << endl;
          //mchsieh display remote media port number
          int remport = rem_addr.GetPort();
          cout << "remote media port number " << remport << endl;

          /* --- Set the self session info to send back --- */
          session = new SessionDescription;
          MediaInfo media;
          media.SetDirection(MediaInfo::Receive);
          media.SetDataType(MediaInfo::Application);
          media.SetTransportType(MediaInfo::UDP_Transport);
          /* Set self IP address and port number for the session */
          media.SetRxAddress(IptelIPAddress(SIPLibrary::HostInAddr));
          //mchsieh using chat make them standard
```

```
            media.SetRxPort(40000);
            session->AddSession(media);
            cout << "Self Session is " << *session << endl;
            call->SetSelfSessionDescription(session);

            cout << "Accepting the call..." << endl;
            //Accept the call then startup the tcpserver
            int result = 0;
            result=call->Accept();
            if(result == 0)
            {
                  cout << "MCHSIEH call accept success" << endl;
                  cout << "Starting the tcpserver " << endl;
                  tcpserver->StartTool(40000);
                  cAccepted = true;
            }
            else
            {
                  cout << "MCHSIEH call accept failure" << endl;
            }
      }//end if application
   }//end session
   else {
      cout << "No remote session in INVITE" << endl;
   }
}//end onnewincomingcall
```

This code segment uses the method `FindSession()` to locate the right session for the call. The method `Accept()` is used to accept the call (i.e., send a 200 OK message back to the caller). The `tcpserver` is a simple object that uses TCP/IP to interact with the user regarding the time the user wants to be notified. This code segment is responsible for the establishment of the session and for starting the `tcpserver`. Once the user has finished interacting with the Alarm server, the user sends a BYE message to terminate the session. The Alarm server has to catch this message in order to terminate the `tcpserver` and start the sleeping process. The method `OnHangup()` must be used to accomplish this. Thus, the method `OnHangup()` is called when the remote endpoint wants to terminate the SIP session. The following code segment shows the method `OnHangUp()`:

```
int MySIPCall::OnHangup()
{
/*
  mchsieh
  This function gets called when the remote side terminates the
  session.
```

```
  Once the transaction is completed the TransactionCompleted flag is
  set to true.
  "mytime" is used to determine if the alarm server has called back the
  user or not.
*/
  cout << "Call closed by remote" << endl;
  oBusy = false;
  call = NULL;

  SessionDescription *session;
  /* Get remote IP and port */
  session = GetRemoteSessionDescription();
  MediaInfo rem_media;

  /* audio session */
  if(session->FindSession(rem_media,
        MediaInfo::Receive, MediaInfo::Audio) ) {
     audio->StopTool();
     SIPCall::OnHangup();
  }
  /* end audio session */

  /* application session */
  if ( session->FindSession(rem_media,
       MediaInfo::Receive, MediaInfo::Application,
       MediaInfo::UDP_Transport) ) {
    /* check mytime to see the alarm has called back or not */
    FILE* myfile = fopen("dserver.txt","r");
    //mytime is a global variable
    fscanf(myfile,"Time %d",&mytime);
    fclose(myfile);
    if (mytime == 0)
      {
            oTransComp = false;
      }
    else
      {
            oTransComp = true;
      }

    cout << "MCHSIEH I am in MySIPCall::OnHangUp() " << endl;
    //mchsieh stop the tcpserver
    tcpserver->StopTool();
    oAccepted = false;

    SIPCall::OnHangup();
  }
  /* end application session */
  return 0;
}
```

The flag `oTransComp` is used to notify the Alarm server when to begin sleeping. The code for sleeping is shown next:

```
//do the sleeping
FILE* myfile = fopen("dserver.txt","rw");
fscanf(myfile,"Time %d",&mytime);//mytime is global
fprintf(myfile,"Time %d",0);
fclose(myfile);
cout << "MCHSIEH mytime = " << mytime << endl;
Sleep(mytime*1000);//sleep x secs
```

mytime is a global variable used to store the number of seconds that the process must sleep until the event time arrives. The above code segment from the main function has shown how this is done.

In the main function, once the Alarm server has waited out the duration specified by the user, it calls the user back using the audio session. The following code segment shows the statements that the Alarm server uses in order to compose an INVITE message to call back the caller:

```
/* Get the destination URL */
//myaddress is the global variable
IptelAddress dest = myaddress;
dest.SetScheme("sip");
//start the call object
MySIPCall *mycall;
mycall = new MySIPCall(sip);
call = mycall;

/* Set the session description info for the outgoing message */
SessionDescription *session;
MediaInfo media;
session = new SessionDescription;

media.SetDirection(MediaInfo::Receive);
media.SetDataType(MediaInfo::Audio);
IptelIPAddress rxaddr(SIPLibrary::HostInAddr);
media.SetRxAddress(rxaddr);
media.SetRxPort(audio->GetSelfPort());
FormatInfo f(0);
media.AddFormat(f); /* audio format PCMU */
session->AddSession(media);
cout << "Self Session " << *session << endl;
mycall->SetSelfSessionDescription(session);

//mchsieh Set the Subject
mycall->SetSubject("Call from alarm clock");

cout << "Calling " << dest << " from " << src  << endl;

mycall->Initiate(dest,src);
```

```
    oTransComp = false;
```

The variable `myaddress` contains the address of the caller and is set in the method `OnNewIncomingCall()`. This variable is used to set the destination URL of the person that the Alarm server should call. Figure 6.8 shows an output from the Alarm server that was implemented in CINEMA.

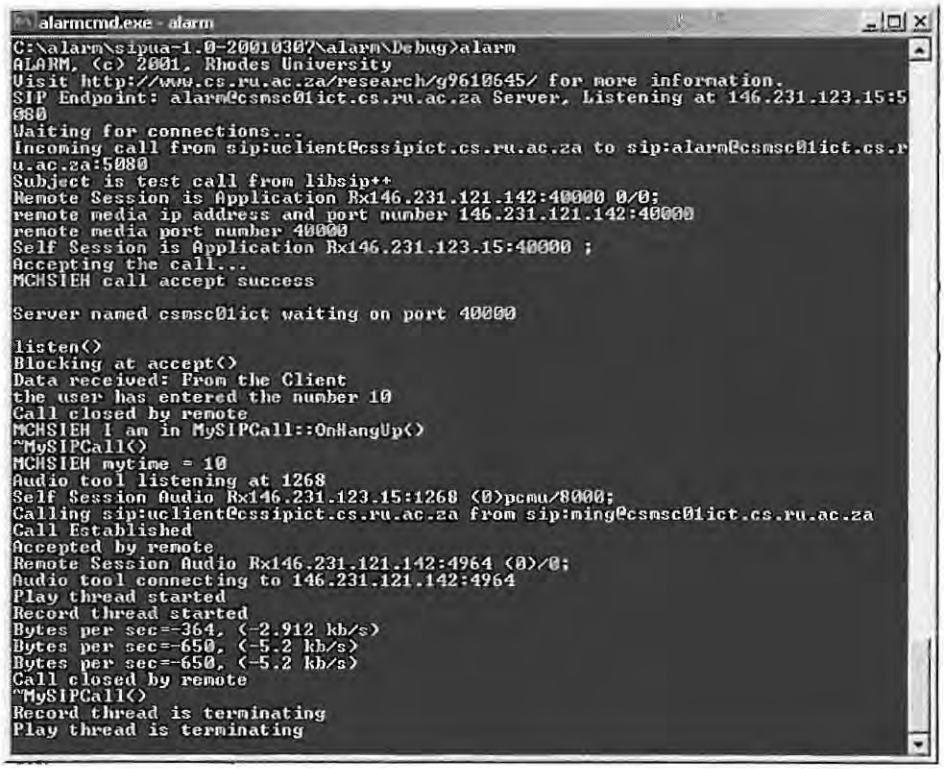


**Figure 6.8 CINEMA Alarm server output**

### 6.4.2 Alarm Client

Certain modifications had to be made to the sample user agent from CINEMA in order to add text support. The caller uses the sample user agent (sipua) to initiate the call. Once the call is established, the method `OnCallEstablished()`, from the class `MySIPCall`, is called. Then, the user agent is able to start the application to be used in the session: `audiotool` for audio, `tcpclient` for text-based communication. The

code is similar to the code in the Alarm server; the relevant code segment from the method `OnCallEstablished()` is shown here:

```
/*------------------------ Application -------------------------*/
if(session->FindSession(rem_media, MediaInfo::Receive,
   MediaInfo::Application, MediaInfo::UDP_Transport)) {
        cout << "MCHSIEH I am in application at the moment " << endl;
        /* get the IP address and port no */
        IptelIPAddress rem_addr = rem_media.GetRxAddress();
        cout << "remote media ip address and port number " << rem_addr
             << endl;
        struct sockaddr_in addr;
        /* remote media ip address */
        rem_addr.GetIpAddress(addr);
        /* remote media port no */
        addr.sin_port = htons(rem_media.GetRxPort());

        /* another way to get the ip address */
        //rem_ipaddress is global
        rem_ipaddress = inet_ntoa(addr.sin_addr);
        cout << "remote media ip address " << rem_ipaddress << endl;
        //we need to get the port number as well
        rem_portno = rem_addr.GetPort();
        cout << "remote media port no " << rem_portno << endl;

        tcpclient->StartTool(rem_ipaddress,rem_portno);
        oACKsent = true;

    }//end if application
```

When the Alarm Client sends the initial INVITE to the Alarm server, the session description has to be `Media::Application` to specify that a text session will be used for communication. This is shown in the following code segment:

```
    media.SetDirection(MediaInfo::Receive);
    media.SetDataType(MediaInfo::Application);//set the text session
    media.SetTransportType(MediaInfo::UDP_Transport);
    IptelIPAddress rxaddr(SIPLibrary::HostInAddr);
    media.SetRxAddress(rxaddr);
    media.SetRxPort(40000);//40000 for chatting
    session->AddSession(media);
    cout << "Self Session " << *session << endl;
```

Once the user has entered a time for notification, the user terminates the session by entering "bye". Figure 6.9 shows a screenshot of the output from the Alarm Client. The

reminder service also works with a graphical user interface, shown in Figure 6.10. This was created with `sipc v1.51`.


**Figure 6.9 CINEMA Alarm Client output**

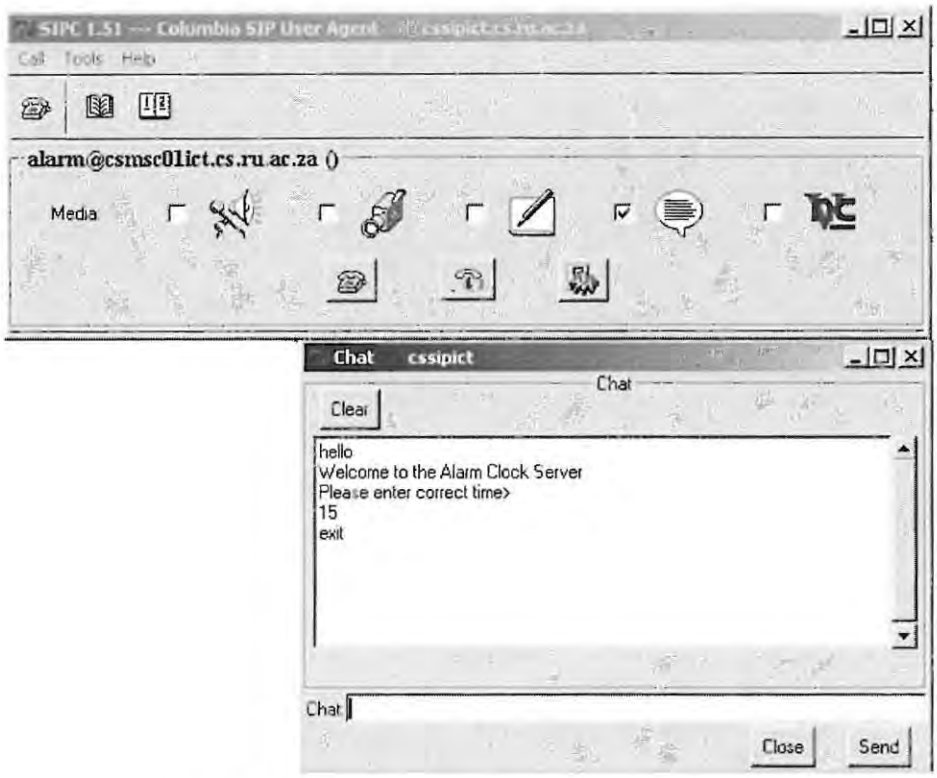


**Figure 6.10 CINEMA `sipc` reminder service output**

Figure 6.10 shows the screen during a chat interaction between a caller and the reminder service. Certain modifications were required in order to set the correct port numbers for the chat program. Thus, this reminder service is generic and can be accessed via standard interfaces. Any SIP user agent with a chat program can access the reminder service, and the GUI provides a user-friendly interface with the service.

## 6.5 VOCAL's reminder service

The VOCAL reminder service is in fact an alarm clock service that reminds the user of a specific event that is about to occur. Thus alarm, as a keyword, is used in various parts of the text and code. Implementation of a reminder service in VOCAL required modification of the voicemail server and the voicemail client into an Alarm server (`amserver`) and an Alarm Client (`amclient`), respectively. The architecture of the system remained the same as that for the voicemail service as shown in Figure 6.3. A

protocol to be used between amserver and amclient had to be chosen. We kept the original Voice Mail Control Protocol (VMCP) because this allowed the reminder service extra functions, such as PLAYFILE and RECORDFILE. Thus, the reminder service in VOCAL was given extra functionality, allowing the user not only the option to choose a time to be alerted of an event but also to record a voice message that can be read back to the user at notification time. The implementation of this service was discussed and published in the proceedings of the South African Telecommunication Networks and Applications Conference (SATNAC) [Hsieh et al., 2002].

### 6.5.1 Alarm server (amserver)

The Alarm server (amserver) communicates with the Alarm Client using Voice Mail Control Protocol (VMCP). The Alarm server uses an explicit finite state machine to control the behavior of the Alarm Client. (A basic understanding of finite state machines is required in order to modify the voicemail server into an Alarm server.) No modifications to VMCP are required for this service. In the original voicemail server the sequence of states is as follows: StatePlayGreeting -> StateRecordMessage -> StateEndofSession. The name of the states describes each state's function. For example, StatePlayGreeting plays the greeting message for the caller. A new state was added for the reminder service: StatePlayGreeting -> StateRecordDtmf -> StateRecordMessage -> StateEndofSession. The state StateRecord Dtmf can record the time of an event inputted by the user via Dual-Tone MultiFrequency (DTMF). The input string is of the form "time of event" followed by "#". Once "#" is detected, amserver goes to the next state, StateRecordMessage. The code for processing the DTMF is shown in the following code segment:

```
int
StateRecordDtmf::ProcessDTMF (pEvent evt)
{
  using std::fstream;
  static string thetime;
  cpLog (LOG_DEBUG, "processdtmf received DTMF.");
  cpLog (LOG_DEBUG, "received dtmf char %c ",(*evt->IParm())[0]);
  char receiveddtmf = (*evt->IParm())[0];
```

```
if(receiveddtmf=='#')
{
    //end of the input string opening a file to write the time to
    std::ofstream timefile ("/tmp/time.txt");
    if (timefile.is_open())
    {
          timefile << "Time ";
          timefile << thetime;
          timefile << endl;
          timefile << "CallerString ";
          //get the caller's url store it in a string
          string callerstr = ((VmSession*)getSession())->
                             getLine()->getVmcp ()->
                             getSessionInfo ().CallerId.c_str ();
          timefile << callerstr;
          timefile << endl;
          cpLog(LOG_DEBUG,"recorded dtmf %s",thetime.c_str());
          timefile.close();
    }

    //end of StateRecordDtmf goto StateRecordMessage
    StateRecordMessage recordMessage ("StateRecordMessage");
    return recordMessage.Process (getSession());
}
else
{
    //still inputting the time concat to the time string
    thetime = thetime + receiveddtmf;
    //here we must send the digits back to amclient for confirmation
    getSession()->getLine()->getVmcp()->sendDtmf(receiveddtmf);
    return StateStay;//mchsieh stay in the current state
}
}//end ProcessDTMF
```

The use of DTMF in this system is important because it allows the service to be accessed from PSTN networks. PSTN networks use DTMF tones to communicate the number to be dialed. Other services such Interactive Voice Response (IVR) also use DTMF to transport the user's input.

### 6.5.2 Alarm Client

The Alarm Client is based on the user agent code from VOCAL. More information about the structure of the user agent can be found at [Vovida, 2001c].

The user agent from VOCAL consists of two sections, one section to model the behavior of a phone and another to handle the devices controlled by the user agent. The user agent

uses an explicit finite state machine to model the behavior of a phone. The user agent can handle different devices such as `SoundCardDevice` and `VmcpDevice`, which it uses to communicate with `vmserver`. No modifications are required to the state machine for the reminder service. However, a new device had to be added to implement the reminder service: the name `AlarmDevice` was chosen.

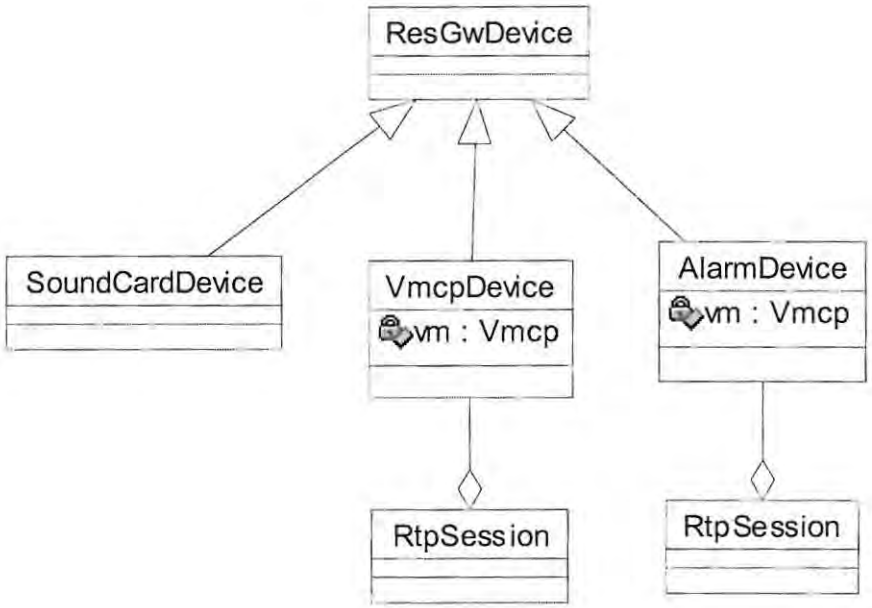


**Figure 6.11 VOCAL devices class diagram**

All the devices listed in Figure 6.11 are inherited from the Residential Gateway Device (`ResGwDevice`). The complete class diagram for the user agent can be found at [Vovida, 2001c]. The `AlarmDevice` must communicate with the Alarm server using VMCP and be able to send the digits for the time of the event. Both the `VmcpDevice` and the `AlarmDevice` hold links to the RTP stack via the class `RtpSession`. Modifications are required to the user agent to correctly use this new device. Options to specify the location of the Alarm server are also provided in the configuration file.

The `AlarmDevice` is activated when the `process()` function is called. The behavior of the device is determined by the VMCP messages that it receives from the Alarm

server. A switch statement is used to switch between the different VMCP messages. When the AlarmDevice receives a VMCP CLOSE message it must remember to call back by calling the function provideCallBack(), as shown in the following code segment:

```
int
AlarmDevice::process (fd_set* fd)
{
   ...
   switch (Msg)
   {
            case Vmcp::Close:
            {
                cpLog(LOG_DEBUG,"VMCP:Close");
                reportEvent(sessionQ, DeviceEventHookDown);
                hookStateOffhook = false;
                am.sendClose();
                close(ss);
                deviceMutex.unlock();

                //mchsieh when it gets here the amserver is telling
                //the amclient to terminate the session with the ua
                //remember to call back
                provideCallBack();

                return 0;
            }
            ...
   }//end switch
   ...
}//end process
```

Functions such as recvRTPDTMF() and provideDtmf(), for receiving and sending DTMF tones, are included in AlarmDevice. The AlarmDevice receives DTMF tones from the caller and sends DTMF tones to the Alarm server.

The function provideCallBack() must:

1. capture the device event;
2. read the event time stored in the file;
3. sleep for the required amount of time;
4. wake up and report the event DeviceEventHookUp to the state machine;
5. compose the SIP URL for the calling destination and then report it;

6. set the `PlayFile` flag, so that once the session is established it can play the recorded message.

Notice that reporting the event `DeviceEventHookUp` triggers the user agent to initiate a call to a second party. The function `provideCallBack()` is given in the following code segment; the comments included with the code segment explain what each statement does, reflecting the six points stated above.

```
void
AlarmDevice::provideCallBack()
{
        //capture the device event
        Sptr < UaDeviceEvent > event = new UaDeviceEvent( sessionQ ) ;
        assert( event != 0 );
        cpLog(LOG_DEBUG,"MCHSIEH i am in provideCallBack");
        //=====================================================
        //simple procedure to read in the time from the file
        int myinttime;
        string buffer, sleeptime, callerstring;
        ifstream timefileread ("/tmp/time.txt");
        if (! timefileread.is_open())
        {
            cout << "Error opening file";
            exit (1);
        }

        while (! timefileread.eof() )
        {
            timefileread >> buffer;
            if(buffer == "Time")
            {
                  timefileread >> sleeptime;
            }
            if(buffer == "CallerString")
            {
                  timefileread >> callerstring;
            }
        }
        //convert from string to int
        std::istringstream inputstream(sleeptime);
        inputstream >> myinttime;
        sleep(myinttime);//sleep for myinttime seconds
        cpLog(LOG_DEBUG,"MCHSIEH reportEvent DeviceEventHookUp");
        //=====================================================
        //report the event DeviceEventHookUp
        reportEvent(sessionQ, DeviceEventHookUp);
        hookStateOffhook = true;
        event->type = DeviceEventHookUp;

        if (event->type != DeviceEventUndefined)
```

```
    {
        assert( sessionQ != 0 );
        event->callId = callId;

        #ifndef WIN32
        sessionQ->add( event );
        #else
        Sptr <SipProxyEvent> proxyEvent;
        proxyEvent.dynamicCast( event );
        sessionQ->add( proxyEvent );
        #endif
    }
    //=====================================================
    //compose the destination URL for example
    //Data newTextEntry ="sip:6399@146.231.123.15:5060";
    Data newTextEntry ="sip:" + callerstring + ":5060";
    cpLog( LOG_DEBUG, "URL is %s", newTextEntry.getData() );

    event->type = DeviceEventCallUrl;
    event->text = newTextEntry;

    if (event->type != DeviceEventUndefined)
    {
        assert( sessionQ != 0 );
        event->callId = callId;

        #ifndef WIN32
        sessionQ->add( event );
        #else
        Sptr <SipProxyEvent> proxyEvent;
        proxyEvent.dynamicCast( event );
        sessionQ->add( proxyEvent );
        #endif
    }
    //=====================================================
    //remember to set the PlayFile flag
    PlayFile = true;
}//end provideCallBack
```

## 6.6 Interactive services discussion

Both CINEMA and VOCAL provide good APIs for programmers to modify the user agent. Programmers with a sound understanding of C++ can create their own interactive services by modifying user agents.

The investigation into interactive service creation uncovers that there is no standard way of creating interactive services. The standard mechanisms (see Chapter 5) SIP-CGI and CPL are not applicable here because they are concerned with how a call is handled, such

as how to forward or block a call. Those mechanisms do not directly provide for interaction whereby users can input data at the time the service is executed. One solution would be to provide a predefined interface between the user agent and the server supporting the services. User agents invoking client-related methods defined in the interface would expect results from the server in a certain format, and, similarly, from the server invoking the server-related methods in the interface. The interface would be valid only for the media part of the session and would not relate to the signaling part of the session, which is already handled by SIP-CGI and CPL.

The interface suggested above would be designed to aid programmers creating interactive services. However, a new language would be required for users to create interactive services. This language would have flexibility and ease of use, as do SIP-CGI and CPL, and it would have the interactive interface previously described. One foreseen difficulty in designing a new language concerns trying to determine what the user's input will be from the server side. For example, in creating the reminder service we necessarily had to determine what kind of input the user would need to communicate with the server: would it be DTMF rather than RTP or else a normal data stream using TCP? Choosing RTP meant that all endpoints would have to be RTP capable, but this would restrict users without a RTP-capable media agent. The same consequence would occur if the data stream using TCP were chosen. RTP was ultimately chosen for creating the reminder service in VOCAL, because the major benefit was that this service could be accessed by users on the PSTN network via a SIP to PSTN translator. Designing a new language for users to create interactive services can be a topic for future research.

Another extension to the study of interactive services would be an investigation into the interoperability of interactive services with other services. For example, could a call-forwarding service work with a voicemail service and notification service in general?

## 6.7 Summary

This chapter introduced interactive services, looking first at CINEMA and VOCAL voicemail services. We used this as a starting point to investigate how new interactive services may be implemented for CINEMA and VOCAL. As a practical exercise, we implemented a reminder service for both environments. We concluded with a brief discussion on interactive service creation, and proposed the creation of a generalized interface for creating new interactive services.

# Chapter 7 Internetworking Services

## 7.1 Introduction

An investigation into internetworking services is especially relevant in the context of service creation because it can bridge the SIP, H.323 and MGCP networks, and so extend the reach of basic services (see Chapter 4), and also because it gives each of the networks the ability to access services on the other networks.

Many papers have been written comparing the various advantages and disadvantages of the two competing Internet telephony signaling protocols, SIP and H.323. Notable are those by [Dalgic and Fang, 1999], [Rosenberg and Schulzrinne, 1998b], [Nortel Networks, 2000a] and [Wind River, 2002]. We sidestep this interesting debate by focusing on the issue from an internetworking and service perspective. The importance of internetworking is shown in [Glasmann et al., 2003] where it is noted that since H.323 is the more mature standard it has achieved smooth internetworking with the PSTN, with clear advantages for IP telephony, at least in the short to medium term.

The Media Gateway Control Protocol (MGCP) is another important protocol in Internet telephony. MGCP is a protocol used to control media gateways. In this chapter we also focus on internetworking between SIP and MGCP.

Both CINEMA and VOCAL provide gateways that allow SIP and H.323 networks to interoperate with each other; that is, they allow users from either network to communicate with each other and possibly to access services on the other network. As for SIP-MGCP translators, none was available in CINEMA while one was available in VOCAL. The SIP-MGCP translator in VOCAL was in a developmental stage and so not fully functional; hence, we decided to develop our own.

In this chapter we discuss the operation of the internetworking mechanisms already available in CINEMA and VOCAL. After that, we introduce the MGCP protocol and describe our development of an internetworking service between SIP and MGCP.

## 7.2 CINEMA internetworking

CINEMA's internetwork server is called `SIP323`. The internetwork server acts as a signaling gateway between a SIP and a H.323 network, by working either with an external H.323 gatekeeper or by using its own built-in H.323 gatekeeper. On the SIP side of the network the signaling gateway acts as a SIP proxy. When the server starts up it can be configured to register with a SIP registrar and a H.323 gatekeeper, thus acting as a visible entity in both environments.

The internetwork server `SIP323` uses the OpenH323 v2 library provided by [OpenH323, 2003]. At the time of testing this server CINEMA had just discontinued free licenses previously issued to academic institutions. Professor Henning Schulzrinne of Columbia University, a co-author of the SIP RFC and a founder of the CINEMA project, chose to release the software commercially to SIP Communications Ltd. (SIPCOMM). Thus, the software suite from Columbia University was released for commercial purposes. As a result, we were left with only a single call license version of the gateway.

### 7.2.1 `SIP323` Operation

The call system for the internetwork server `SIP323` is based on aliases. If a user were registered on the SIP registrar as Bob@work.com then normally he would be given the alias Bob on `SIP323`. (If the alias Bob were already taken, Bob@work.com would not be able to register with the system, but would have to change his alias to register.) Nonetheless, calls from the H.323 network directed at Bob will be redirected to Bob@work.com, and vice versa from the SIP network. In this case `SIP323` is configured to work as a H.323 gatekeeper using its built-in feature.

A successful call was set up between a single SIP endpoint and a single H.323 endpoint using `SIP323` as a signaling gateway. The signaling gateway was set up statically; that is, all calls from the SIP side were directed to a H.323 endpoint specified beforehand, and vice versa. This setup was done with the H.323 endpoint designated as csmsc01ict.cs.ru.ac.za (146.231.123.15) and the SIP endpoint as edo.dsl.ru.ac.za (146.231.112.107).

Figure 7.1 depicts a call between `Netmeeting` (a H.323 client) and `sipc` (a SIP user agent). The figure shows that the H.323 endpoint is calling cssipict.cs.ru.ac.za (146.231.121.142), which is the `SIP323` server. In `Netmeeting` the SIP endpoint is shown as ming@edo.dsl.ru.ac.za, which is the exact location for user Ming, while in `sipc` the H.323 endpoint is shown as ming@cssipict.cs.ru.ac.za (the username in this case is less important than the hostname, cssipict.cs.ru.ac.za, which is the location of the `SIP323` server).

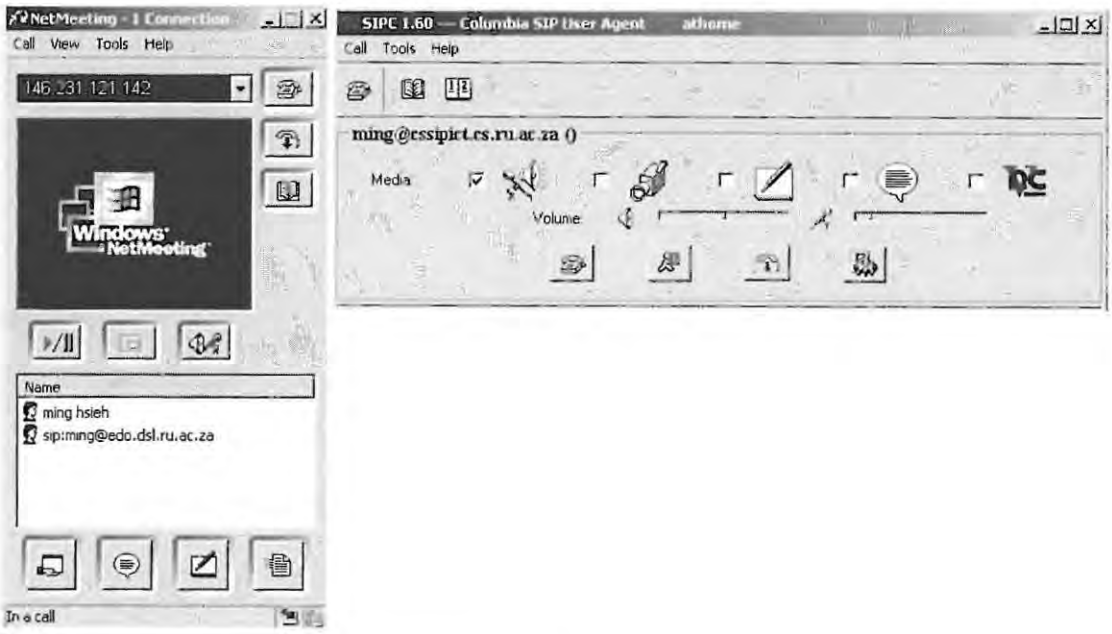


**Figure 7.1 `SIP323` (`Netmeeting` and `sipc`) operation**

SIP323 has a debugging option that shows the various messages exchanged during the set up of the call. However, the output shows only the SIP messages exchanged between SIP323 and sipc, as well as the API calls that were made in order to access the CINEMA library (such as Initiate and WaitForResponse). No H.323 messages output were available in order to verify what exact H.323 messages were sent. A possible way around this problem is to set up a packet sniffer that is able to decode H.323 packets, and capture the packets sent to and from SIP323.

Due to the limitations of the single-license version of the signaling gateway, no experiments could be done to test service interoperability.

## 7.3 VOCAL internetworking

The internetworking services in VOCAL are called translators. VOCAL provides these to translate H.323 and MGCP messages to SIP, and vice versa, in order that H.323 and MGCP endpoints may access the SIP network. Because the MGCP translator in VOCAL was still under development, we decided to develop our own SIP-MGCP internetworking service.

VOCAL's H.323 translator server is called SIPH323CSGW, a SIP-H.323 Call Signaling Gateway. SIPH323CSGW uses the OpenH323 v4 libraries [OpenH323, 2003].

SIPH323CSGW interacts with VOCAL in order to correctly provide access to the SIP network. Users on both sides, SIP and H.323, must be correctly registered with VOCAL in order to use the translator. (There are two possible views in the VOCAL provisioning GUI, the Administrator GUI and the Technician GUI; the administrator generates the users and the technician sets up the servers.)

The essential VOCAL components involved to set up internetworking services are the Marshal, Redirect and SIPH323CSGW servers. The Marshal server proxies the SIP requests to the Redirect server. The Redirect server has to be correctly configured via the

Technician GUI in order to redirect the requests to SIPH323CSGW. SIPH323CSGW needs to be correctly set up so that it knows which SIP port number to listen to and to which SIP URL to send SIP requests. The H.323 port numbers, on the other hand, are standard and cannot be changed. The configuration process is explained in detail in the next section.

### 7.3.1 SIPH323CSGW Operation

The following steps were taken to configure the translator (SIPH323CSGW):

1. Make certain that the executable siph323csgw are in the correct directory. This involved compiling the correct project from the VOCAL project directory and placing the executable in the directory /usr/local/vocal/bin together with the configuration file siph323csgw.conf.
2. Configure the executable so that the server can run correctly. This involved editing the configuration file siph323csgw.conf. The options configured were gatekeeper ID, endpoint ID, SIP port and SIP remote IP address. The SIPH323CSGW will listen on SIP port number 5155, and the SIP remote IP address is the address where SIPH323CSGW will forward all requests from the H.323 side. This address can be the address of the Marshal server or Proxy server.
3. Start VOCAL and SIPH323CSGW separately; that is, ./vocalstart start, and then ./siph323csgw -f siph323csgw.conf.
4. Use the Administrator GUI for VOCAL to create a user for the H.323 endpoint; for example, to associate the number 3000 to the endpoint 146.231.112.107, the user 3000 was edited so that it uses Access List authentication type, and the IP address of SIPH323CSGW was entered as the server at which user 3000 must authenticate.
5. Use the VOCAL Technician GUI to create a digital dial plan (as for 3000, to continue the example above). Digital dial plans are similar to the dial patterns for the user agents; they allow the Redirect server to redirect the calls to the correct destination. For example, entering the options

   Key: ^sip:107

Contact: sip:3000@146.231.123.15:5155;user=phone

will forward calls with the username (or number) 107 in the SIP URL to `SIPH323CSGW`, and the calls will be directed at the user 3000. Now, calls to 107@146.231.123.15 will be forwarded to 3000@146.231.123.15:5155. This method ensures that the Redirect server will redirect calls to `SIPH323CSGW`. Notice that the SIP port chosen is 5155. This differs from the standard SIP port number 5060, because the Marshal server, which uses the port number 5060, and `SIPH323CSGW` in the example are running on the same machine and a conflict would occur if they both ran on the same port number.

6. Configure `Netmeeting` (a H.323 endpoint) to use the gatekeeper to make calls and to register with the gatekeeper (e.g., as the phone number 3000).
7. Configure the VOCAL user agent to register and proxy all calls via VOCAL. Ensure that a user number has been set up for the user agent (e.g., a user with the number 2000). Also, set up the dial pattern to use a number, for example the number 107, to dial the H.323 endpoint. Thus, entering the option Dial_Pattern string 2 ^107 will make the call to 107@146.231.123.15, which will be appropriately forwarded by the Redirect server.
8. Start `Netmeeting` and the VOCAL user agent and ensure that they have been configured as previously described. To make a call from the user agent, type "a" to go offhook, followed by "107" to call the number 107. To make a call from `Netmeeting` type the number "2000".

Figure 7.2 shows the various messages exchanged during the set up of a call from a SIP endpoint to a H.323 endpoint. The terms TCSAck and TCSet stand for Terminal Capabilities Set Acknowledgement and Terminal Capabilies Set, respectively. The terms OLC and OLCAck stand for Open Logical Channel and Open Logical Channel Acknowledgement, respectively.

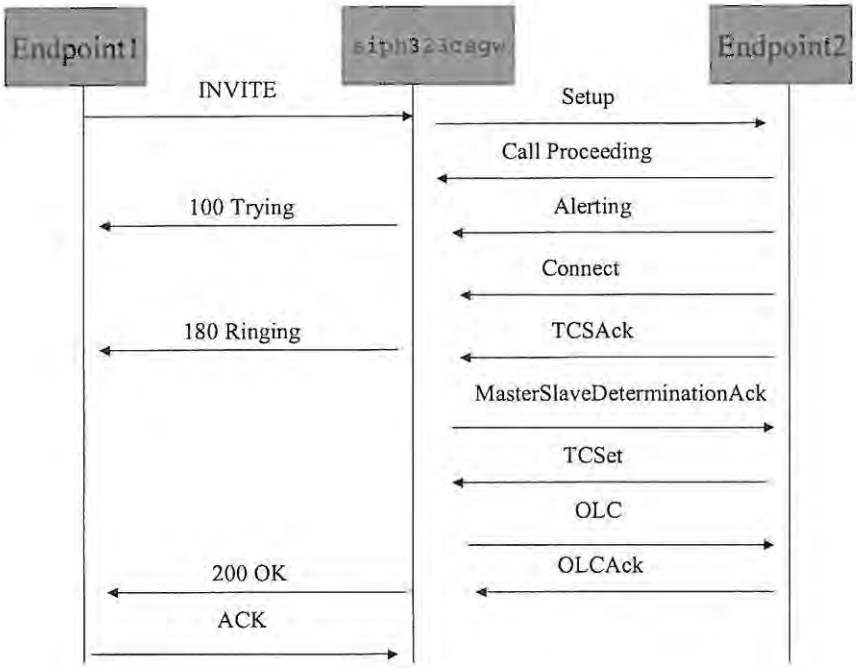


**Figure 7.2 VOCAL `SIPH323CSGW` message exchange**

### 7.3.2 Accessing services via `SIPH323CSGW`

In this section we examine whether or not the services in VOCAL may be executed from the H.323 network using the internetwork server. Users in VOCAL are usually assigned with numbers, similar to telephone numbers. In the next few experiments, SIP users are assigned numbers in the 2xxx range, while H.323 users are assigned numbers in the 3xxx range (where "x" can be in the range 0-9). Figure 7.3 shows the general setup and the different participants for the exercises described below.

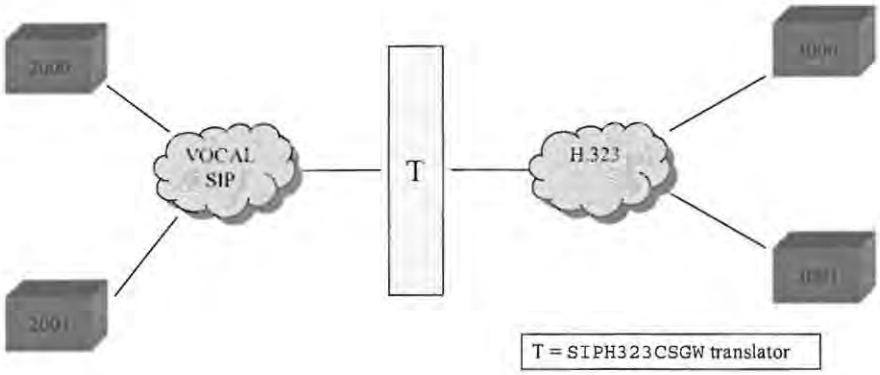


**Figure 7.3 `SIPH323CSGW` call-forwarding scenario**

**Call Forwarding—**

*Call from H.323 to SIP*

The first exercise involves a call from a H.323 endpoint to a SIP endpoint that has call-forwarding service enabled. The intention was to successfully forward the call to the second SIP endpoint. Here, the three endpoints involved are 2000 (SIP), 2001 (SIP) and 3000 (H.323). The experiment was set up by configuring the first SIP endpoint (2000) to forward all calls to the second endpoint (2001). When a call was attempted from 3000 to 2000, the call was forwarded to 2001. The execution of the service in this scenario was successful.

*Call from SIP to H.323*

Since the H.323 endpoint is considered by VOCAL as another entity in its environment, we examined whether or not call forwarding can be enabled for a H.323 endpoint via the SIP network. The three endpoints involved were 3000 (H.323), 3001 (H.323) and 2000 (SIP). User 3000 was set to forward all calls to user 3001 in the SIP provisioning server. A call was attempted, and set up, from 2000 to 3000. As expected, the call-forwarding

service was not executed. In order for VOCAL to correctly redirect the call to the translator, a digital dial plan had to be used (sip:3000@146.231.123.15:5155; user=phone); this essentially bypasses the feature server in the SIP network, which contains the information about forwarding the call, and the service could not be executed for user 3000. An obvious and principled solution to that problem would be to have a gatekeeper on the H.323 network, to enable call forwarding (e.g., for user 3000).

**Call Screening—**

*Call from H.323 to SIP*

This scenario explores whether or not call screening can work from both networks. A H.323 endpoint (3000) was configured to use `SIPH323CSGW` to make calls to the SIP network. A SIP endpoint 2000 was provisioned to screen out incoming calls from user 3000. A call attempted from the H.323 endpoint was successfully rejected by `SIPH323CSGW`. The result of this experiment shows that call screening works with this particular setup. The call was made from the H.323 endpoint and forwarded by the translator; the translator forwarded the call to the Marshal server. As a result, the service was executed because the call was redirected to the Feature server.

*Call from SIP to H.323*

A similar setup for the H.323 endpoint was used to see if the call-screening feature could be provisioned for the H.323 endpoint. The H.323 endpoint (3000) was provisioned with call screening enabled. User 3000 configured the call-screening feature, using the SIP provisioning server, to screen out calls from the SIP endpoint 2000. Not surprisingly, the result was the same as in the call-forwarding scenario above (calling from SIP to H.323): the call was completed between the H.323 endpoint and the SIP endpoint, not the intended result. This occurred because the call was redirected directly to the translator and the Feature server was not being used. As before, an obvious and principled solution would be to have call screening for user 3000 enabled in the H.323 network.

**PSTN Gateway Access—**

In this experiment we were able to use the H.323 translator to access a service in H.323 from SIP. The service in H.323 was an ISDN gateway. `SIPH323CSGW` was configured to forward calls from SIP to PSTN via the ISDN gateway. This was accomplished by Jason Penton, at the Department of Computer Science, Rhodes University, who successfully set up the gateway `OpenISDNGw` [Penton et al., 2001a]. The `OpenISDNGw` gateway was developed by Carlos Sevilla (csevilla@inf.uc3m.es) using the OpenH323 library. The gateway allows users on the SIP network to make calls to the PSTN network. Special options were set in the VOCAL provisioning GUI for this service:

Key: ^sip:0.{9}

Contact: sip:$USER@146.231.123.15:5155;user=phone

Similar changes were made to the dial-pattern configuration for the VOCAL user agent to reflect this dial option. This forces the users to dial the "09" prefix before dialing a PSTN number, in order to distinguish PSTN numbers from internal SIP phone numbers. The ability for users to make calls to the PSTN is a good example of the usefulness of internetworking services.

## 7.4 Media Gateway Control Protocol (MGCP)

MGCP is a protocol used to control telephony gateways from external control elements called call agents. Telephony gateways are VoIP gateways that provide conversion between signals on telephone circuits to IP packets on the Internet and vice versa. Numerous examples of other types of gateways that can be controlled using MGCP are provided in RFC2705 [Arango et al., 1999]. A tutorial description of MGCP can be found in [Allen, 2000].

MGCP was born out of the need for traditional telephony networks to inter-work with IP-based networks. Signaling gateways, which handled the D-channel signaling in traditional telephony, had to interact with media gateways that did the conversion of the B-channel media in traditional telephony [Radvision Corporation, 2002]. Consequently, a signaling gateway had to use a different protocol each time it needed to communicate

with a different media gateway. MGCP was introduced to solve this problem. Signaling gateways interacted with call agents by using their base signaling protocol while the call agent interacted with media gateways by using MGCP. This approach to internetworking closely follows the route of traditional telephony. The approach has also been used by Nortel Networks [Nortel Networks, 2000b], whereby signaling is handled separately from the media.

One advantage of using MGCP is that old technologies can be recycled. For example, telephones with RJ-11 connections can be used to connect with residential gateways and make calls on an IP network. The residential gateway, which does the conversion of the media and sends the signals or events to the call agent, can be controlled via MGCP [Radvision Corporation, 2002]. The signals or events being sent to the call agent are: phone off hook and on hook, and DTMF tones.

Due to the various capabilities of MGCP, such as the capability to access gateways in a standard manner, a decision was made to provide a SIPMGCP translator for SIP users. As a practical example, the translator could be used to allow SIP users access to the SMS gateway setup. The SMS gateway was set up in the Department of Computer Science, Rhodes University, by Ashley Jacobs [Jacobs and Clayton, 2002]. Since an MGCP stack was only available in VOCAL, it was chosen as the development environment.

### 7.4.1 Introduction to MGCP

Call agents communicate with gateways, which in turn communicate with endpoints. According to [Arango et al., 1999], examples of possible endpoints that can be connected to the gateways are:

1. Digital channels – provide 8Khz*8bit services as in ISDN lines.
2. Analog lines – classical telephony units, phones with RJ11 connections.
3. Interactive Voice Response (IVR) endpoints – provide access to IVR services, allow special announcements to be played or allow users to record messages.

MGCP is a master-slave protocol where call agents act like the masters controlling the media gateways, which are considered the slaves. Gateways communicate with call agents using MGCP, while call agents communicate with the rest of the network using any commonly known signaling protocols (usually SIP or H.323). There are eight commands that can be used:

1. `NotificationRequest` or `RequestForNotification` (`RQNT`) – used by the call agent to tell the gateway to notify it of specified events. This command is commonly used to determine whether or not an endpoint is available.
2. `CreateConnection` (`CRCX`) – tells the gateway to create a connection on the specified endpoint.
3. `ModifyConnection` (`MDCX`) – tells the gateway to modify a connection based on a new session description.
4. `DeleteConnection` (`DLCX`) – usually issued by the call agent but can also be issued by the gateway, if the gateway experiences problems with a connection to an endpoint.
5. `Notify` (`NTFY`) – issued by the gateway to the call agent to notify it of an event that occurred from one of its endpoints.
6. `AuditEndpoint` (`AUEP`) – used by the call agent to determine the status of an endpoint.
7. `AuditConnection` (`AUCX`) – used by the call agent to retrieve the parameters associated with a connection.
8. `RestartInProgress` (`RSIP`) – used by the gateway to indicate to the call agent that an endpoint or a group of endpoints will be taken out of service or be restarted.

Each command in MGCP must be followed by a response. This is usually a status code message telling either party whether a command has been successfully or unsuccessfully completed. This is usually a 200 ok message and can contain SDP information. More information about other possible status code messages can be found in RFC2705 [Arango et al., 1999].

We use a normal call flow to explain how MGCP works.

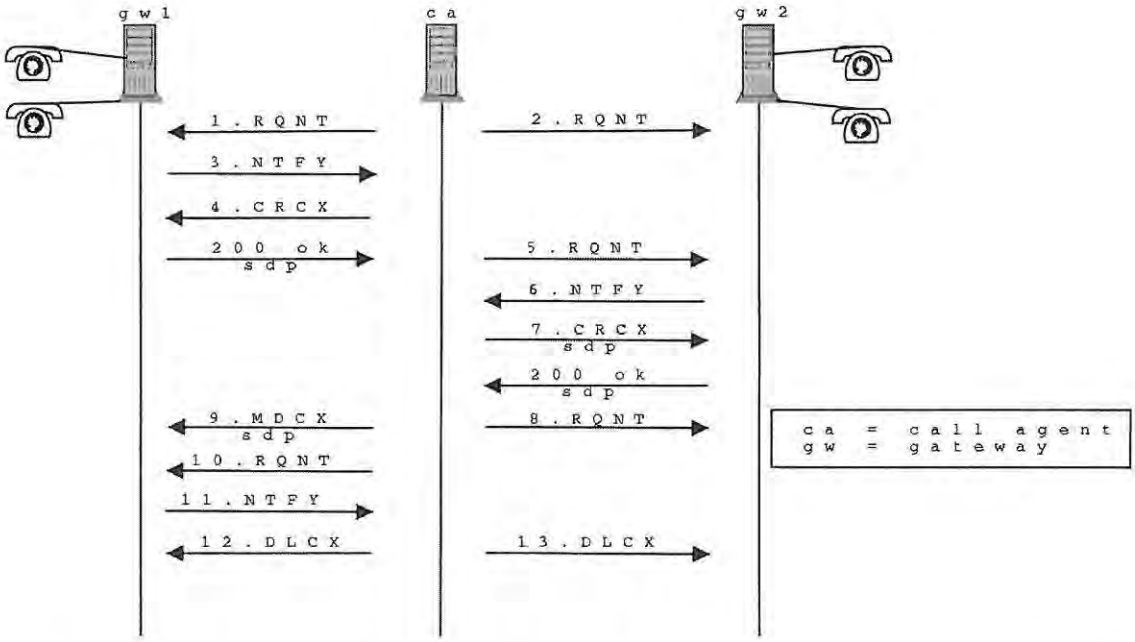


**Figure 7.4 Example MGCP call flow (for abbreviations, see list of commands in section 7.4.1)**

The steps depicted in Figure 7.4 are explained below:

Steps 1 and 2 are request notification messages; they are used to determine if there are any endpoints attached to the gateways and to request the gateways if it detects any *event packages*, such as hook up and hook down or DTMF tones. The usual response would be a 200 ok status message to show that they are alive. (The command messages in this diagram are usually answered with a 200 ok status message unless stated otherwise.)

In step 3, gateway 1 (gw1) detects an off-hook signal from one of its endpoints and sends this information as an event package to the call agent.

In step 4, upon receiving this event package, the call agent immediately sends the command `CreateConnection` (`CRCX`) to the gateway instructing it set up the connection and to play the dial tone. The gateway responds with a 200 ok message containing the SDP with information of the media types and ports that it is listening on.

This 200 ok message is different from the other 200 ok mentioned previously, because it contains SDP information.

In step 5 the call agent sends a `RequestForNotification` (RQNT) to gateway 2 (gw2). This will request the gateway to play the ringing tone on the specified endpoint and will notify the call agent if the gateway detects any off-hook event.

In step 6 the off-hook event is detected, packaged and sent to the call agent. This means that an endpoint attached to gateway 2 has answered the call.

In step 7 the command `CreateConnection` (CRCX) is sent to gateway 2 with the SDP information from gateway 1. Gateway 2 responds with a 200 ok status message with SDP information regarding the media types and ports it is listening to.

Step 8 is a notification request message to the gateway to notify the call agent if it detects any on-hook (hu) event.

In step 9 the SDP information from gateway 2 is sent to gateway 1 in a `ModifyConnection` (MDCX) message. This is to notify gateway 1 what types of media gateway 2 is willing to send and receive.

Step 10 is a notification request message to the gateway to notify the call agent if it detects any on-hook (hu) event. At this point a call has been set up between the two endpoints, each attached to their respective gateways, with the necessary media information for the communication to flow between the gateways. It should be emphasized that the endpoints do not communicate directly to each other, they communicate via the gateways.

In step 11, gateway 1 detects an on-hook (hu) event and sends this to the call agent. This means that the endpoint attached to gateway 1 has resolved to terminate the call.

In steps 12 and 13, `DeleteConnection` (DLCX) messages are sent to gateway 1 and gateway 2 in order to terminate the call and the connection between the two gateways.

Call agents must manage many calls, and gateways must manage many connections, simultaneously. The parameters `CallID` and `ConnectionIdentifier` are used in MGCP messages to identify to which call and connection the messages belong.

### 7.4.2 VOCAL MGCP implementation

VOCAL has implemented the MGCP stack as a two-level stack. A low-level stack is used to build, parse and encode messages while a high-level stack is used as a callable programming interface or API [Vovida, 2001d]. The code was initially written in C language and later ported to C++ [Dang et al., 2002]. The example call agent provided by VOCAL was written in C with function calls to handle the flow of MGCP messages.

Figure 7.5 diagrams the function calls that relate to the call flows diagramed in Figure 7.4; the endpoints have been omitted to avoid cluttering the diagram.

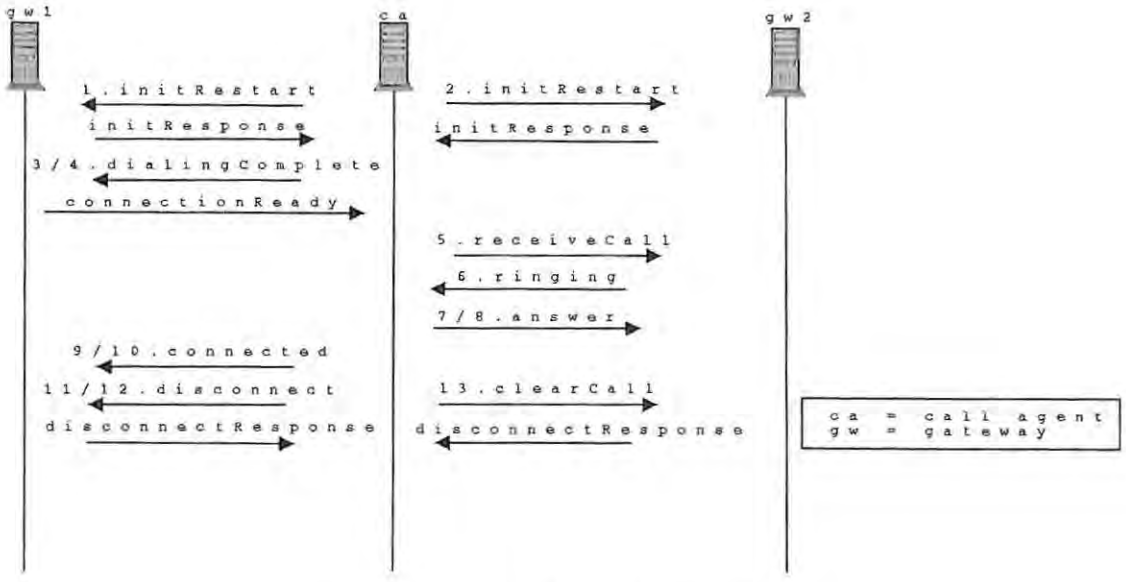


**Figure 7.5 Example MGCP function flow**

As an example, the function `initRestart` sends the `RQNT` message while the function `initResponse` is called when the call agent receives the status message (200 ok) from the gateways. The procedure is similar for the other functions and messages. The functions will be discussed in detail in the implementation of the SIPMGCP translator (see section 7.4.3).

### 7.4.3 SIPMGCP translator general architecture

**Error! Reference source not found.** illustrates the basic architecture that we have used to implement the SIPMGCP internetworking service. The translator in the middle is seen by the MGCP network as a call agent and as a user agent by the SIP network. The translator consists of a call agent and a user agent.

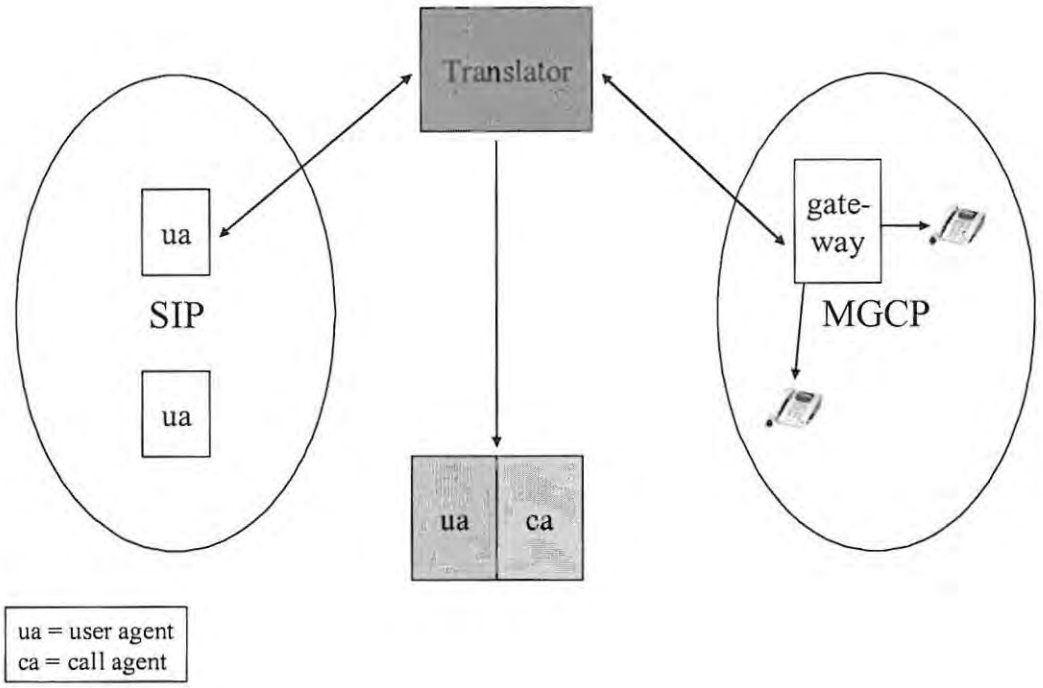


**Figure 7.6 SIPMGCP translator architecture**

A state machine was used to model the translator. This state machine had to model the call from a MGCP perspective and SIP perspective. A decision was made to use the state machine from the MGCP side to model the call for both SIP and MGCP. This state machine was implemented using a data structure written in C. The valid states are `Init`, `Idle`, `NewCall`, `RingBack`, `IncomingCall`, `Ringing`, `Connected`, `Disconnect` and `CallStateMax`.

```
typedef enum CallState
{
    //@{
    /// Init
    Init = 0,
    /// Idle
    Idle,
    /// New Outgoing Call
    NewCall,
```

```
    /// Ring Back
    RingBack,
    /// New Incoming Call
    IncomingCall,
    /// Ringing
    Ringing,
    /// Connected
    Connected,
    /// Half Disconnect
    Disconnect,
    CallStateMax
    //@}
};
```

The state machine, call data and various other C typedef structures were used as in VOCAL. To implement the translator, modifications were required on the MGCP functions, including introducing some SIP functions. The rest of this section will explain how the SIP messages will replace the ones in Figure 7.4 and which functions are used.

### 7.4.4 SIPMGCP translator: a call from SIP to MGCP

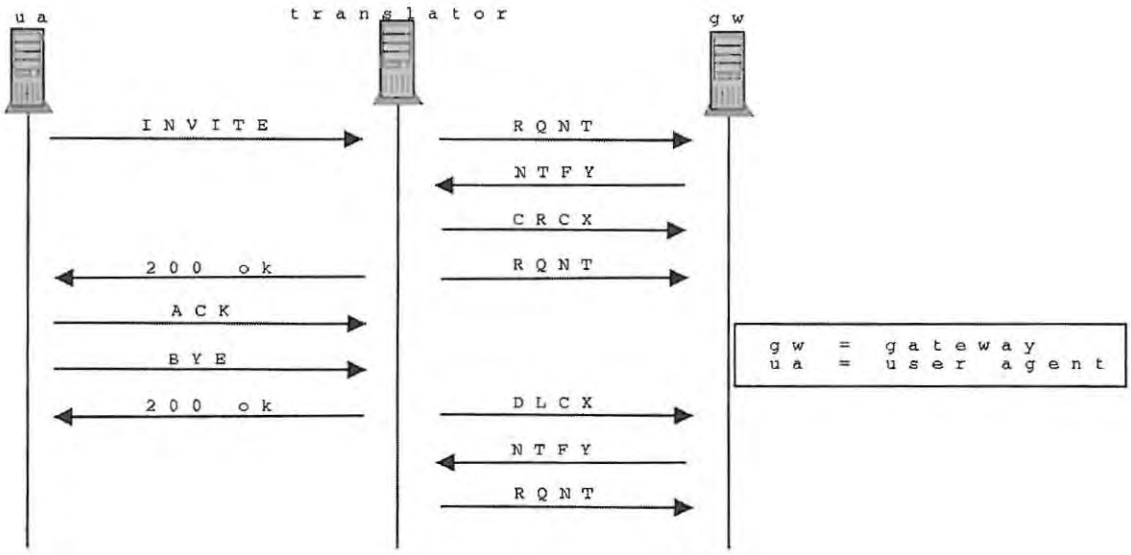


**Figure 7.7 Call from SIP to MGCP**

Figure 7.7 shows the call flow from a SIP user agent to a MGCP endpoint. The user agent initiated the call. The SIP INVITE is the first message that the translator receives from

the user agent. The translator sets up the call for the MGCP endpoint using steps 5-8 in Figure 7.4 (and described in section 7.4.1). The SDP information from the INVITE message is used in the command `CreateConnection (CRCX)` to the MGCP endpoint. Once a call has been set up with the MGCP endpoint the response to the INVITE message is sent using a 200 OK message that includes the SDP media information received from the MGCP endpoint. The user agent responds with an ACK message to complete the session setup. At this stage a call has been set up between the SIP and MGCP endpoint, thus communication in terms of media streams can flow directly between them. Later, the user agent decides to terminate the session by sending a BYE message to the translator. The translator immediately sends the message 200 OK to acknowledge this request. In regard to the termination process, there is no requirement to check if termination has taken place correctly on the MGCP endpoint before sending the 200 OK to the user agent. This prevents delay due to the translator sending the command `DeleteConnection (DLCX)` and necessarily waiting for the response from the gateway.

To implement the call from SIP to MGCP various new functions had to be added to the flow shown in Figure 7.5; these are depicted in Figure 7.8.

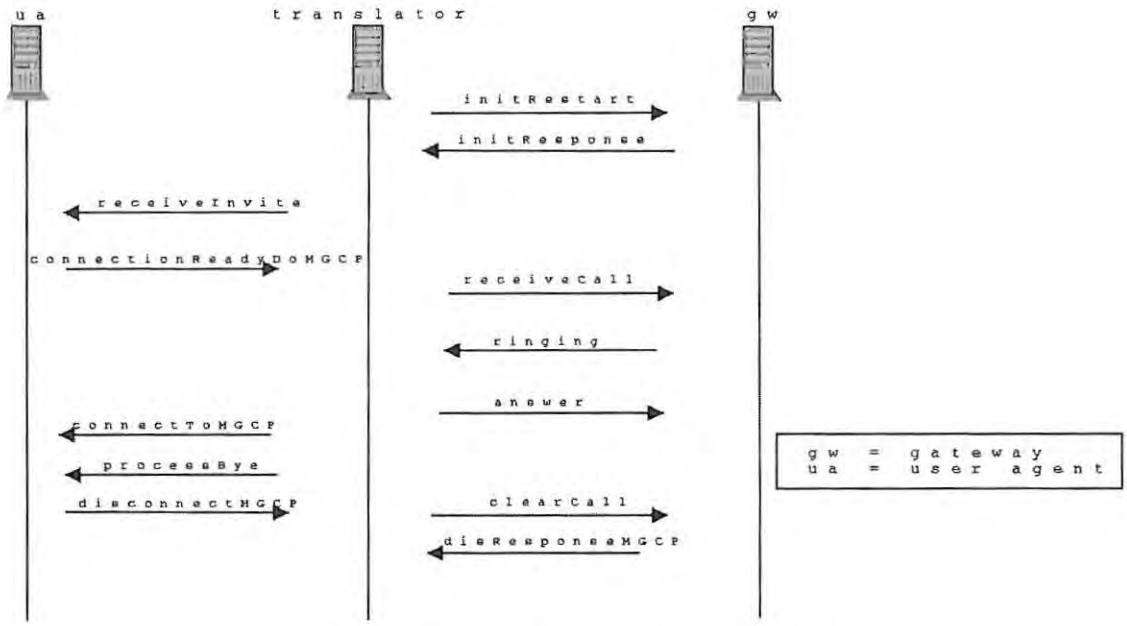


**Figure 7.8 Functions flow from SIP to MGCP**

The first function that is called is `receiveInvite`. The SIP stack in the SIPMGCP translator receives SIP messages using the function `receiveSIPmessage` and processes them using the function `processSIPmessage`. The latter determines what SIP message has been received; if it is an INVITE message the function `receiveInvite` is called. Within `receiveInvite` the `CallData` and the `connectionId` is set up. The state of the endpoint is changed to `NewCall`. The SDP from the INVITE message is also copied, ready to be sent along with the `CRCX` message to the MGCP endpoint, then `connectionReadyDoMGCP` is called.

```
// receiveInvite
// requirements:
// process the invite message and create connection to mgcp
// preconditions:
// valid invite msg and endpoint pointer
// postconditions:
// call the function connectionReadyDoMGCP
void receiveInvite(Sptr<InviteMsg> invite, Endpoint* ep)
{
    cout << "endpoint.cxx receiveInvite " << endl;
    // processInvite will process the invite message check that it has
a valid from sip url and return the sdp
    Sptr<SipSdp> sdp = processInvite(invite);
```

```
    //doing the mgcp part
    CallData* call = newCallData (ep);
    sprintf (call->callIdentifier, "%X", newCallIdentifier());
    int destEpId;
    destEpId = 0;
    ep->call = call;

    stateChange (ep, NewCall);

    //we need to set the connectionId and the session description to be
sent to the gw
    strcpy (call->endpoint[0].connectionId, call->callIdentifier);
    call->endpoint[0].localConnection = sdp->getSdpDescriptor ();

    //debugging
    //cout << "ep->id = " << ep->id << endl;

    connectionReadyDoMGCP (ep);

} /* receiveInvite */
```

The function connectionReadyDoMGCP changes the state of the endpoint to RingBack, sets up the destination endpoint and notifies it of the call event ReceiveCall.

```
// connectionReadyDoMGCP
// requirements:
// change the state to RingBack, setup destintation endpoint and notify
the endpoint
int
connectionReadyDoMGCP (Endpoint* ep)
{
    cout << "endpoint.cxx connectionReadyDoMGCP " << endl;

    stateChange (ep, RingBack);

    //debugging
    //cout << "ep->id = " << ep->id << endl;

    CallData* call1 = ep->call;
    /* Warning: No locking on shared endpoint and call data */

    Endpoint* destEp = &eps[call1->endpoint[0].id];
    destEp->call = call1;

    notifyEndpoint (destEp, ReceiveCall);
    return 1;
}   /* connectionReadyDoMGCP */
```

Call events are processed in the function `processEvent`; if the event is `ReceiveCall` then the function `receiveCall` is called. The `receiveCall` function changes the state to `IncomingCall`, copies the `requestIdentifier` and sends the command `NotificationRequest` or `RequestForNotification` (RQNT message) to the gateway.

```
int
receiveCall (Endpoint* ep)
{
    cout << "endpoint.cxx receiveCall " << endl;

    stateChange (ep, IncomingCall);

    sprintf (ep->requestIdentifier, "%X", newRequestIdentifier());
    MgcNotificationRequest rqnt ("testID", ep->requestIdentifier);
    ep->gw->send (rqnt);
    if (rqnt.getResponseCode() == TransactionExecuted)
    {
        ringing (ep);
    }
    return 1;
}   /* receiveCall */
```

The `ringing` function is a simple function that just changes the state to `Ringing`.

```
int
ringing (Endpoint* ep)
{
    cout << "endpoint.cxx ringing " << endl;

    stateChange (ep, Ringing);
    return 1;
}   /* ringing */
```

After a `Notify` (NTFY) message is received from the gateway, indicating that the endpoint has accepted the call, the function `answer` is called. The function `answer` creates a CRCX message with the necessary SDP information, sends this message to the gateway and waits for a response. If the response is 200 OK (`TransactionExecuted`) then it will send a RQNT message to the gateway. This notification request message is to notify the translator of any events that could occur in that connection. For example, if the endpoint decides to terminate the session, the translator would receive an on-hook (hu) event.

```
int
answer (Endpoint* ep)
{
    cout << "endpoint.cxx answer " << endl;

    cout << "ep->id = " << ep->id << endl;

    CallData* call = ep->call;
    assert (call);
    MgcCreateConnection     crcx     ("testID",     call->callIdentifier,
"sendrecv");
    crcx.setRemoteConnectionDescriptor                                    (call-
>endpoint[0].localConnection);
    ep->gw->send (crcx);

    if (crcx.getResponseCode() == TransactionExecuted)
    {
        strcpy (call->endpoint[0].connectionId,
                (crcx.getConnectionId ()).data());
        call->endpoint[0].localConnection
crcx.getLocalConnectionDescriptor ();
        notifyEndpoint (&eps[call->endpoint[0].id], Connect);

        sprintf (ep->requestIdentifier, "%X", newRequestIdentifier());
        MgcNotificationRequest rqnt ("testID", ep->requestIdentifier);
        ep->gw->send (rqnt);

        if (rqnt.getResponseCode() == TransactionExecuted)
        {
            stateChange (ep, Connected);
        }
    }
    return 1;
}    /* answer */
```

Once a response is received from the gateway for `NotificationRequest` (`RQNT` message) the function `connectToMGCP` is called. The function `connectToMGCP` constructs the 200 OK message to be sent back to the user agent. It also includes the SDP information received from the endpoint and changes the state to `Connected`.

```
int
connectToMGCP (Endpoint* ep)
{
    cout << "endpoint.cxx connected " << endl;

    CallData* call = ep->call;
    assert (call);

    //confirmation the other party has accepted the call
    //send the 200 OK for SIP
```

```
    send200OK(call, msg);

    stateChange (ep, Connected);

    return 1;
}    /* connectToMGCP */
```

At this stage the user agent is connected to the endpoint via the gateway, and media communications can flow between the two. From this point on, either the user agent or the endpoint could resolve to terminate the session. We will demonstrate the termination sequence after showing the call setup from a MGCP endpoint to a SIP user agent.

### 7.4.5 SIPMGCP translator: a call from MGCP to SIP

Figure 7.9 represents a call flow from a MGCP endpoint to a SIP user agent. We have assumed that the endpoint is correctly set up; in other words the initialization sequence has been completed. We also assume that all MGCP requests have 200 OK responses.

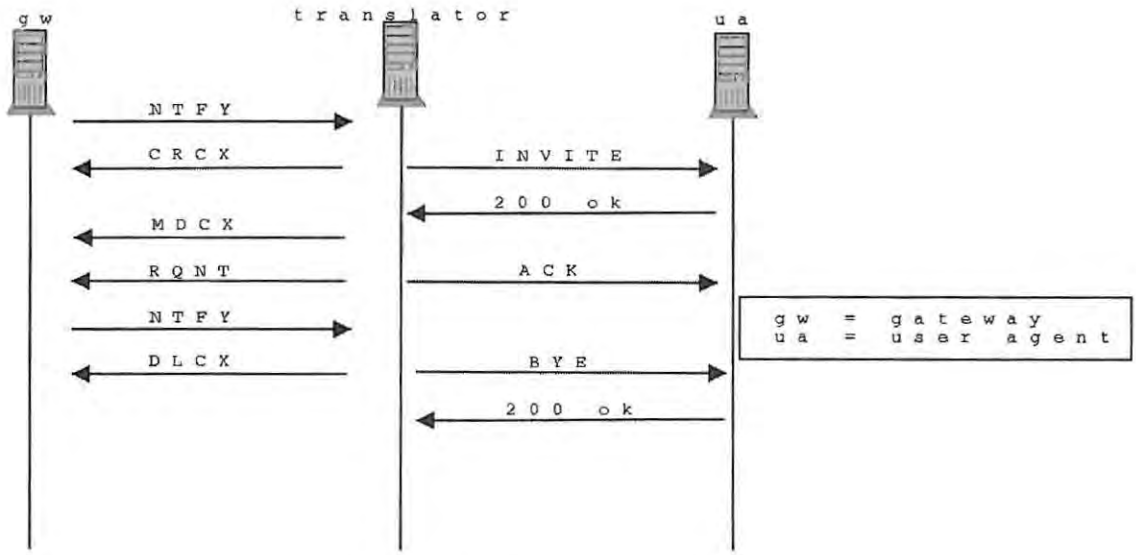


**Figure 7.9 Call from MGCP to SIP**

The first message that the translator receives is NOTIFY (NTFY message), signaling that the gateway has detected an off-hook (hd) event from one of its endpoints. The translator acknowledges this message with a 200 OK message and sends `CreateConnection`

(CRCX message) to set up the connection on the gateway. The gateway responds with a 200 OK containing the SDP with the media streams that it is willing to listen on. The translator sends the INVITE message with the SDP it has acquired from the gateway, on behalf of the endpoint, to the user agent. If the user agent accepts the call it sends a 200 OK, with the SDP that the user agent is willing to listen on, back to the translator. The translator uses the SDP information contained in the 200 OK to create a ModifyConnection (MDCX) message to be sent to the gateway. If the response of the gateway to MDCX is 200 OK, then the translator sends a NotificationRequest (RQNT). This indicates that the connection has been set up and the translator is requesting notification of events such as on-hook (hu) events from the gateway. Once this has been completed, the translator sends the ACK message to the user agent to complete the SIP session setup. Note that the translator does not immediately send the ACK once it receives the 200 OK from the user agent, because the MGCP endpoint could still elect to terminate the call setup resulting in an incorrect call setup.

At this point a call has been set up between the MGCP endpoint and the SIP user agent and media can flow directly between the two. From here on, either entity may opt to terminate the call. In Figure 7.9 we represent the situation in which the MGCP endpoint has decided to terminate the call. A NTFY message is sent to the translator to indicate that the gateway has detected an on-hook (hu) event from the endpoint. The translator can send the message DeleteConnection (DLCX) to the gateway without confirmation from the user agent since the endpoint has already terminated the media. The translator sends the BYE message and waits for the 200 OK from the user agent.

To implement the call flow indicated in Figure 7.9 various changes had to be made to the functions flow first illustrated in Figure 7.5. The result of these changes is shown in Figure 7.10.

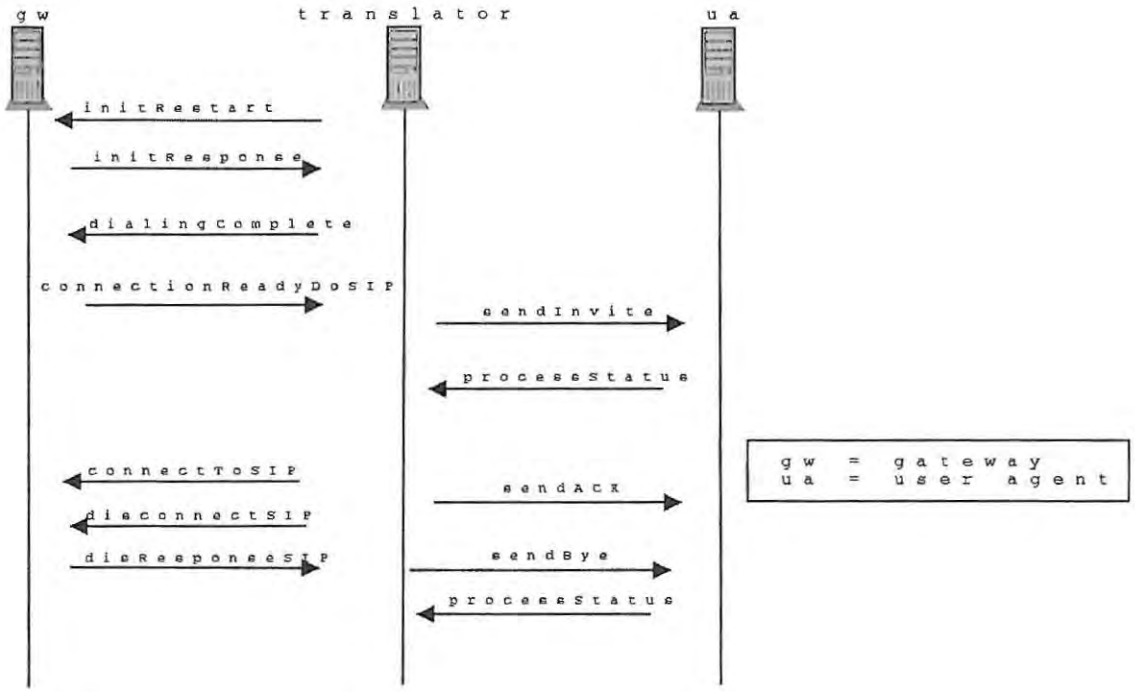


**Figure 7.10 Functions flow from MGCP to SIP**

As in Figure 7.5, `initRestart` and `initResponse` are needed to initialize the endpoints that are attached to the gateways. The function `dialingComplete` is called after the translator receives a `NTFY` message from the gateway. The function sets up the endpoints that need to be called. In this particular case, since there is only one user agent, the destination endpoint identity (`destEpId`) is set to zero. The state changes to `NewCall` and a `CRCX` is sent back to the gateway. If the transaction has been successfully executed then the function `connectionReadyDoSIP` is called.

```
// dialingComplete
// postconditions:
// assume the initial endpoint has been initialized
// requirements:
// setup the destId, change state and send CRCX
// preconditions:
// call connectionReadyDoSIP
int
dialingComplete (Endpoint* ep)
{
    //
    cout << "endpoint.cxx dialingComplete " << endl;

    CallData* call = newCallData (ep);
```

```
    sprintf (call->callIdentifier, "%X", newCallIdentifier());
    int destEpId;
    if (ep->id == 0)
    {
        destEpId = 1;
    }
    call->endpoint[1].id = destEpId;
    ep->call = call;

    stateChange (ep, NewCall);

    MgcCreateConnection     crcx     ("testID",     call->callIdentifier,
"sendrecv");

    ep->gw->send (crcx);

    if (crcx.getResponseCode() == TransactionExecuted)
    {
        strcpy (call->endpoint[0].connectionId,
                (crcx.getConnectionId ()).data());
        call->endpoint[0].localConnection                              =
crcx.getLocalConnectionDescriptor ();
        connectionReadyDoSIP (ep);
    }
    else
    {
        cout << "failed TransactionExecuted\n";
    }
    return 1;
}    /* dialingComplete */
```

`connectionReadyDoSIP` is a simple function that changes the state to `RingBack` and calls the function `sendInvite`. Currently, the call method is very simple and uses the parameters passed to the translator to determine where the user agent is located. An improvement on this method is discussed at the end of this chapter (section 7.5).

```
// connectionReadyDoSIP
// requirements:
// simple method of calling the SIP user agent
// passing it as a parameter to the translator
int
connectionReadyDoSIP (Endpoint* ep)
{
    cout << "endpoint.cxx connectionReadyDoSIP " << endl;

    stateChange (ep, RingBack);

    CallData* call1 = ep->call;
    //sendInvite does:
    //construct the INVITE message
    //add the sdp information
    //send the message
```

```
    sendInvite(uaip, uaportno, localSIPportno, call1);

    return 1;
}   /* connectionReadyDoSIP */
```

`sendInvite` constructs the INVITE message, adds the required SDP information and sends the message. Constructing the INVITE message is a lengthy process involving constructing the various objects that relate to the fields in the SIP message and then getting the `InviteMsg` object to point to them.

```
// sendInvite
// constructs an INVITE message using the given parameters
// sends them asyncronzously
// uaip = ip address of the sip endpoint we are going to call
// uaportno = portno of the sip endpoint we are going to call
// localSIPportno = portno locally
void sendInvite(char* uaip, int uaportno, int localSIPportno, CallData*
call1)
{
    SIPNewCall = 1;
    // construct a SIP TO URL for the message.
    Sptr<SipUrl> toUrl;
    try
    {
      Data toHost = uaip;
      Data toPort = uaportno;
      toUrl = new SipUrl();
      toUrl->setHost(toHost);
      toUrl->setPort(toPort);
    }
    catch (SipUrlParserException e)
    {
      cout << "SipUrlParserException" << endl;
    }

    // construct an invite message using the toUrl, modify other fields
later
    InviteMsg msg( toUrl );

    SipSubject subject;
    subject.set("New call");
    msg.setSubject(subject);

    // construct the fromUrl
    SipFrom from = msg.getFrom();
    // temporary display name
    from.setDisplayName( "Bob Smith" );
    Data fromHost = localhostname;
    Data fromPort = localSIPportno;
    Sptr<SipUrl> fromUrl = new SipUrl();
    fromUrl->setHost(fromHost);
    fromUrl->setPort(fromPort);
```

```
    from.setUrl( fromUrl );
    msg.setFrom(from);

    // construct the via
    SipVia via = msg.getVia();
    via.setPort(localSIPportno);
    msg.removeVia(0);
    msg.setVia(via, 0);

    // construct the Contact list
    SipContact contact = msg.getContact();
    contact.setUrl(fromUrl);
    msg.removeContact(0);
    msg.setContact(contact, 0);

    // sdp will point to the SDP body in the message.  if we change
    // sdp, we change the SDP contents of the body.
    Sptr<SipSdp> sdp;
    sdp.dynamicCast( msg.getContentData(0) );
    // copy the sdp from the gateway
    assert (sdp != 0) ;
    sdp->setSdpDescriptor(call1->endpoint[0].localConnection);

    // now, we will send the INVITE message
    sipStack->sendAsync(msg);
    cout << "sending the INVITE message" << endl;

} /* sendInvite */
```

The translator waits for the response from the user agent. The translator processes status messages in the function `processStatus`. Inside the function we determine if we have received a 200 OK for the INVITE message by checking the command sequence (`getCSeq`) associated with the status message. If the 200 OK is intended for the INVITE then we call the function `connectToSIP`.

```
void processStatus(Sptr<StatusMsg> status)
{
   cout << "processing Status messages" << endl;
   int statusCode;
   statusCode = status->getStatusLine().getStatusCode();
   cout << "got status code: " << statusCode << endl;

   if(statusCode < 200)
   {
     if(statusCode == 180)
     {
       cout << "Ringing SIP side" << endl;
     }
   }

   if(statusCode == 200)
```

```
    {
      SipCSeq testCSeq = status->getCSeq();
      if(testCSeq.getMethod() == "INVITE")
      {
        //received a 200 ok for an invite that we have sent
        cout << "call accepted " << endl;
        //the 200ok contains sdp information
        //copy the sdp from the 200 to the ep, this is to be used for
MDCX
        Endpoint* ep = &eps[0];
        connectToSIP(status, ep);
        SIPNewCall = 0;
        localStatusMsg = status;
      }
      else if(testCSeq.getMethod() == "BYE")
      {
        //received a 200 ok for an bye that we have sent
        cout << "200 OK received for BYE message" << endl;
        cout << "call terminated" << endl;
      }
    }
}
```

connectoToSIP creates the MDCX message to be sent to the gateway where the CRCX message originated. This modifies the connection information stored on the gateway according to the media information from the user agent. With this information available, the gateway identifies the type of media and the port number to use. If the MDCX message is sent successfully, a RQNT message is also sent to the gateway to notify the translator of any event such as on hook (hu). If the message RQNT is sent successfully it will call the sendACK function.

```
int
connectToSIP (Sptr<StatusMsg> status, Endpoint* ep)
{
    cout << "connected to sip user agent" << endl;
    //create the MDCX message
    MgcModifyConnection mdcx ("testID",
                              call->callIdentifier,
                              call->endpoint[0].connectionId,
                              "sendrecv");
    //use a temporary SipSdp to hold the session description
    Sptr<SipSdp> tempSdp;
    tempSdp.dynamicCast(status->getContentData(0));
    mdcx.setRemoteConnectionDescriptor (tempSdp->getSdpDescriptor());
    ep->gw->send (mdcx);
    if (mdcx.getResponseCode() == TransactionExecuted)
    {   //if the transaction has executed send the RQNT to listen for
any events
        sprintf (ep->requestIdentifier, "%X", newRequestIdentifier());
```

```
        MgcNotificationRequest rqnt ("testID", ep->requestIdentifier);
        ep->gw->send (rqnt);
        if (rqnt.getResponseCode() == TransactionExecuted)
        {
            stateChange (ep, Connected);
            sendACK(status);
        }
    }
    return 1;
}   /* connectToSIP */
```

The sendACK function sends the ACK message back to the user agent to complete the session setup for SIP.

```
void sendACK(Sptr<StatusMsg> status)
{
    cout << "sending ACK " << endl;
    AckMsg ack(*status);
    sipStack->sendAsync(ack);
    cout << "connection established from MGCP to SIP" << endl;
} /* sendACK */
```

Now we have reached a common point for both call flows (a call from SIP to MGCP and from MGCP to SIP). Both calls have been successfully initiated and established. We are now ready to move onto the next phase, the termination of the calls. This phase also involves two parts, termination from the SIP side and termination from the MGCP side.

### 7.4.6 SIPMGCP translator: termination of calls

**Termination from SIP**

When the translator receives a BYE message from the user agent, it immediately sends a 200 OK back to the user agent and calls the function disconnectMGCP.

```
void processBye(Sptr<ByeMsg> bye)
{
    cout << "processing BYE" << endl;
    send200BYE(bye);
    disconnectMGCP(&eps[0]);
}
```

The function disconnectMGCP changes the state to Disconnect and notifies the endpoint of the call event ClearCall.

```
int disconnectMGCP(Endpoint* ep)
{
    stateChange (ep, Disconnect);
    CallData* call = ep->call;
    assert (call);
    notifyEndpoint (&eps[call->endpoint[0].id], ClearCall);
    return 1;
} /* disconnectMGCP */
```

The function processEvent processes call events; if it detects a ClearCall event then it calls the function clearCall. clearCall creates and sends the DLCX message.

```
int
clearCall (Endpoint* ep)
{
    cout << "endpoint.cxx clearCall " << endl;
    stateChange (ep, Disconnect);
    CallData* call = ep->call;
    assert (call);
    MgcDeleteConnectionAgent dlcx ("testID");
    dlcx.setCallId (call->callIdentifier);
    dlcx.setConnectionId (call->endpoint[0].connectionId);
    ep->gw->send (dlcx);
    if (dlcx.getResponseCode() == ConnectionDeleted ||
          dlcx.getResponseCode() == TransactionExecuted)
    {
      //clearing the call
       callDataInit (ep->call);
       ep->call = NULL;
      //clearCall will not call disResponseMGCP it will only send the
dlcx and wait for the response
      //in processEvent, once it receives the hangup event from ep[1]
it'll call disResponseMGCP
    }
    return 1;
}    /* clearCall */
```

Once the translator receives the notification from the gateway that the endpoint has replaced the hook, the on-hook (hu) event, it calls the disResponseMGCP function that reinitializes the state and endpoint.

```
// disResponseMGCP
// gets called when the translator receives notification of line event
"hu"
// from the gateway
int
disResponseMGCP (Endpoint* ep)
{
    cout << "endpoint.cxx disResponseMGCP " << endl;
    stateChange (ep, Init);
    initRestart (ep);
    return 1;
} /* disResponseMGCP */
```

**Termination from MGCP**

The translator receives a NTFY message from the gateway notifying it of the on-hook event at the endpoint. The function disconnectSIP is called. disconnectSIP changes the state to Disconnect and creates the DLCX message. This message is sent to the gateway to terminate the connection. If the message is sent successfully, then the call data is reinitialized, followed by a call to disResponseSIP.

```
int
disconnectSIP (Endpoint* ep)
{
    cout << "endpoint.cxx disconnect " << endl;
    stateChange (ep, Disconnect);
    CallData* call = ep->call;
    assert (call);
    MgcDeleteConnectionAgent dlcx ("testID");
    dlcx.setCallId (call->callIdentifier);
    dlcx.setConnectionId (call->endpoint[0].connectionId);
    ep->gw->send (dlcx);
    if (dlcx.getResponseCode() == ConnectionDeleted ||
          dlcx.getResponseCode() == TransactionExecuted)
    {
      //clearing the call
        callDataInit (ep->call);
        ep->call = NULL;
        disResponseSIP (ep);
    }
    return 1;
}   /* disconnectSIP */
```

disResponseSIP reinitializes the state and the endpoint followed by a call to sendBye to terminate the session on the SIP side.

```
int
```

```
disResponseSIP (Endpoint* ep)
{
    cout << "endpoint.cxx disconnectResponse " << endl;
    stateChange (ep, Init);
    initRestart (ep);
    sendBye();
    return 1;
}    /* disResponseSIP */
```

BYE messages in the VOCAL SIP stack can be constructed from status or ACK messages depending on whether the SIP user agent was acting as the User Agent Client (UAC) or the User Agent Server (UAS). If the translator initiated the call with the user agent then the translator would have been acting as an UAC, hence it would use the status message in order to construct the BYE message. If the call was initiated by the user agent then the translator was acting as the UAS and it would use the ACK message to construct the BYE message. The function `sendBye` handles both cases.

```
//have to construct bye from status or ack, depending on whether you
are the UAC or UAS
void sendBye()
{
    if(localStatusMsg != 0)//we are the uac we did received a 200ok
    {
      cout << "sending BYE from uac" << endl;
      ByeMsg byeStatus(*localStatusMsg);
      sipStack->sendAsync(byeStatus);
      localStatusMsg = 0;
    }

    if(localAckMsg != 0)//we are the uas we did received a ack
    {
      cout << "sending BYE from uas" << endl;
      ByeMsg byeAck(*localAckMsg);
      sipStack->sendAsync(byeAck);
      localAckMsg = 0;
    }
}
```

Figure 7.11 is a screenshot of the SIPMGCP translator in action.

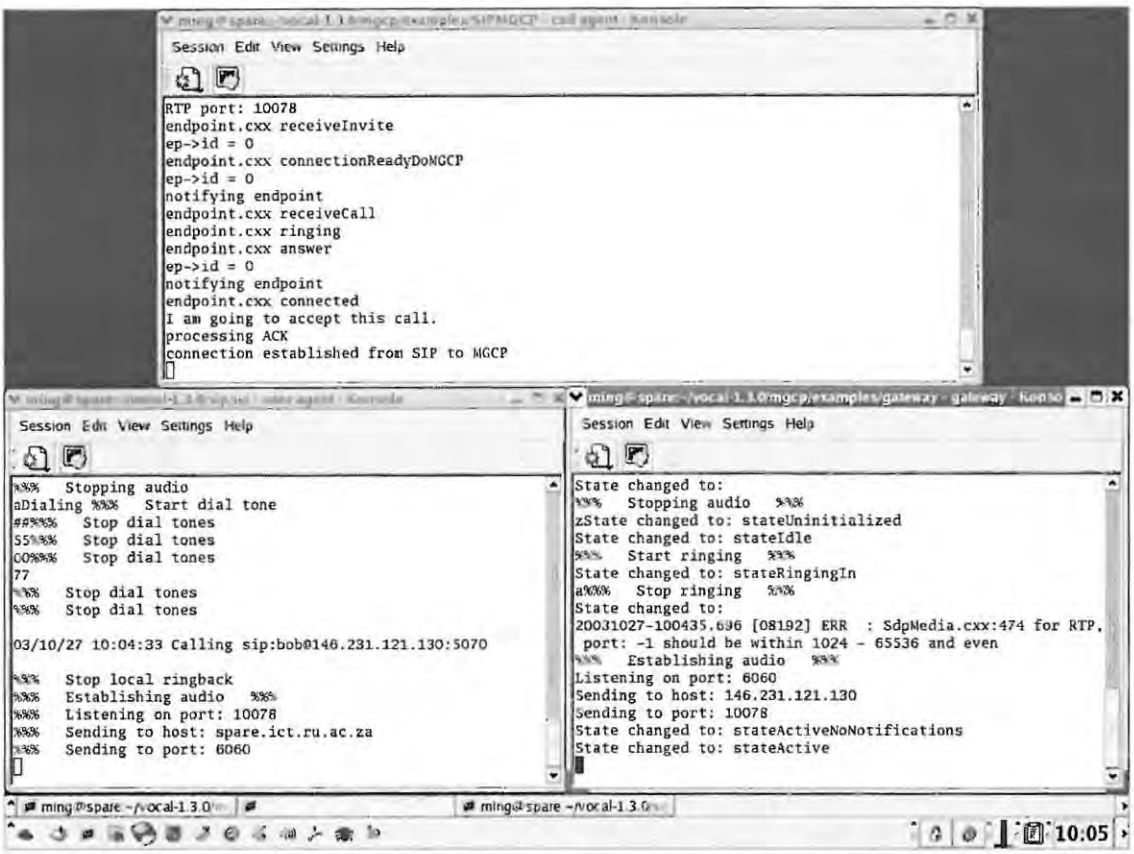


**Figure 7.11 Screenshot of SIPMGCP translator**

The top window depicts the SIPMGCP translator, showing various function calls including debugging messages. The bottom left window depicts the user agent from VOCAL with no changes made to it. The bottom right window displays the example MGCP gateway from VOCAL, also with no changes made to it.

## 7.5 Internetworking services discussion

In general, accessing another network even for a basic service such as call delivery appears to be not an easy task.

VOCAL is an open-source software suite. Users from all over the world contribute to this software. Study of the translator, or `SIPH323CSGW` call signaling gateway, has become a popular project branching from the main VOCAL software suite. Programmers and

companies who once worked with H.323 software have contributed to the project in an effort to ensure interoperability between the H.323 network and the SIP network. The amount of on-going work aiming to produce an internetworking service of industrial strength is substantial. Companies that use H.323 software, possibly with plans for a transition to SIP, might benefit from participating in this type of development. `SIPH323CSGW` uses the latest OpenH323 libraries for the H.323 side of the gateway, thus ensuring interoperability with H.323 networks.

As mentioned, due to the commercialization of CINEMA, which occurred midway through the study, little experimentation was possible with `SIP323`, the SIP to H.323 call signaling gateway. CINEMA has since improved `SIP323` and its testing may resume once licenses for the software are obtained.

Various design issues were contended with during the implementation of the SIPMGCP translator. Although we could have implemented the translator by plugging a complete user agent into it, this would have meant that the translator had to be in control of two state machines (namely, the SIP user agent and the MGCP call agent) needing synchronization. The current design is apt because it uses just one state machine to control both the SIP and MGCP sides.

The example call agent was written in C language with functions handling the actual MGCP messages. The example gateway, on the other hand, was written in C++ with MGCP messages being handled by methods in a class. The VOCAL MGCP API was originally written in C and later ported to C++. A logical progression towards improving the translator would be to first port the call agent code to C++ before incorporating the SIP portion of the service. This would entail restructuring the code of the call agent into logical objects with classes and methods. Due to time constraints, this approach was sidestepped. However, an investigation of that approach could extend to future work. Porting the code to C++ has many benefits, one of which is extensibility: C++ would allow for the translator to be extended easily.

Thus far, the SIPMGCP translator is a proof-of-concept implementation; improvements are possible. For example, the function `connectionReadyDoSIP` uses a simple method to call the user agent. One conceivable improvement would be to implement a database by mapping numbers to the SIP URLs of the user agents. `connectionReadyDoSIP` would call a function to listen for DTMF tones from the gateway and look up the resulting SIP URL from the number that the user has dialed. (DTMF tones are handled as line events in MGCP, thus a session need not be in place in order to receive DTMF tones.)

MGCP provides a fine-grain control over the media resources and was designed specifically to control media gateways [Radvision Corporation, 2002]. We may have chosen a signaling protocol, like SIP, to build our own media gateway, allowing SIP users direct access to the gateway. However, a custom-made media gateway, using a signaling protocol to control the gateway, would likely work for just one type of media conversion. Then, as soon as a new network arrives, requiring a new media gateway to do media conversion, a new custom-made media gateway would have to be assembled (requiring time and effort on the part of the programmer). MGCP provides a standard interface for accessing media gateways; with MGCP-compliant gateways readily available in the market, when a new network arrives, all that is required is the purchase of a MGCP-compliant gateway (which can do the required conversion of media and add it to the current network).

Once internetworking services are readily in place one may start investigating the possibility of accessing advanced services on the other networks.

An ISDN gateway was available as a service on the H.323 network. This was set up by fellow colleague Jason Penton [Penton et al., 2001a]. We used the internetworking service from VOCAL, the `SIPH323CSGW` call signaling gateway, to successfully access the ISDN gateway on the H.323 network. This allowed SIP users PSTN connectivity and exists as an example of how an internetworking service can be used to access a service on another network.

Collaborative work with Ashley Jacobs produced an MGCP-compliant SMS gateway [Jacobs and Clayton, 2003]. This is a service in MGCP. The SIPMGCP translator developed for internetworking (see section 7.4.3) could feasibly be used in conjunction with the SMS gateway to allow SIP users to send SMSs.

Future work in the area of accessing services on another network could focus on improving the translator to better support (many) other services. This approach would mean updating the translator each time a new service arrives. Another approach could be to constrain the services so that they can be accessed via the translator. Constraining the services would mean reducing the number of messages that each service uses. The major hindrance to a translator involves the number of messages that it must process from either side. Consequently, reducing the number of messages that the service uses could allow quicker services.

## 7.6 Summary

This chapter introduced internetworking services. We examined how communication between the SIP and H.323 networks was made possible in CINEMA and VOCAL. Once this internetworking service was in place, we went on to investigate whether or not services on the other network could be accessed using the internetworking service.

Next, a SIPMGCP translator was developed in order to investigate internetworking between the SIP and MGCP networks.

Finally, a discussion of internetworking issues was provided.

# Chapter 8 Conclusions and Extensions

## 8.1 Summary

This thesis began by providing a basic justification for an investigation into service creation, in the context of multimedia real-time communication. The extent of Internet expansion in the next stages will depend on whether it can establish itself as a real-time multimedia-access network.

We cited the two dominant signaling protocols, SIP and H.323, available for real-time multimedia sessions over IP. SIP was chosen as the protocol employed to investigate services for this study. Reasons for the choice and a basic introduction to the SIP protocol were given in Chapters 1 and 2.

Among all possible implementations of SIP environments, CINEMA and VOCAL were chosen (Chapter 3). These architectures were selected not only because the source code was available, and therefore modifications were possible, but also because the author supports an open-source philosophy whereby an individual is allowed to craft modifications to the software, with these modifications becoming widely available to the public. The CINEMA software suite was provided on an academic basis, while the VOCAL software suite is open source.

Once the environments were deployed, the investigation into service provisioning began. We proposed an initial division of services into *basic* and *advanced*. Basic services are ones that provide the minimal functionality required by any telephony environment and are assumed to be available at all times. Advanced services need basic services in order to operate, thus these services have been the focus of the research. We further divided advanced services into four groups: *call-related*, *interactive*, *internetworking* and *hybrid* (Chapter 4), with each containing a *composite* subgroup for services that are a concatenation of services in the larger group.

Next, we discussed examples of service creation for three of the advanced services categories: *call-related*, *interactive* and *internetworking*.

In the category *call-related* we investigated the tools provided to implement services in each environment (Chapter 5). SIP-CGI is used in CINEMA, while CPL is used in VOCAL. Various services were implemented to exercise the capabilities of each environment. Call-blocking, username lookup and missed call services were developed for CINEMA using SIP-CGI. These services were developed with relative ease since their creation required only basic knowledge of scripting languages. The service call blocking used an available standard script, while the services username lookup and missed call showed how well SIP-CGI can be used with backend systems and other Internet protocols (namely the MySql backend database system and the SOAP protocol, respectively). Similar call-related services were developed for VOCAL using CPL. A discussion was provided to contrast the similarities and differences for each of the call-related service-creation mechanisms.

No standard tools are available for implementing interactive services; yet, we began with an investigation into the implementations of the voicemail service in both environments (Chapter 6). The voicemail service was used as a template to develop, in both CINEMA and VOCAL, another interactive service, namely a reminder or notification service. Services in the *interactive* category are more difficult to implement than call-related ones. A discussion outlining the difficulties encountered in implementing interactive services followed.

Work on the *internetworking* category of services focused initially on internetworking between H.323 and SIP; later we developed a service able to internetwork between MGCP and SIP (Chapter 7). Various tests were carried out to probe the extent of successful internetworking between H.323 and SIP in VOCAL. Less experimentation was done in CINEMA, due to the sudden commercialization of the software used for this environment during the course of the study. The results of the tests done in VOCAL show

that the services provisioned for the SIP endpoints may be executed when a call is made from the H.323 endpoint, but understandably a complete H.323 network including a properly configured H.323 gatekeeper is needed for the same functionality to occur on the H.323 side as well. Next we successfully developed the service for internetworking between MGCP and SIP, also in VOCAL.

The development of services in the *internetworking* category was relatively more difficult in comparison to services developed in the categories *call-related* and *interactive*, because the developer required knowledge of two protocols. Nonetheless, we discussed the development of internetworking service for SIP and MGCP. This was followed by a discussion concerning accessing services on the other network using the newly formulated internetworking service.

## 8.2 Conclusions

The aim of the research was to answer certain broad questions related to service creation in CINEMA and VOCAL, namely:

1. "Where in the architecture can services be deployed?"
2. "What level of expertise is required to create services?"
3. "How easy is it to create a service?"

We summarize the answers to these questions in the subsections below, organizing them according to the services categories introduced in Chapter 4.

### 8.2.1 Call-related services

With respect to the question "Where in the architecture can call-related services be deployed?", we found that scripts in CINEMA can be located at the server or at the client, while in VOCAL all scripts are stored on the server. In CINEMA, it is up to the user to make sure that his or her script, wherever it is located, is executed; in VOCAL the user need not worry about the execution of the service because the scripts are centralized. Due

to centralization, the VOCAL administrator can easily manage scripts, whereas the CINEMA administrator has control only over the scripts that are located on the server.

We proposed a solution to the management of scripts in CINEMA, introducing a central repository to store scripts, from which a user could download them using a user agent. The advantage of this solution is that a generic user agent can be used to access services on the server. The greatest disadvantage is that users will be left with less flexible services; another disadvantage is that the user becomes dependent on the availability of the server.

With respect to the question "What level of expertise is required to create call-related services in this environment?", we found that users with knowledge of a scripting language can write services in CINEMA, while in VOCAL programmers with sufficient knowledge of CPL are required to create new services. CPL can be used to describe services in Internet telephony but CPL does not completely describe all possible services. Whenever a service that cannot be described using the available set of constructs in CPL is needed, extensions to CPL need to be made. This requires the programming skills of an experienced programmer, who will make extensions to the CPL language, which require modifications to the interpreter, using commonly available tools. CINEMA, compared to VOCAL, was limited by the lack of SIP-CGI support for outgoing calls. If the nature of the service requires support for outgoing calls, then the programmer will have to contend with the rather large job of extending the SIP-CGI framework.

With respect to the question "How easy is it to create call-related services in this environment?", we found that services in CINEMA could be easily created using SIP-CGI, and similarly in VOCAL using CPL. The scripting knowledge required to create services using SIP-CGI can be more or less extensive depending on the service being created. For example, the services call forwarding and call blocking were easier to create compared to the services user name lookup and SMS missed call. Configuring services in VOCAL is fairly easy using the GUI interface, while configuring services in CINEMA requires editing the script for the service.

### 8.2.2 Interactive services

With respect to the question "Where in the architecture can interactive services be deployed?", we found that for both environments, CINEMA and VOCAL, the interactive services were best deployed onto interactive servers, which are separate from the core servers. Logic suggests that the location of interactive servers is separate from the core servers to provide modularity and ease of deployment and management. This is consistent with current approaches in software development, whereby complex modules are kept separate from the core. The actual location of the service can be centralized or at the edge. If the service is centralized (i.e., the interactive and the core servers together can be seen logically as a central server), then it can be managed more easily by the administrator. This type of deployment is exemplified by the current trend in telephony whereby a user may utilize a centralized voicemail service. If the service is located at the edge, then the service is vulnerable to the uptime of the user agent, which is in general uncertain. Considering the voicemail service again, its location at the edge could result in an inability to deliver the service and thus result in lost messages.

With respect to the question "What level of expertise is required to create interactive services in this environment?", we found that experienced programmers are required to create interactive services in both environments. Both environments also required in-depth knowledge of the protocol, their system architecture, as well as the APIs for each SIP stack.

With respect to the question "How easy is it to create interactive services in this environment?", we found that in both environments, CINEMA and VOCAL, there is no standard way of implementing interactive services, which makes their implementation difficult. The standard ways of creating services using SIP-CGI and CPL in call-related services cannot be applied to interactive services because interactive services require the service to directly interact with the user at the time of execution; neither SIP-CGI nor CPL support this, at least not directly. Naturally, once a standard framework is in place

for this category of services, this difficulty will be reduced. For example, one could develop a language to describe interactive services. This language should offer flexibility and ease of use, as SIP-CGI and CPL do, and it would support an interactive interface with the user at time of execution.

### 8.2.3 Internetworking services

With respect to the question "Where in the architecture can internetworking services be deployed?", it is most natural, for both CINEMA and VOCAL, that signaling gateways (translators) are located at the edge.

More specifically, in CINEMA, the `sip323` gateway can be deployed with or without H.323 gatekeeping or SIP proxying capabilities. In VOCAL, the `siph323csgw` gateway, acts as a call-routed gatekeeper on the H.323 side, and as a SIP user agent on the SIP side. The `SIPMGCP` translator was built as an internetworking service for this category of services and was modeled on the architecture of the `siph323csgw` gateway in VOCAL. The `SIPMGCP` translator was successfully seen on the MGCP side as a call agent and on the SIP side as a SIP user agent.

MGCP is a master-slave protocol with entities such as gateways, call agents and endpoints. The call agent acts as the master, where most of the intelligence is contained, and the gateway acts as the slave that executes the commands provided by the master. The endpoints in MGCP are lines or simple devices connected to the gateways, such as PSTN lines or POTS phones. This is obviously different from H.323 and SIP, where endpoints have PC-capable computational abilities. Thus, services in MGCP are limited and the major reason for internetworking with MGCP is to allow SIP endpoints the ability to access the gateways in MGCP.

The approach to internetworking used in this study is at a protocol level. Other approaches were considered, such as internetworking at a service level, whereby a gateway is developed for each service. This approach has the advantage of being more

secure because users from other networks are only allowed to enter the network to access a specific service, but it has the disadvantage of not being able to scale well, as more gateways must be developed and deployed once more services are exposed to other networks.

With respect to the question "What level of expertise is required to create internetworking services in this environment?", experienced programmers, for both CINEMA and VOCAL, are required. Also the programmers must be knowledgeable of the two protocols to be bridged in order to create internetworking services.

With respect to the question "How easy is it to create internetworking services in this environment?", we could not provide an answer in regard to CINEMA, since no MGCP stack or API was available. However, an MGCP stack and API were available in VOCAL and the `SIPMGCP` translator was developed. Developing this service required a substantial amount of analysis and design before implementation could take place.

Table 8.1 summarizes the questions and answers discussed in the conclusion.

**Table 8.1 Table of conclusions**

| **Question** | **Environment** | **Type of service** | | |
|---|---|---|---|---|
| | | *Call-related* | *Interactive* | *Internetworking* |
| Where in the architecture can services be deployed? | CINEMA | Server or Client | Server | Signaling gateway |
| | VOCAL | Server only | Server | Signaling gateway |
| What level of expertise is required to create | CINEMA | User | Experienced programmer | Not applicable |

| | | | | |
|---|---|---|---|---|
| services? | VOCAL | Beginner programmer | Experienced programmer | Experienced programmer |
| How easy is it to create a service? | CINEMA | Easy | Moderate | Not applicable |
| | VOCAL | Relatively easy | Moderate | Difficult |

## 8.3 Extensions

Our investigation into the internetworking services for CINEMA was limited due to its commercialization, which occurred during the course of this study. In the future, we would like to investigate whether or not the experiments performed for the internetworking services in VOCAL will yield the same results in CINEMA. This will simply require the CINEMA software license.

We noted that outgoing SIP-CGI support is not available in CINEMA. It would be interesting to investigate the level of difficulty in implementing this feature. New services could result once outgoing SIP-CGI is enabled.

An important extension to the work reported here will be a systematic investigation of how services in each category might interact with other services in the same or a different category. For example, imagine the following scenario: Alice has an outgoing call-blocking service enabled, which does not allow her to make long-distance calls, while Bob has decided to forward his calls to a distant number (he has the call-forwarding service enabled). What would happen when Alice calls Bob? Would Alice's call be blocked or would it be forwarded to the distant number? The traditional telephony solution is that the call would be forwarded to Bob, while Alice would be charged the local call rate while the long-distance rate would be charged to Bob. It should be noted that it is not just a question of apportioning the cost of the call. For example, Alice might belong to a country that allows calls only within its borders and this country wants to

strictly enforce its policy of no calls to foreign countries. This is just one example of how the interaction of services is potentially problematic and illustrates that the outcome is not always obvious. A possible general solution would be an intelligent policy server that could solve conflicts arising from the sequential execution of services.

A fourth service category, *hybrid* services introduced in Chapter 4, was not investigated. Hybrid services are services that contain elements from more than one of the other categories. This service category could be investigated in future research.

A subgroup of each of the service categories was designated *composite* services. These comprise services that are the result of the concatenation of services in one of the greater categories. An investigation of such composite services could also extend the work accomplished thus far.

Finally, we investigated services that can be created using CPL. Since CPL is protocol- and platform-independent and can be used by both SIP and H.323, it would be interesting to investigate whether a single CPL script can be used to deliver a particular service, irrespective of the network hosting it. For example, if one CPL script is used to block calls from within the SIP network, can the same CPL script be used to block calls from within the H.323 network? If the answer is yes, this suggests the necessary creation of a centralized CPL server to serve both networks. The centralized CPL server would act as an agent for both networks. For example, when a CPL script needed to be executed, a request would be sent to the CPL server, which will fetch and execute the CPL script. This request would be contained within a H.323 or a SIP message, depending on the network from which the request originates.

Appendix A

## Registrar Log File

This is the log file for the Registrar server in the Registration sequence diagram.

ReceiveUDP(): received UDP packet (length 406) from 146.231.123.15:5060:

REGISTER sip:csmsc01ict.cs.ru.ac.za SIP/2.0
Via: SIP/2.0/UDP 146.231.123.15:5060
CSeq: 1 REGISTER
Expires: 3600
Contact: sip:ming@146.231.123.15:5060;q=0.1;action=proxy
From: sip:ming@csmsc01ict.cs.ru.ac.za
Authorization: Basic bWluZ0Bjc21zYzAxaWN0LmNzLnJ1LmFjLnphOg==
Date: Mon, 28 Oct 2002 10:13:27 GMT
Call-ID: 770215716@146.231.123.15
To: sip:ming@csmsc01ict.cs.ru.ac.za
Content-Length: 0

The registrar received this message from the user.

ResponseSendSocket(): Sending 383 bytes to socket 300 via UDP

SIP/2.0 401 Must authenticate with username ming@cssipict.cs.ru.ac.za
Via: SIP/2.0/UDP 146.231.123.15:5060
From: sip:ming@csmsc01ict.cs.ru.ac.za
To: sip:ming@csmsc01ict.cs.ru.ac.za
Call-ID: 770215716@146.231.123.15
CSeq: 1 REGISTER
Date: Mon, 28 Oct 2002 10:15:07 GMT
Server: Columbia-SIP-Server/1.0
Content-Length: 0

WWW-Authenticate: Basic realm="cssipict.cs.ru.ac.za"

The registrar sends the response back to the user, requesting the correct username and password. A challenge is made to the user agent using the WWW-Authenticate header field with the correct realm that the user must authenticate with.

ReceiveUDP(): received UDP packet (length 410) from 146.231.123.15:5060:

REGISTER sip:csmsc01ict.cs.ru.ac.za SIP/2.0
Expires: 3600
Authorization: Basic bWluZ0Bjc3NpcGljdC5jcy5ydS5hYy56YTptaW5nAA==
To: sip:ming@csmsc01ict.cs.ru.ac.za
Call-ID: 770215717@146.231.123.15
Via: SIP/2.0/UDP 146.231.123.15:5060
From: sip:ming@csmsc01ict.cs.ru.ac.za
Contact: sip:ming@146.231.123.15:5060;q=0.1;action=proxy
CSeq: 1 REGISTER
Date: Mon, 28 Oct 2002 10:13:38 GMT
Content-Length: 0

This is the second REGISTER message that the registrar receives from the user. Now, the user has entered the correct username and password. This is encoded in the Authorization header field.

ResponseSendSocket(): Sending 525 bytes to socket 300 via UDP

SIP/2.0 200 OK
Via: SIP/2.0/UDP 146.231.123.15:5060
From: sip:ming@csmsc01ict.cs.ru.ac.za
To: sip:ming@csmsc01ict.cs.ru.ac.za
Call-ID: 770215717@146.231.123.15
CSeq: 1 REGISTER
Date: Mon, 28 Oct 2002 10:15:18 GMT

Server: Columbia-SIP-Server/1.0
Content-Length: 0
Contact: <sip:ming@csmsc01ict.cs.ru.ac.za>; expires="Mon, 28 Oct 2002 11:26:37 GMT"; action=proxy; q=1.00
Contact: <sip:ming@146.231.123.15:5060>; expires="Mon, 28 Oct 2002 11:15:18 GMT"; action=proxy; q=0.10
Expires: Mon, 28 Oct 2002 11:15:18 GMT

The registrar server sends a "200 OK" message back to the user to indicate that the registration has successfully taken place, along with a list of the current registrations that the user has made.

This log file shows that the SIP messages exchanged between the user and the server are the same. This simplifies the task of debugging the system since all one must do is check that the messages are actually being sent or received from both sides.

Appendix B

## CINEMA Proxy Sequence

This section, showing the interactions between user agents and proxy servers, relates to Figure 3.5. The messages received and transmitted from the proxy server are shown here. The messages shown here from the user agents are ones following "200 OK" from the server. The caller Ming is located at sip:ming@csmsc01ict.cs.ru.ac.za, the proxy server is located at cssipict.cs.ru.ac.za, and the callee g9610645 is located at sip:g9610645@edo.dsl.ru.ac.za.

## Proxy server log file:

ReceiveUDP(): received UDP packet (length 515) from 146.231.123.15:5060:
INVITE sip:g9610645@cssipict.cs.ru.ac.za SIP/2.0
Via: SIP/2.0/UDP 146.231.123.15:5060
CSeq: 1 INVITE
Contact: sip:ming@146.231.123.15:5060
Expires: 3600
Subject: test
From: sip:ming@csmsc01ict.cs.ru.ac.za
Date: Fri, 01 Nov 2002 10:41:28 GMT
Call-ID: 837041375@146.231.123.15
Content-Type: application/sdp
Priority: normal
To: sip:g9610645@cssipict.cs.ru.ac.za
Content-Length: 124

v=0
o=ming 464079749521 1036147288 IN IP4 146.231.123.15
s=test
c=IN IP4 146.231.123.15
t=0 0
m=audio 10000 RTP/AVP 0

This is the message that is received by the proxy server from the caller.

ResponseSendSocket(): Sending 274 bytes to socket 300 via UDP

SIP/2.0 100 Trying

Via: SIP/2.0/UDP 146.231.123.15:5060

From: sip:ming@csmsc01ict.cs.ru.ac.za

To: sip:g9610645@cssipict.cs.ru.ac.za

Call-ID: 837041375@146.231.123.15

CSeq: 1 INVITE

Date: Fri, 01 Nov 2002 10:43:22 GMT

Server: Columbia-SIP-Server/1.0

Content-Length: 0

This message is being sent back to the caller to show that the server is busy processing the request.

SendRequest(): Proxying request to socket 372:

INVITE sip:g9610645@146.231.112.107 SIP/2.0

Via: SIP/2.0/UDP cssipict.cs.ru.ac.za:5060; branch=3440148629-0

Via: SIP/2.0/UDP 146.231.123.15:5060

CSeq: 1 INVITE

Contact: sip:ming@146.231.123.15:5060

Expires: 3600

Subject: test

From: sip:ming@csmsc01ict.cs.ru.ac.za

Date: Fri, 01 Nov 2002 10:41:28 GMT

Call-ID: 837041375@146.231.123.15

Content-Type: application/sdp

Priority: normal

To: sip:g9610645@cssipict.cs.ru.ac.za

Content-Length: 124

v=0
o=ming 464079749521 1036147288 IN IP4 146.231.123.15
s=test
c=IN IP4 146.231.123.15

t=0 0
m=audio 10000 RTP/AVP 0

This INVITE message differs from the first INVITE message. Rather, this is the message proxied to the callee.

ResponseSendSocket(): Sending 298 bytes to socket 300 via UDP
SIP/2.0 180 Ringing
CSeq: 1 INVITE
Contact: sip:g9610645@146.231.112.107:5060
Subject: test
Via: SIP/2.0/UDP 146.231.123.15:5060
From: sip:ming@csmsc01ict.cs.ru.ac.za
Call-ID: 837041375@146.231.123.15
To: sip:g9610645@cssipict.cs.ru.ac.za; tag=889683254943.146.231.112.107
Content-Length: 0

The server receives and sends the "180" Ringing message from user g9610645 to user Ming. The two messages are essentially the same.

ReceiveUDP(): received UDP packet (length 520) from 146.231.112.107:5060:
SIP/2.0 200 OK
CSeq: 1 INVITE
Contact: sip:g9610645@146.231.112.107:5060
Subject: test
Via: SIP/2.0/UDP cssipict.cs.ru.ac.za:5060; branch=3440148629-0
Via: SIP/2.0/UDP 146.231.123.15:5060
From: sip:ming@csmsc01ict.cs.ru.ac.za
Content-Type: application/sdp
Call-ID: 837041375@146.231.123.15
To: sip:g9610645@cssipict.cs.ru.ac.za; tag=889683254943.146.231.112.107
Content-Length: 129

v=0
o=g9610645 851846485795 1036147309 IN IP4 146.231.112.107
s=test
c=IN IP4 146.231.112.107
t=0 0
m=audio 1436 RTP/AVP 0

The server also receives and sends the "200 OK" message from user Ming to user g9610645. The two messages are essentially the same, so they are displayed here once.

## Log file from Ming's User Agent:

11/01/2002 12:41:37.547000

Sent to: 146.231.112.107:5060

ACK sip:g9610645@146.231.112.107:5060 SIP/2.0

Via: SIP/2.0/UDP 146.231.123.15:5060

Contact: sip:ming@146.231.123.15:5060

CSeq: 1 ACK

Subject: test

From: sip:ming@csmsc01ict.cs.ru.ac.za

Date: Fri, 01 Nov 2002 10:41:37 GMT

Call-ID: 837041375@146.231.123.15

To: sip:g9610645@cssipict.cs.ru.ac.za; tag=889683254943.146.231.112.107

Content-Length: 0

This ACK message is being sent from the caller (Ming) to the callee (g9610645) in order to complete the setup of the session.

11/01/2002 12:42:12.057000

Recv from: 146.231.112.107:5060

BYE sip:g9610645@146.231.112.107 SIP/2.0

Via: SIP/2.0/UDP 146.231.112.107:5060

CSeq: 2 BYE

Subject: test

From: sip:g9610645@cssipict.cs.ru.ac.za; tag=889683254943.146.231.112.107

Date: Fri, 01 Nov 2002 10:42:24 GMT
Call-ID: 837041375@146.231.123.15
To: sip:ming@csmsc01ict.cs.ru.ac.za
Content-Length: 0

Ming sends this BYE message to terminate the session with g9610645.

11/01/2002 12:42:12.797999
Sent to: 146.231.112.107:5060
SIP/2.0 200 OK
Contact: sip:ming@146.231.123.15:5060
Subject: test
CSeq: 2 BYE
Via: SIP/2.0/UDP 146.231.112.107:5060
From: sip:g9610645@cssipict.cs.ru.ac.za; tag=889683254943.146.231.112.107
Call-ID: 837041375@146.231.123.15
To: sip:ming@csmsc01ict.cs.ru.ac.za; tag=394468795226.146.231.123.15
Content-Length: 0

Ming receives the "200 OK" message as a notification that the session has terminated properly with g9610645.

Appendix C

# CINEMA File List

## Some files in *libcine* and their functionalities:

**Base64.c:** Converts given string to base64 encoding and decodes base64 strings to normal strings. This function is used for the Authorization header field in the REGISTER SIP sequence (described in Chapter 3).

**Error.c:** Error handling and debugging functions; useful when debugging the server.

**Host2ip.c:** Returns an IP address when given a hostname.

**Http.c:** Parses HTTP headers such as Content-Length, Content-Type, WWW-Authenticate and other authorization headers.

**Log.c:** Provides logging functions; useful to capture one's errors to a file.

**Parser.c:** Parses RFC822 headers such as accept, allow and contact.

**Tcp.c:** Handles TCP connections; contains the functions ReceiveTCP and ReceiveTCPRequests.

**Udp.c:** Handles UDP connections; contains the function ReceiveUDP.

## Some files in *libsip* and their functionalities:

**Authenticate.c:** Authenticates the SIP client.

**AuthenticateSIP.c:** Authenticates SIP requests.

**Client.c:** Contains the code for a thread, which handles a SIP client.

**Policy_core.c:** Executes the low-level core logic of a SIP transaction.

**Request.c:** Contains the functions to handle requests.

**Response.c:** Contains the functions to generate responses.

**Sip.c:** Parses SIP-specific headers.

## The actual code contained in the project `sipd`:

**Cgi.c:** Handles SIP-CGI request.

**Method.c:** Handles all the SIP requests.

**Proxy.c:** Proxies a SIP request.

**Redirect.c:** Handles redirection requests.

**Register.c:** Does the registration sequence for a user.

**Script.c:** Handles the action related to scripts, such as uploading, adding and removing scripts.

**Sipd.c:** The starting point for the operation of the server. It reads-in settings from the configuration file and the arguments from the command line. The server can also be run in the daemon mode.

Appendix D

# VOCAL SMS Service

**File:** CPLOpSMS.cxx

```
#include "CPLOpSMS.hxx"
//includes for sms
#include <sys/types.h>  /* socket() */
#include <sys/socket.h> /* socket() */
#include <netinet/in.h> /* sockaddr_in */
#include <arpa/inet.h>  /* sockaddr_in */
#include <stdio.h>    /* fdopen() */
#include <netdb.h>    /* getprotobyname() */
#include <string.h>  /* perror() */
#include <stdlib.h>  /* exit() */

#define SMS_SERVER "omniscient.ict.ru.ac.za"
#define PROXY_SERVER SMS_SERVER
#define PORT 80
#define AUTH "YXNobGV5OmFzaGxleQ=="
///
CPLOpSMS::CPLOpSMS()
{
#ifdef MEM_TEST
    cpLog(LOG_DEBUG, "CPLOpSMS[%x]", this );
#endif
}  //CPLOpSMS


///
CPLOpSMS::~CPLOpSMS()
{
#ifdef MEM_TEST
    cpLog(LOG_DEBUG, "~CPLOpSMS[%x]", this );
#endif
}  //~CPLOpSMS

bool
CPLOpSMS::setAttributes( const char* type, const char* value )
{
  if (strcmp( type, "destination" ) == 0 )
   myDestination = value;
  else
   if ( strcmp( type, "msg" ) == 0 )
     myMsg = value;
   else
     return false; //Unknown attribute return
   return true;
}

const char* const
CPLOpSMS::name() const
{
  return "CPLOpSMS";
}
```

```
void
CPLOpSMS::logData(void) const
{
  cpLog( LOG_DEBUG, "%s - destination(%s) msg(%s)", name(),
myDestination.c_str(), myMsg.c_str() );
}

bool
CPLOpSMS::setNextOperator(Sptr <Operator> anOperator, int nodeId =
CPLNodeEndofNodes)
{
  addOperator( anOperator );
  return true;
}

const Sptr < State >
CPLOpSMS::process(const Sptr <SipProxyEvent> anEvent )
{
  //Code that sends an instant message
   int sock;
   struct protoent * proto;
   struct sockaddr_in la, sa;
   struct hostent *h;
   FILE * stream;
   char buffer[1024], soap[1024], message[1024];

   /* gets IP address of host */
   if((h = gethostbyname(PROXY_SERVER)) == NULL) {
      perror("gethostbyname");
      exit(1);
   }

   /* creates an internet socket description */
   sa.sin_family = h->h_addrtype;
   memcpy((char *) &sa.sin_addr.s_addr, h->h_addr_list[0], h-
>h_length);
   sa.sin_port = htons(PORT);

   /* bind any port number */
   la.sin_family = AF_INET;
   la.sin_addr.s_addr = htonl(INADDR_ANY);
   la.sin_port = htons(0);

   /* we could just say 6 and avoid this call, but getprotobyname() is
the right way to do it */
   proto = getprotobyname("tcp");

   /* create a network socket */
   if((sock = socket(PF_INET, SOCK_STREAM, proto->p_proto)) == -1) {
      perror("can not create socket");
      exit(1);
   }

   /* bind the local address */
   if(bind(sock, (struct sockaddr *)&la, sizeof(la)) == -1) {
      perror("can not bind");
```

```
      exit(1);
   }

   /* connect to the remote address */
   if(connect(sock, (struct sockaddr *)&sa, sizeof(sa)) == -1) {
      perror("can not connect");
      exit(1);
   }

   /* associate a stream with this socket */
   if((stream = fdopen(sock, "a+")) == NULL) {
      perror("can not associate stream with socket");
      exit(1);
   }

   /* we should now be connected to the server, so we can prepare a
message */

      sprintf(message, "%s%s%s%s%s%s%s%s",
         "      <sms:sendsms>\n",
         "        <sms:phone>", myDestination, "</sms:phone>\n",
         "        <sms:message>", myMsg, "</sms:message>\n",
         "      </sms:sendsms>\n"
      );

   /* encapsulate the message in some soap */
   sprintf(soap, "%s%s%s%s%s%s%s%s",
      "<?xml version=\"1.0\" encoding=\"iso-8859-1\" ?>\n",
      "<soap:Envelope       xmlns:soap=\"http://www.w3.org/2001/12/soap-
envelope\">\n",
      "  <soap:Body>\n",
      "                                                              <sms:sms
xmlns:sms=\"http://omniscient.ict.ru.ac.za/sms/sms.xsd\">\n",
      message,
      "    </sms:sms>\n",
      "  </soap:Body>\n",
      "</soap:Envelope>\n"
   );

   /* send the request to the web server */
   fprintf(stream, "POST http://%s/sms/ HTTP/1.0\n", SMS_SERVER);
   fprintf(stream, "Host: %s\n", SMS_SERVER);
   fprintf(stream, "Content-Length: %d\n", strlen(soap));
   fprintf(stream, "Authorization: Basic %s\n", AUTH);
   fprintf(stream, "User-Agent: netsms.c\n\n");
   fprintf(stream, "%s\n", soap);

   /* reclaim memory */
   free(soap); free(message);

   /* read the response from the web server */
   while(fgets(buffer, sizeof(buffer), stream) != NULL) {
      printf("%s", buffer);
   }

  return CPLOperator::process( anEvent );
}
```

**File:** CPLOpSMS.hxx

```
#ifndef _CPLOPSMS_HXX
#define _CPLOPSMS_HXX
#include "CPLOperator.hxx"

class CPLOpSMS: public CPLOperator
{
 public:
  CPLOpSMS();
  ~CPLOpSMS();
  bool setAttributes( const char* type, const char* value );

  const Sptr < State > process(const Sptr <SipProxyEvent> anEvent );
  bool   setNextOperator(Sptr<Operator>   anOperator,   int   nodeId   =
CPLNodeEndofNodes);
  const char* const name() const;
  void logData(void) const;

 private:
  string myDestination;
  string myMsg;

};
#endif
```

# References

NB: Material on the Internet has been collected on an attached disc for easy retrieval.


[Allen, 2000] Allen, D. 2000. Megaco and MGCP. Commweb Article. http://www.commweb.com/shared/article/showArticle.jhtml?articleID=8702913

[Andreetto et al., 2001] Andreetto, A., Canal, G., Lago, P., Licciardi, C.A. 2001. An architecture for IN-internet hybrid services. Computer Networks, 35(2001): 537-549.

[Anjum et al., 2001] Anjum, F., Caruso, F., Jain, R., Missier, P., Zordan, A. 2001. CitiTime: a system for rapid creation of portable next-generation telephony services. Computer Networks, 35(2001): 579-595.

[Arango et al., 1999] Arango, M., Dugan, A., Elliott, I., Huitema, C. and Pickett, S. 1999. Media Gateway Control Protocol (MGCP) Version 1.0. Request for Comments 2705, IETF, October 1999.

[Berners-Lee et al., 1997] Berners-Lee, T., Fielding, R., Gettys, J., Mogul, J. and Nielsen, H. 1997. HyperText Transfer Protocol — HTTP/1.1. Request for Comments 2068, IETF, January 1997.

[Berners-Lee et al., 1998] Berners-Lee, T., Fielding, R. and Masinter, L. 1998. Uniform Resource Identifiers (URI): Generic Syntax. Request for Comments 2396, IETF, August 1998.

[Chapron and Chatras, 2001] Chapron, J.E. and Chatras, B. 2001. An analysis of the IN call model suitability in the context of VoIP. Computer Networks, 35(5): 521-535.

[Chatzipapadopoulos, 2000] Chatzipapadopoulos, F., De Zen, G., Magedanz, T., Venieris, I.S., Zizza, F. 2000. Harmonised Internet and PSTN service provisioning. Computer Communications, 23(2000): 731-739.

[Collins, 2000] Collins, D. 2000. Carrier-Grade Voice Over IP (2nd edition). McGraw-Hill Professional, New York.

[Dalgic and Fang, 1999] Dalgic, I. and Fang, H. 1999. Comparison of H.323 and SIP for IP telephony signaling. Proceedings of Photonics East, Boston, Massachusetts.

[Dang et al., 2002] Dang, L., Jennings, C. and Kelly, D.G. 2002. Practical VoIP Using VOCAL (1st edition). O'Reilly & Associates, Sebastopol, CA.

[Davidson and Peters, 2000] Davidson, J. and Peters, J. 2000. Voice Over IP Fundamentals (1st edition). Cisco Press, Indianapolis, IN.

[Glasmann et al., 2003] Glasmann, J., Kellerer, W. and Müller, H. 2003. Service architectures in H.323 and SIP – a comparison. IEEE Communications Surveys and Tutorials, 5(2) http://www.comsoc.org/livepubs/surveys/public/2003/oct/index.html

[Glitho, 2001] Glitho, R.H. 2001. Emerging alternatives to today's advanced service architectures for Internet telephony: IN and beyond. Computer Networks, 35(5): 551-563.

[Gurle et al., 1999] Gurle, D., Hersent, O. and Petit, J.P. 1999. IP Telephony Packet-based Multimedia Communications Systems (1st edition). Addison Wesley Publishing, Boston, MA.

[Halse and Wells, 2002] Halse, G. and Wells, G. 2002. A bi-directional SOAP/SMS gateway service. Southern African Telecommunication Networks and Applications Conference (SATNAC) Proceedings, Champagne Resort, Eastern Cape, South Africa.

[Handley and Jacobson, 1998] Handley, M. and Jacobson, V. 1998. SDP: Session Description Protocol. Request for Comments 2327, IETF, April 1998.

[Handley et al., 1999] Handley, M., Schulzrinne, H., Schooler, E. and Rosenberg, J. 1999. SIP: Session Initiation Protocol. Request for Comments 2543, IETF, March 1999.

[Hsieh et al., 2001] Hsieh, M., Okuthe, J. and Terzoli, A. 2001. Deploying a SIP environment in which to study service creation. Southern African Telecommunication Networks and Applications Conference (SATNAC) Proceedings, Wild Coast Sun, KwaZulu-Natal, South Africa.

[Hsieh et al., 2002] Hsieh, M., Okuthe, J. and Terzoli, A. 2002. An investigation into multimedia service creation using SIP. Southern African Telecommunication Networks and Applications Conference (SATNAC) Proceedings, Champagne Resort, Eastern Cape, South Africa.

[IEC, 2003a] International Engineering Consortium (IEC). 2003. On-line tutorials: Intelligent Network (IN), last accessed November 2003: http://www.iec.org/online/tutorials/in/index.html.

[IEC, 2003b] International Engineering Consortium (IEC). 2003. On-line tutorials: International Intelligent Network (IN), last accessed November 2003: http://www.iec.org/online/tutorials/intern_in/index.html.

[IEC, 2003c] International Engineering Consortium (IEC). 2003. On-line tutorials: H.323, last accessed November 2003: http://www.iec.org/online/tutorials/h323/.

[Jacobs and Clayton, 2002] Jacobs, A. and Clayton, P. 2002. Utilizing MGCP to design an H.323 endpoint SMS service. Southern African Telecommunication Networks and Applications Conference (SATNAC) Proceedings, Champagne Resort, Eastern Cape, South Africa.

[Jacobs and Clayton, 2003] Jacobs, A. and Clayton, P. 2003. Investigating call control using MGCP. Southern African Telecommnunication Networks and Applications Conference (SATNAC) Proceedings, Fancourt Hotel and Country Club Estate, Eastern Cape, South Africa.

[Jiang et al., 2002] Jiang, W., Singh, K., Lennox, J., Narayanan, S. and Schulzrinne, H. 2002. CINEMA: Columbia InterNet Extensible Multimedia Architecture. Columbia University, Technical Report Number CUCS-011-02.

[Johnston, 2001] Johnston, A. 2001. SIP: Understanding the Session Initiation Protocol (1st edition). Artech House, London.

[Lennox et al., 1999a] Lennox, J., Schulzrinne, H. and La Porta, T.F. 1999a. Implementing Intelligent Network Services with the Session Initiation Protocol. Columbia University, Technical Report Number CUCS-002-99.

[Lennox et al., 1999b] Lennox, J., Rosenberg, J. and Schulzrinne, H. 1999b. Programming Internet Telephony Services. Columbia University, Technical Report Number CUCS-010-99.

[Lennox and Schulzrinne, 2000a] Lennox, J. and Schulzrinne, H. 2000a. Call Processing Language framework and requirements. Request for Comments 2824, IETF, May 2000.

[Lennox and Schulzrinne, 2000b] Lennox, J. and Schulzrinne, H. 2000b. CPL: A language for user control of internet telephony services, Internet Draft, IETF, November 2000.

[Lennox et al., 2001] Lennox, J., Schulzrinne, H. and Rosenberg, J. 2001. Common Gateway Interface for SIP. Request for Comments 3050, IETF, January 2001.

[Nortel Networks, 2000a] Nortel Networks. 2000a. A comparison of H.323v4 and SIP. Nortel Networks, Technical Document S2-00505, Santa Clara, CA.

[Nortel Networks, 2000b] Nortel Networks. 2000b. White Paper – The role of Megaco/H.248 in media gateway control: a protocol standards overview, Santa Clara, CA.

[OpenH323, 2003] OpenH323. 2003. The OpenH323 Project. Coordinated by Equivalence Pty Ltd., New South Wales, Australia, last accessed March 2003: http://www.openh323.org.

[Penton et al., 2001a] Penton, J.B., Terzoli, A., and Wentworth, P. 2001a. Deploying a feature-rich H.323 environment in which to practice the creation of services. Southern African Telecommunication Networks and Applications Conference (SATNAC) Proceedings, Wild Coast Sun, KwaZulu-Natal, South Africa.

[Penton et al., 2001b] Penton, J.B., Terzoli, A., and Wentworth, P. 2001b. Retrieving emails via traditional PSTN telephones, an H.323 service. Southern African Telecommunication Networks and Applications Conference (SATNAC) Proceedings, Wild Coast Sun, KwaZulu-Natal, South Africa.

[Penton and Terzoli, 2002] Penton, J.B. and Terzoli, A. 2002. CANS: Customizable Alarm Notification System, an H.323 signaling service. Southern African Telecommunication Networks and Applications Conference (SATNAC) Proceedings, Champagne Resort, Eastern Cape, South Africa.

[Radvision Corporation, 2002] Radvision Corporation. 2002. White Paper – Implementing Media Gateway Control Protocols, Radvision Corporation, New York.

[Rosenberg and Schulzrinne, 1998a] Rosenberg, J. and Schulzrinne, H. 1998a. The Session Initiation Protocol: providing advanced telephony services

across the internet. Bell Labs Technical Journal, Oct.-Dec. 1998: 144-160.

[Rosenberg and Schulzrinne, 1998b] Rosenberg, J. and Schulzrinne, H. 1998b. A comparison of SIP and H.323 for Internet Telephony. Proceedings of the International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV). Cambridge, U.K, July 1998.

[Rosenberg and Schulzrinne, 1999] Rosenberg, J. and Schulzrinne, H. 1999. Internet Telephony: architecture and protocols – an IETF perspective. Computer Networks, 31(3), 237-255.

[Rosenberg and Shockey, 2000] Rosenberg, J. and Shockey, R. 2000. The Session Initiation Protocol: a key component for Internet telephony. Commweb Article. http://www.cconvergence.com/shared/article/showArticle.jhtml?articleID=8700868

[Singh and Schulzrinne, 2000] Singh, K. and Schulzrinne, H. 2000. Unified messaging using SIP and RTSP. IP Telecom Services Workshop, page 7, Atlanta, Georgia, September. 2000 http://www1.cs.columbia.edu/~kns10/publication/vmail.pdf

[SIPCharter, 2003] SIPCharter. 2003. The SIP-Charter Working Group at the IETF, last accessed March 2003: http://www.ietf.org/html.charters/sip-charter.html.

[SOAP, 2000] SOAP: Simple Object Access Protocol. 2000. W3C Note, 8 May 2000, last accessed March 2003: http://www.w3.org/TR/SOAP.

[Vovida, 2000] Vovida. 2000. Voice Mail Feature, 28 February 2000: http://www.vovida.org/document/pdf/VoiceMail.pdf

[Vovida, 2001a] Vovida. 2001a. VOCAL System Architecture, 20 February 2001: http://www.vovida.org/document/Training/2_VOCAL_Architecture.pdf

[Vovida, 2001b] Vovida. 2001b. Call Processing Language Feature Server, 5 June 2001: http://www.vovida.org/document/pdf/feature_server_CPL.pdf

[Vovida, 2001c] Vovida. 2001c. SIP User Agent, 30 May 2001: http://www.vovida.org/document/pdf/user_agent.pdf

[Vovida, 2001d] Vovida. 2001d. Media Gateway Control Protocol (MGCP) Stack, 22 March 2001: http://www.vovida.org/downloads/mgcp/README-mgcp-1.2.0.txt

[Vovida, 2003] Vovida. 2003. Why Open Source at Vovida.org?, last accessed March 2003: http://www.vovida.org/fom-serve/cache/485.html

[Wind River, 2002] Wind River. 2002. White Paper – SIP vs. H323: a business analysis, 12 February 2002: http://www.windriver.com/whitepapers/sip.pdf