ECOLE CENTRALE D'ELECTRONIQUE

ece

GROUPE ECE

DESIGN AND IMPLEMENTATION OF SIP VOIP ADAPTER

Author: Adrià Guixà Ibànez

Thesis Director: Dr. Laurent Gerorge

Paris, December 15, 2009

Abstract

The SIP VoIP Adapter is a Java application that is able to establish a SIP communication acting as a User Agent, which uses an external device as a sound device, to play and acquire the audio from the call established through Asterisk PBX.
The SIP VoIP Adapter has been also implemented to be able to use and independent audio data codecs for both audio communications, the RTP communications between SIP clients, and the UDP communication between the SIP VoIP Adapter and the 6lowPAN end device.

Acknowledgements

In the first place, I would like to thank my thesis director, Dr, Laurent George, for accepting me as a Erasmus student here, at the Ecole Central d'Electronique de Paris (ECE), and Mohammed Ikbal Benakila, since he has been so helpful and, in the meantime, he has made me feel comfortable at the working time along this season.

Firstly, the most deserved acknowledgement to my parent, Josep and Roser, and my brother, Roger. It is because of your wholehearted support that I finished my degree.

Secondly, I am grateful to all my new friends, here in Paris, Arguine, Carolina, Claudia, Domingo, Frider, Ines, Juanito, Lucia, Marta, Nicolas, Rossella, Silvia and Valenti. I would like to thank you for all your help and for the great moments that we had here.

To all my closest friends from Barcelona, like Agusti, Aleix, Ana B., Ana E, Anxo, Guarch, Josep, Mireia, Nacho, Pau, Ruth, Saul and Sobri. Without all of you, all these tough years would not have been the same including our trips, our spare time and, above all, the time we spent together at the university.

Finally I would like to thank my closest friends from my town, Esparreguera, like Alex, Anna, Alvaro, Arnau, Esteve, Esther, Ferran, Genis, Iban, Ingrid, Joana, Laia, Marta, Miquel, Pages, Sheila and Roger. After all that we have enjoyed together and everything that we have next, I felt like telling you that.

"Do, or do not... There is no try."

Jedi Master Yoda.
Star Wars: The Empire Strikes Back

Contents

List of Figures

Chapter 1

Introduction

1.1 Background

The Session Initiation Protocol (SIP) is a signaling protocol for initiating, managing and terminating voice and video sessions across packet networks, developed in the IETF. SIP is ASCII-based, resembling HTTP, and reuses existing IP protocol (DNS, SDP, etc.) to provide media setup and teardown. It also may be extended to accommodate features and services such as call control services, mobility, interoperability with existing telephony systems, and more.

Nowadays, SIP protocol has been generally accepted as one of the most important services available for audio communications over Internet. Thanks to its great features, several private branches exchange has adapted SIP as the main protocol for Voice over Internet Protocol (VoIP) services.

One of the most important PBX software applications is Asterisk, which is an open source software implementation that allows attached telephones to make calls to another, and connect Voice over Internet Protocol (VoIP) services interoperating with SIP signaling protocol.

1.2 Motivation

Next to body language, speech is the most natural mode of human communication. This is probably the reason why the main form of fast communication is the phone call. Nowadays, Internet proliferation in households around the world, in addition to its great potential, has allowed a rapid introduction of IP telephony, which consist of the transmission of voice conversations over IP-based networks.

Thanks to the great potential offered by IP telephony as for example the possibility of making free calls, as well as the introduction of Private Branch Exchange's (PBX) to manage these communications. VoIP (Voice Over IP) has begun to be accepted as the internal way of communication of many organizations. Actually, the vast majority of employee in organizations, has a phone IP in their desk, however, if he or she is highly mobile, this is no longer feasible, and the need to use a mobile phone is more than evident, despite their high cost. Therefore, in environments where the employee moves inside a limited area, such as a building, it could be a great idea to cut costs providing a local interface for the wireless communication.

Based on these premises, 6lowPAN is a very interesting low cost solution in the deployment of radio communication, providing enough transmission rates for voice communication. Thus, a 6lowPAN end device could be used as a VoIP.

1.3 Objectives

Due to the memory limitations in the hardware embeded in 6lowPAN devices, it has been necessary to chose another solution instead of implementing a VoIP softphone application on the 6lowPAN device. Therefore, the option adobted has been break the VoIP softphone down into two parts: the SIP signaling treatment and the audio media processing.

Most of 6lowPAN devices provide audio playing and audio acquiring from a microphone, as well as interoperability with existing wired networks over Internet Protocol. Therefore, this features can provide us the solution to be applied on the SIP VoIP softphone implementation. So, the 6lowPAN device could be used as a sound device (acquiring and playing the audio), and will be the computer where the 6lowPAN router is connected, the responsible of manage SIP signaling.

Therefore, the goal of this master thesis is to design and implement a SIP phone application, called SIP VoIP Adapter, able to use a particular external device as a sound device, acquiring and playing the audio from the SIP call application. The external device to use is a 6lowPAN device provided with a speaker and microphone, and it will be interconnected with the computer over Internet Protocol.
Furthermore, the 6lowPAN router and the 6lowPAN end device configurations are not part of this thesis.

1.4 Thesis Organization

This thesis is divided into ten chapters. In the first chapter, the reader is introduced to the background of the study, followed by problem discussion and objectives. The chapters two and three are a protocols review in which the reader is provided with the main concepts of the Session Initial Protocol (SIP) and the Real-Time Transport Protocol (RTP) to understand the subsequent work. Then the reader is presented with the technology chapter which discusses the Java Technologies used for the SIP VoIP Adapter's implementation. Chapter five discusses the design and functionality of the classes that conforms the SIP VoIP Adapter. The next chapter presents the class design and functionality of the Test application, developed to test the SIP VoIP Adapter. Chapter seven covers the installation and configuration of Asterisk PBX, as well as the system configuration where to test the SIP VoIP Adapter. In chapter eight, the reader is presented with the experimental results and analysis of the SIP VoIP Adapter functionality. The results which come after analysis the experimental results are explained in the conclusions, as well as the future work to perform.

Chapter 2

Session Initial Protocol (SIP)

2.1 Introduction

This chapter provides a basic knowledge of Session Initial Protocol (SIP). It describes the architecture and the components in the SIP infrastructure to acquire a basic knowledge of SIP, required to understand the implementation of this thesis work. This chapter also describes the registration and call operations needed in a SIP infrastructure network.

2.2 SIP Overview

SIP is an application layer signaling protocol, developed by the Internet Engineering Task Force (IETF), that enables the creation, modification, and termination of sessions that are independent of the underlying transport protocols and session type being established. SIP is a client-server protocol that closely resembles two other internet protocols, HTTP and SMTP, in which requests are issued by the client and responses are managed by the servers. The messages exchanged between them contain the information required for establishing a session. The SIP purpose is the communication between end multimedia devices. It makes this communication possible thanks to the protocols RTP/RTCP and SDP. The RTP protocol is used for carrying voice data in real time, while the SDP protocol is used for the negotiation capabilities of participants, type of coding, etc.

2.3 SIP Components

SIP supports functionalities for the establishment and termination in multimedia sessions: localization, availability, resource utilization, and negotiation characteristics. For the implementation of this functionalities, SIP offers several components, which can be defined by two fundamental ones: User Agents (UA) and servers.

2.3.1 User Agent (UA)

A User Agent is an application that can initiate, receive, and terminate a call in a SIP session. The user agent initiating a call acts as a client while sending a initial INVITE request and acts as a server while receiving a BYE request from the client.

It is formed by two different parts, the User Agent Client (UAC) and the User Agent Server (UAS). UAC is a logical entity which creates SIP requests and receive responses to that requests. UAS is a logical entity which creates responses to SIP requests. Both are included in all user agents, to allow the communication between user agents through client-server communications.

2.3.2 SIP Servers

The server components of the SIP infrastructure can be divided into three types:

Proxy Server

A proxy server is an intermediate entity that receives SIP requests and forwards them on behalf of the requester. It works as a client and server to enable the call establishment between users. This server has the functionality to enclose route the requests which receives from another entities closer to the destination.

There are two different types of Proxy Severs:

- **Stateful Proxy:** it keeps the transaction states while requests are being processed. It lets break the requests down into several (forking), with the aim of the parallel location of the call, to obtain the best response to be send to the user who made the call.

- **Stateless Proxy:** it does not keep the transaction state while requests are being processed, it only resends messages.

Registrar Server

This is a server that accepts register requests from the user and it keeps information about this requests to offer a localization service and addresses translation in the domain controlled by itself. It also supports authentication A client must register with the registrar server each time a user turns on the SIP user client. The SIP VoIP Adapter have been developed to interact with this kind of Proxy Servers as Asterisk PBX does.

Redirect Server

This is a server that creates redirect responses to the received requests. It redirects the requests through the next server.

2.4 SIP Messages

A SIP message is a simple text message that is similar to HTTP and SMTP messages. It is either a request from a client to a server, or a response from a server to a client. There are two types of SIP messages: Request and Response. Both of them use the same basic format. It consists of a start-line, one or more header fields, an empty line indicating the end of header fields, and an optional message-body. However, the syntax differs in character set and syntax specifics.

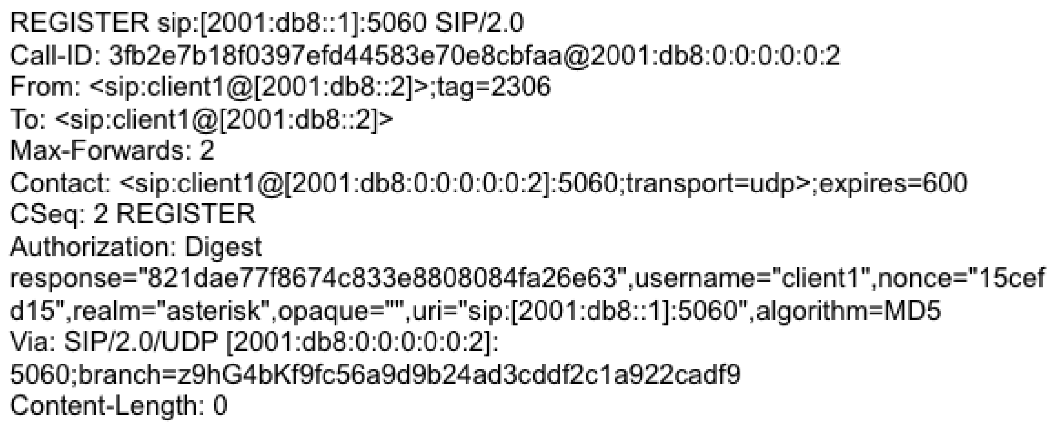
2.4.1 SIP Request Messages

A request is a message sent from the UAC to the server. SIP requests are characterized by the initial message line, called Request-Line, that contains the method name, the Request-URI and the version of the SIP protocol.

There are six basic SIP methods (defined in the RFC 254) which describes the client requests:

- **INVITE:** it lets invite a user or service to participate in a session or to modify parameters in an already existing session.

- **ACK:** it confirms a session establishment.
- **OPTION:** it requests information about the server capacity.
- **BYE:** it indicates the session termination.
- **CANCEL:** it cancels a pending request.
- **REGISTER:** it registers contact information with a SIP server.



```
REGISTER sip:[2001:db8::1]:5060 SIP/2.0
Call-ID: 3fb2e7b18f0397efd44583e70e8cbfaa@2001:db8:0:0:0:0:0:2
From: <sip:client1@[2001:db8::2]>;tag=2306
To: <sip:client1@[2001:db8::2]>
Max-Forwards: 2
Contact: <sip:client1@[2001:db8:0:0:0:0:0:2]:5060;transport=udp>;expires=600
CSeq: 2 REGISTER
Authorization: Digest
response="821dae77f8674c833e8808084fa26e63",username="client1",nonce="15cef
d15",realm="asterisk",opaque="",uri="sip:[2001:db8::1]:5060",algorithm=MD5
Via: SIP/2.0/UDP [2001:db8:0:0:0:0:0:2]:
5060;branch=z9hG4bKf9fc56a9d9b24ad3cddf2c1a922cadf9
Content-Length: 0
```

Figure 2.1: Request message example

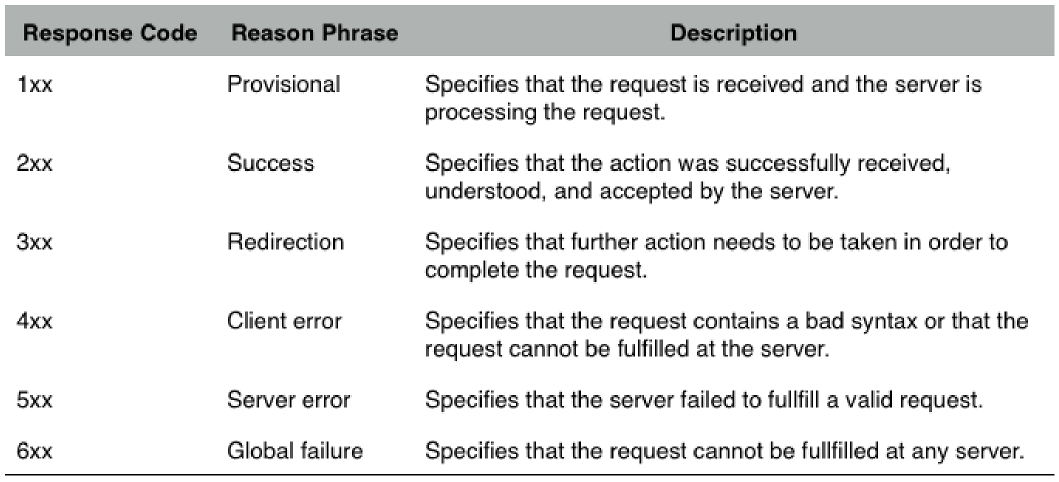
2.4.2 SIP Response Messages

After reception and following interpretation of the SIP message request, the receiver response with a message. This message, is close to the previous one, differing on the first line, called Status-Line, which contains the SIP version, the Status-Code, and a Reason-Phrase. The response code is composed by three digits which allows to classify the several existing types.

The first digit defines the response type. See Figure 2.3.

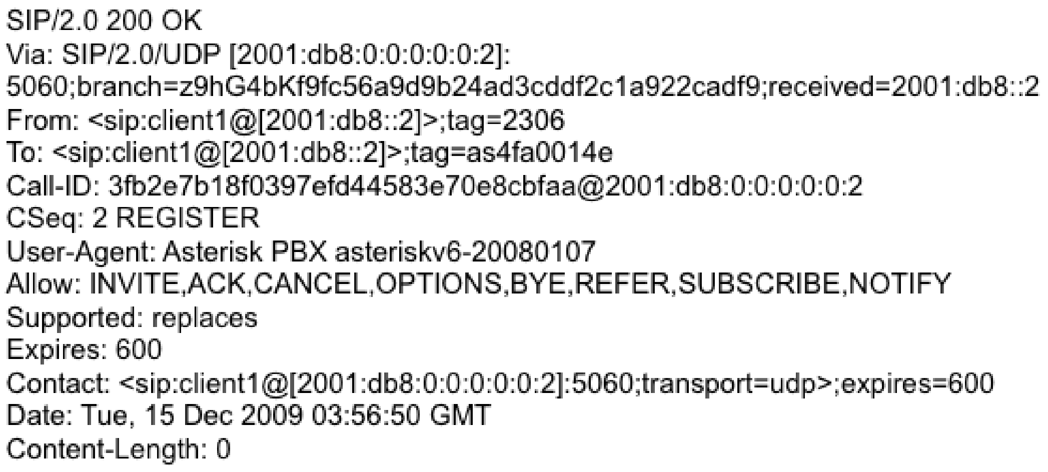
2.4.3 SIP Message Parts

The both SIP messages, responses and requests, are divided in three different parts:



Response Code	Reason Phrase	Description
1xx	Provisional	Specifies that the request is received and the server is processing the request.
2xx	Success	Specifies that the action was successfully received, understood, and accepted by the server.
3xx	Redirection	Specifies that further action needs to be taken in order to complete the request.
4xx	Client error	Specifies that the request contains a bad syntax or that the request cannot be fulfilled at the server.
5xx	Server error	Specifies that the server failed to fullfill a valid request.
6xx	Global failure	Specifies that the request cannot be fullfilled at any server.

Figure 2.2: Response Types



```
SIP/2.0 200 OK
Via: SIP/2.0/UDP [2001:db8:0:0:0:0:0:2]:
5060;branch=z9hG4bKf9fc56a9d9b24ad3cddf2c1a922cadf9;received=2001:db8::2
From: <sip:client1@[2001:db8::2]>;tag=2306
To: <sip:client1@[2001:db8::2]>;tag=as4fa0014e
Call-ID: 3fb2e7b18f0397efd44583e70e8cbfaa@2001:db8:0:0:0:0:0:2
CSeq: 2 REGISTER
User-Agent: Asterisk PBX asteriskv6-20080107
Allow: INVITE,ACK,CANCEL,OPTIONS,BYE,REFER,SUBSCRIBE,NOTIFY
Supported: replaces
Expires: 600
Contact: <sip:client1@[2001:db8:0:0:0:0:0:2]:5060;transport=udp>;expires=600
Date: Tue, 15 Dec 2009 03:56:50 GMT
Content-Length: 0
```

Figure 2.3: Response message example

Start Line

The start line denotes the beginning of a SIP message. A start line can be either a request or a response. A response start line contains the SIP version number, the status code, and the description for the response phrase. The request start line contains a method name, the SIP URI to which the message is sent, and the SIP version number, the status code, and the description for the response phrase.

Headers

The header contains fields that provide additional information about a message, such as From, To, Subject, and Via. Certain header fields, such as From, To, and Subject, contain only a single header field value and appear only once in the header. Another certain header field values, such as Via, Contact, and Route, can appear multiple times in a header and can contain multiple values separated by a comma.

The main header fields are:

- **Via:** it shows the send used transport and identifies the route of the request, therefore each proxy adds a line in this field.
- **From:** it represents the request origin.
- **To:** it represents the request destiny.
- **Call-ID:** it represents a single identifier of the communication between two user agents.
- **CSeq:** it represents the request sequential number.
- **Contact:** it represents the name / origin URI.
- **User Agent:** it represents the user agent client that creates the communication.

Message Body

It contains information about the message that must be passed to the receiver. The message can be either a plain text message or a multimedia message. The Session Description Protocol (SDP) describes the session parameters for a multimedia message. The message body is independent of the SIP protocol and can contain any information.

2.5 SIP Operations

This section shows the SIP message exchange generated to establish a communication between two SIP clients through a Proxy Registrar Server. It presents the process of registration, as well as the process of making a call, with the aim to understand the SIP VoIP Adapter design.

2.5.1 Registration

The entire registration operation of SIP is described in Figure 2.4. It shows how the client initialize the registration by sending a REGISTER request to the SIP Proxy Registrar, then if an authorization is not required by the proxy, it replies with a 200 OK response to the SIP client. Moreover, if it needs an authorization, it replies with a 401 Unauthorized so, the client needs to resend a REGISTER request with the authentication required by the SIP Proxy. If the authentication is valid, the Registrar Server replies with a 200 OK response, allowing the client to make a call. This registration operation is an element of the node discovery mechanism of SIP.

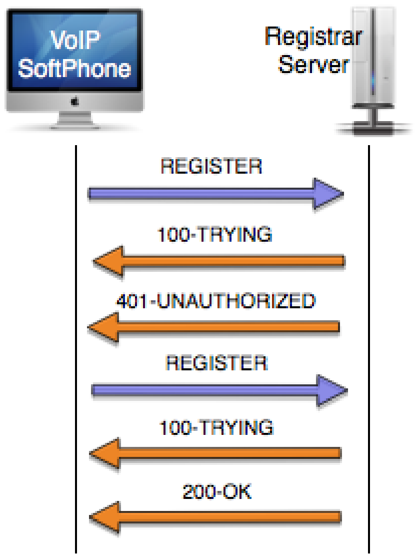


Figure 2.4: SIP Server Register Request

2.5.2 Call Process

After establishing the registration operation, the client is able to dial up another clientÕs SIP URI, being this client also registered in the server, by sending an INVITE request containing a SDP message through the SIP server. The SIP server forwards this INVITE request to the other user, and it also sends back a 100 TRYING response to the caller client. The client called receive the INVITE request and alerts the user to answer the phone call. It also replies to the SIP caller client with a 180 RINGING response, which is routed back along the reverse path of the INVITE request routing path. After the SIP called client answer the call, it sends to the caller client a 200 OK response which contains a SDP message. After the caller client receives the 200 OK response, both SIP clients have learned the endpoint IP address and media description from each other. So, the caller client send an ACK message to the called client to acknowledge the session is build. Therefore the media session can be started between both SIP clients. When one of them wants to end the conversation, first send a BYE request to other client through the SIP server, and then this message is answered by a 200 OK message to confirm the whole call ends.

2.6 SIP Media Session

The INVITE request message, and its consequent 200 OK response message, contains a SDP description to configure the media session that both client are going to start. SDP is a standard description protocol of media content and how multimedia information is transported. SDP must contain sufficient information to enable the session participants to understand each other. This message body contains several fields to describe the media session as Figure 2.6 shows.

2.7 Summary

A primary SIP knowledge was given to help for going deeper into this thesis work. This chapter has discuses the SIP architecture and its main components, as well as the exchange messages which defines the protocol and how it negotiates to establish a communication through a Resgistar Server.

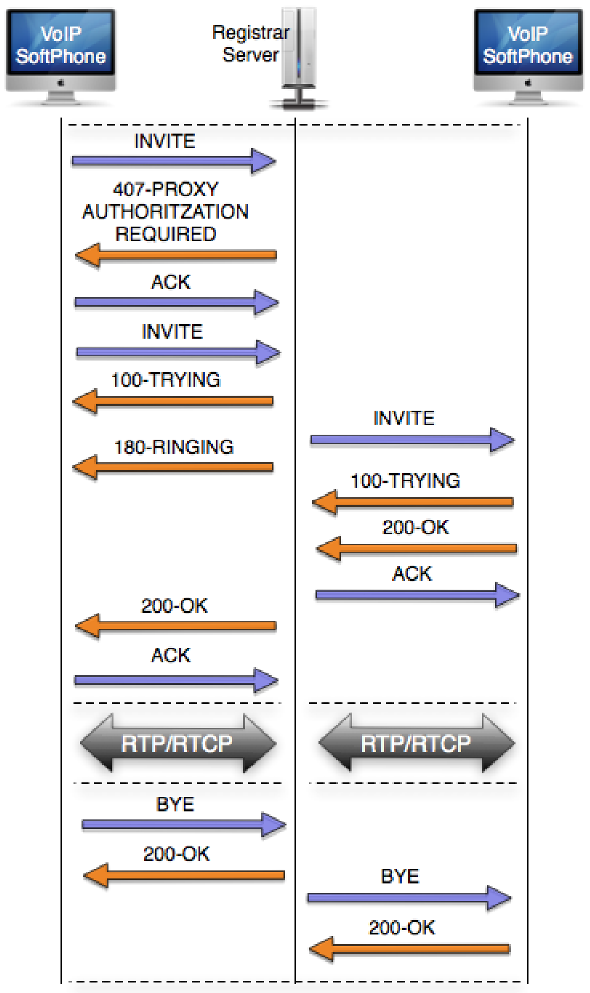
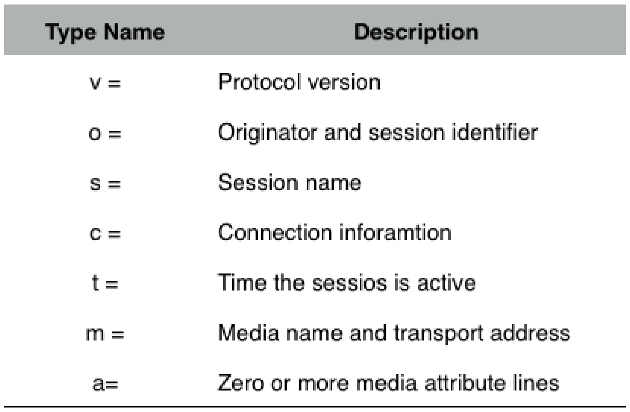


Figure 2.5: SIP messages exchange between clients



Type Name	Description
v =	Protocol version
o =	Originator and session identifier
s =	Session name
c =	Connection inforamtion
t =	Time the sessios is active
m =	Media name and transport address
a=	Zero or more media attribute lines

Figure 2.6: SIP Media Session parameters

Chapter 3

RTP and RTCP

3.1 Introduction

This chapter provides a basic knowledge of Real-Time Transport Protocol (RTP) and Real-Time Control Protocol (RTCP) functionality and packet format. RTP is a header format and control protocol designed to support applications transmitting real-time data over unicast or multicast network services. Therefore, this chapter pretends to describe the main characteristics of both internet protocols for helping to understand the implementation of this thesis work.

3.2 RTP Overview

RTP provides a way to transmit time-based media over the network, adding synchronization and feedback features over the existing transport protocol, thought it is often used over UDP. RTCP is an integrated media synchronization function used together with RTP, to exchange RTCP messages generated periodically for adding system-level functionality to its related RTP streams. RTCP also provides control and identification mechanisms for RTP transmission.

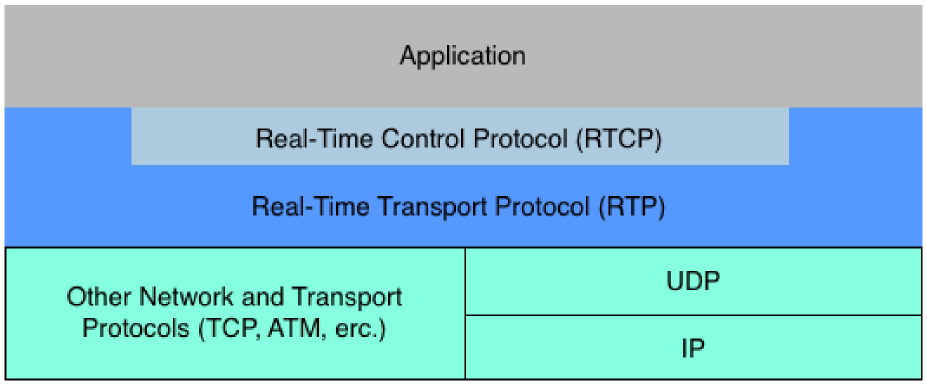

Application
Real-Time Control Protocol (RTCP)
Real-Time Transport Protocol (RTP)
Other Network and Transport Protocols (TCP, ATM, erc.)
UDP
IP

Figure 3.1: RTP Architecture

3.3 RTP Architecture

An RTP session is an association among a set of applications communicating with RTP. It is identified by a network address and two ports which are used, one for the media data and the other for control data. A participant is a user participating in the session, which it can work as a receiver, transmitter, or both, to receive and transmit media data. A user is able to transmit different type of media, each of them in a different session.

3.3.1 Data Packets

A series of packets are transmitted as the media data for a session, this data packets which are originated from a particular source are called RTP streams. Each RTP data packet in a stream contains two structured parts, the header and the data.

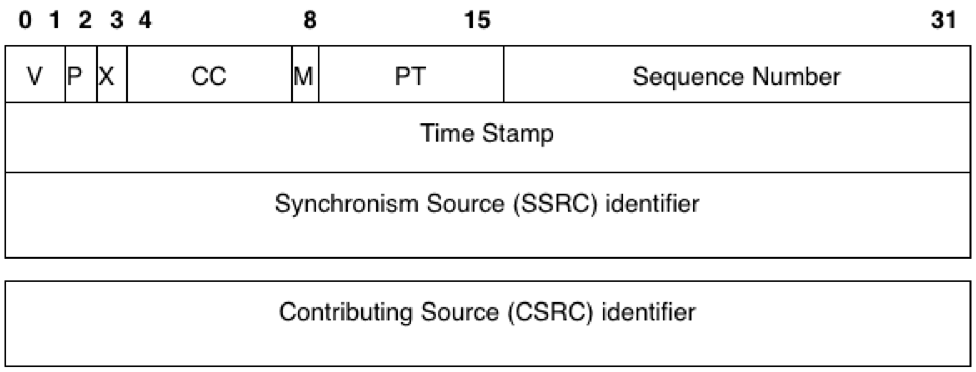

0 1 2 3 4 8 15 31
V
P
X
CC
M
PT
Sequence Number
Time Stamp
Synchronism Source (SSRC) identifier
Contributing Source (CSRC) identifier

Figure 3.2: RTP Header Format

The structured RTP data packet's header contains the following fields:

- **Version Number (V):** 2 bits. Indicates the version of the protocol. Current version is 2.
- **Padding (P):** 1 bit. Used to indicate if there are extra padding bytes at the end of the RTP packet.
- **Extension (X):** 1 bit. Indicates presence of an Extension header between standard header and payload data.
- **CSRC Count (CC):** 4 bits. Contains the number of CSRC identifiers that follow the fixed header.
- **Marker (M):** 1 bit. Used at the application level and is defined by a profile. If it is set if the current data has some special relevance for the application.
- **Payload Type (PT):** 7 bits. Indicates the format of the payload and determines its interpretation by an RTP profile. The payload mappings for audio and video are specified in RFC 1890.
- **Sequence Number:** 16 bits. It is incremented by one for each RTP data packet sent and is to be used by the receiver to detect packet loss and to restore packet sequence. The RTP does not take any action when it sees a packet loss, it is left to the application to take the desired action.
- **Timestamp:** 32 bits. Used to enable the receiver to playback the receiver samples at appropriate intervals. It determines the transmission delay and jitter level. When several media streams are presented, the timestamps are independent in each stream.
- **SSRC:** 32 bits. Synchronization source identifier uniquely identifies the source of a stream. The SSRC within the same RTP session will be unique.
- **CSRC:** 32 bits. Contributing Source identifiers enumerate contributing sources to a stream which has been generated from multiples sources.

3.3.2 Control Packets

Control data (RTCP) packets are sent periodically to all the participants in the session, containing information about the quality of service, information about the source of the media being transmitted, and transmission statics.

It exists several types of RTCP packets:

- **Sender Report (SR):** contains the total number of packets and bytes sent as well as information that can be used to synchronize media streams from different sessions.
- **Receiver Report (RR):** contains information about the number of packets lost, the highest sequence number received, and a timestamp that can be used to estimate the roundtrip delay between a sender and the receiver.
- **Source Description (SDES):** contains the canonical name (CNAME) that identifies the source. It is used to be send to session participants.
- **End of participation (BYE):** it allows an end-point to announce that is leaving the conference.
- **Application-specific message (APP):** it provides a mechanism to design application-specific extensions to the RTCP protocol.

RTCP packets are sent as a compound packet that contains at least two of this types: a report packet and a source description packet.

3.4 Summary

The Real-Time Transport Protocol (RTP) provides end-to-end delivery services for the transmission of real-time data, independently of the transport-protocol. Even-though RTP was initially thought for application in audio and video conferences, this protocol can be applied to convey other types of streamed media across the network.

Chapter 4

Technology

4.1 Introduction

This chapter discusses both Java technologies used to develop the SIP VoIP Adapter, JAIN-SIP and Java Media Frameworks (JMF). It pretends introduce the main characteristics in its APIs architectures, to be able to understand the class design and its functionality, of the SIP VoIP Adapter.

4.2 JAIN-SIP

4.2.1 Overview

The Java APIs for Integrated Networks (JAIN) is an activity within the Java Community Process (JSP), that develops API's for telephony services. JAIN-SIP is a Java Application Programming Interface (API) and is a product of the Advanced Networking Technologies Division at the National Institute of Standards and Technology (NIST), to implement the Session Initial Protocol (SIP) for desktop and server applications. JAIN SIP enables transaction stateless, transaction stateful and dialog stateful control over the protocol. JAIN-SIP supports the SIP protocol functionality described in the RFC 3261 Specification and it also supports several SIP RFCs.

The SDP library provide support for the Session Description Protocol (SDP). It contains all the interfaces needed to encode and decode SDP messages.

4.2.2 JAIN-SIP Architecture

JAIN-SIP architecture is intended to support the SIP protocol in a variety of applications. It can be used in SIP Proxy Servers as well as in SIP User Agents.

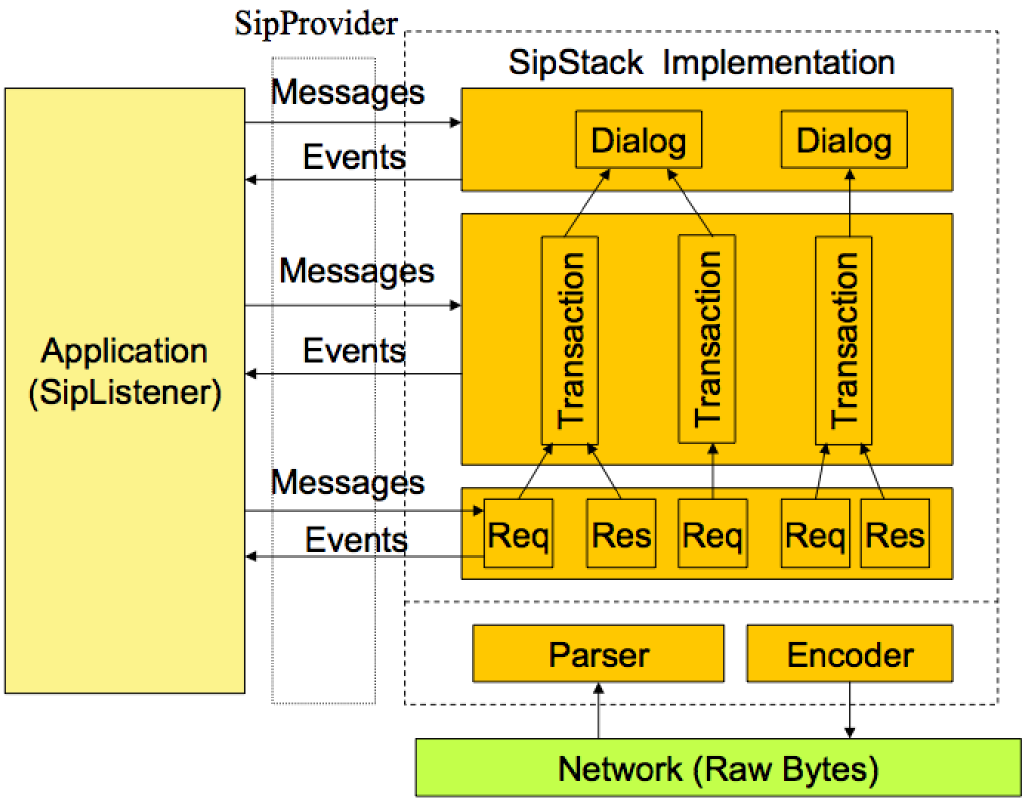


Figure 4.1: JAIN-SIP Architecture

4.2.3 JAIN-SIP Stack

This section discusses the different layers in the JAIN-SIP stack and how they interact to each other as Figure 4.2 shows.

The JAIN-SIP stack is formed by the following components:

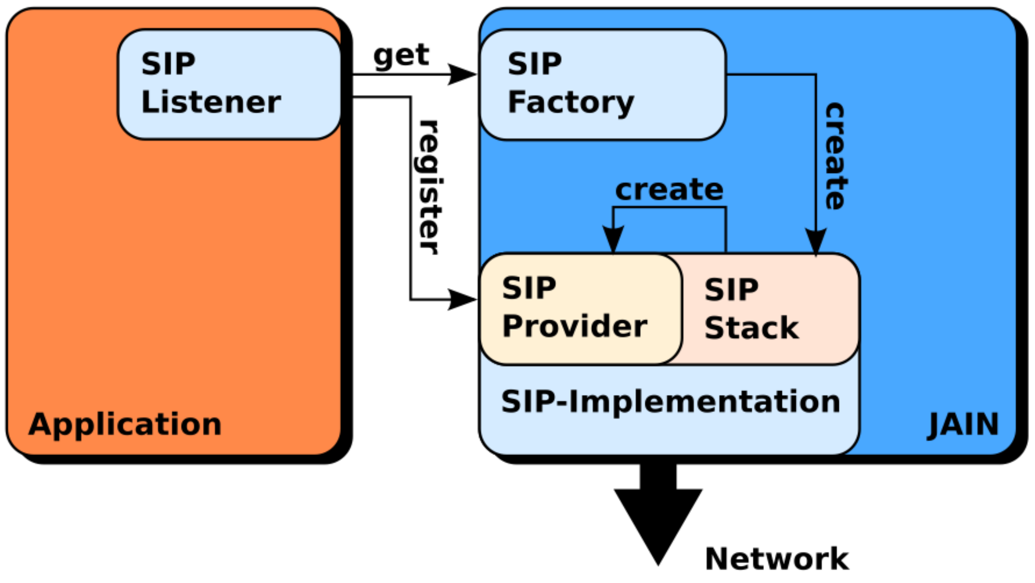


Figure 4.2: JAIN-SIP Application Architecture

SipListener

This interface defines the application's communication channel to the SIP stack. It represents the event consumer and listen for incoming Events that encapsulate messages that may be responses to initialize dialogs or new incoming dialogs.

The events accepted by an application are:

- **RequestEvent:** this event is generated when the SipProvider interface emits a request message.
- **ResponseEvent:** this event is generated when the SipProvider interface emits a response message.
- **TimeoutEvent:** this event is generated when the SipProvider interface notifies a time out.
- **IOExceptionEvent:** this event is generated when the SipProvider interface notifies an input/output exception.
- **TransactionTerminatedEvent:** this event is generated when the SipProvider interface notifies a transaction terminated.
- **DialogTerminatedEvent:** this event is generated when the SipProvider interface notifies a dialog terminated.

SipProvider

This interface represents the event provider, it receives messages from the network and passes them to the application as event.

ListeningPoint

This interface implement a Java representation of the socket that a SipProvider messaging entity uses to send and receive messages.

JAIN-SIP stack

It defines the management and architectural view of the SIP stack and it contains the following components:

- **Dialog:** it represents a peer-to-peer SIP relationship between communications SIP end points.
- **Transaction:** it specifies a request send by a client transaction to a server transaction, along with all responses to that request sent from the server transaction back to the client transaction.
- **Request:** it represents a message sent from the client to the server.
- **Response:** it represents a message sent from the server to the client.
- **Parser:** it represents an interface that accept raw bytes from the network, parse it according to the SIP grammar specified in the RFC 326, and converts to a format the application can interpret and process.
- **Encoder:** unlike the Parser, this interface represent an interface that encodes the messages from the application to raw bytes and sends them over the network.

Network

It contains raw bytes that are sent to the JAIN-SIP stack to be processed.

4.2.4 JAIN-SIP Objects

The JAIN-SIP specifications includes the following main objects:

SipStack Object

It defines the management and architectural view of the SIP stack. It contains methods to represent and modify the properties of the SIP stack. Only one SipStack can be associated with an IP address. However, an application can have multiple SipStack objects. SipStack is instantiated by the SipFactory interface and it must be initialized with a set of configuration properties specifies in the Annex, section A.

Dialog Object

This object facilities sequencing of messages between communicating end points. Dialog are not directly created by the Application, they are established by Dialog creating Transaction as INVITE, SUBSCRIBE, etc. however they are managed by the stack.

A Dialog contains the following data for further message transmission within the dialog:

- **Dialog ID:** it identifies the dialog.
- **Local Sequence Number:** it defines the counter of orders requests from the user to its peer.
- **Remote Sequence Number:** it defines the counter of orders requests from its peer to the user agent.
- **Local URI:** it specifies the address of local party.
- **Remote URI:** it specifies the address of remote party.
- **Remote Target:** it specifies the address from the contact header field of the request or response, or refresh request or responses.
- **Secure:** it determines whether the dialog is secure.
- **Route Set:** it specifies an ordered list of URIs.

Transaction Object

It represents a generic transaction interface each defines the methods that are common to client and server transactions. The Transaction consists of a single request and any responses to that request. User agents and stateful proxies contain a transaction layer, whereas stateless proxies do not contain a transaction layer.

Message Object

It represents a request from a client to a server, or a response from a server to a client. Both types of messages consist of a method name, address, and protocol version, one or more header fields which describes the message routing, and an optional message-body.

4.2.5 Factories

A factory interface provides methods that enable an application to create the specific factory objects.

JAIN-SIP defines four different types:

- **SipFactory:** this interface defines methods to create new Stack objects as well as other factory objects.
- **AddressFactory:** this interface defines methods to create URIs for SIP and Tel URIs.
- **HeaderFactory:** this interface defines methods to create any kind of supported Header object.
- **MessageFactory:** this interface defines methods to create new Request and Response message objects.

4.2.6 SDP Library

JAIN SDP is a very simple API that allows us to encode and decode SDP content. It is also based on a factory pattern, therefore the creation methods uses SdpFactory. JAIN-SDP also provides a number of interface classes that represent the key concepts in the Session Description Protocol. These interfaces breack down into two categories. On one hand, there is the SessionDescription interface, which models the SDP message itself; on the other, there are set of interfaces, each of them representing the SDP body messages.

4.2.7 Summary

The JAIN-SIP architecture, as well as the JAIN-SDP library, provides support for the developing of IP telephony services to be added to applications built on Java. These APIs provide a standard portable interface to share information between SIP clients and SIP servers. In addition it offers several benefits as standardizes the interface to the stack, as well as the message interface, and the event semantic. JAIN-SIP and JAIN-SIP also provides application portability that enables interoperability of applications across stacks.

4.3 Java Media Frameworks (JMF)

4.3.1 Overview

Java Media Framework (JMF) is a Java Application Programming Interface (API) developed by Sun Microsystems in partnership with other companies to enable audio, video and other time-based media to be added to applications and applets built on Java technology. It provides a unified architecture and messaging protocol for managing the acquisition, processing, and delivery of time-based media.

4.3.2 JMF Architecture

Java Media Frameworks (JMF) architecture in intended to support multimedia in a variety of applications. It provides the high-level API, called JMF Presentation and Processing API, that contains generic classes for managing the capture, presentation , and processing of the time-based media. And the low-level API, called JMF Plug-in API that supports the seamless integration of custom processing components and extensions.

JMF 2.1.1 uses the following basic interfaces to model the high-level API:

- **MediaLocator:** describes the location of a media content. MediaLocator is similar to a URL, and can be constructed from a URL.

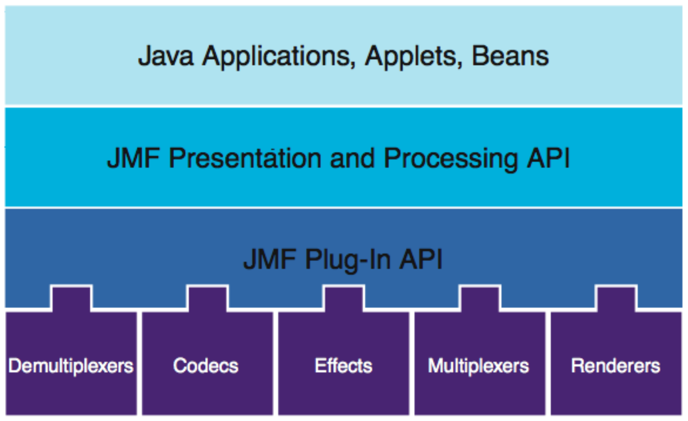

Java Applications, Applets, Beans
JMF Presentation and Processing API
JMF Plug-In API
Demultiplexers
Codecs
Effects
Multiplexers
Renderers

Figure 4.3: JMF Architecture

- **DataSource:** represents the media itself. DataSource encapsulates both the location of media and the protocol and software used to deliver the media. A DataSource manage a set of SourceStreams objects.

  JMF defines several types of DataSource objects that can be categorized according to how data transfer is initiated:

  - **Pull Data-Source:** the client initiates the data transfer and controls the data flow from pull data-sources, so it is the client who delivers the data as requested. JMF defines two types of pull data-sources: PullDataSource and PullBufferDataSource, which uses a Buffer object as its unit of transfer.
  - **Push Data-Source:** the server initiates the data transfer and controls the data flow, so it is the DataSource who is in charge of when data is delivered. JMF defines two types of push data-sources: PushDataSource and PushBufferDataSource, which uses a Buffer object as its unit of transfer.

- **DataSink:** represents an output device other than a representation device. A DataSink is used to read media data from DataSource and render the media to some destination.

- **Player:** reads data from a DataSource and provides processing and control mechanisms over a steam of data.

- **Processor:** is a specialized type of Player that can provide control over how the data is processed before it is presented. A Processor takes a DataSource as input, performs some user-defined processing on the media data, and then outputs the processed media data. A Processor can also be used to present media data to presentation devices or output media data through a DataSource, so that it can be presented by another Player or Processor.

- **Manager:** handles the constructions of Players, Processors, DataSources, and DataSinks. These objects are always created the same way whether the requested object is constructed from a default implementation or a custom one built by the user.

Players

A Player processes an input stream of media data delivered from a DataSource, and renders it to a specific destination depending on the type of media being presented. A Player does not provide any control over the processing that it performs, that's a Processor's work.

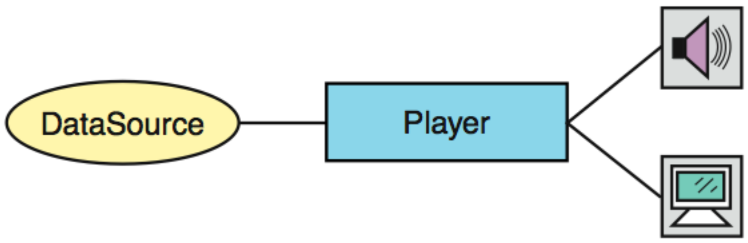


Figure 4.4: JMF Player Model

A Player can be in one of six states defined by Clock and Controller interfaces. The Clock interface defines two states: Stopped and Started. In addition, Controller breaks the Sopped state down into five: Unrealized, Realizing, Realized, Prefetching, and Prefetched. A Player posts TransitionEvents as it moves from one state to another, and it is the ControllerListener interface who provides a way to determine what state a Player is in to respond appropriately.

Processors

A Processor is just a type of Player that provides control over what processing is performed on the input media. In addition, it can output media data through a DataSource so that it can be presented by a Player, or it can also be further manipulated by another Processor.

A Processor allows the application developer to perform specific processing on the media data as effects, mixing, and compositing in real-time.

For each processing operation there is a dedicated piece of software called "plug-in", there are five components of JMF plug-ins that are applied to the media data:

- **Demultiplexer:** extract individual tracks of media from a multiplexed media stream.

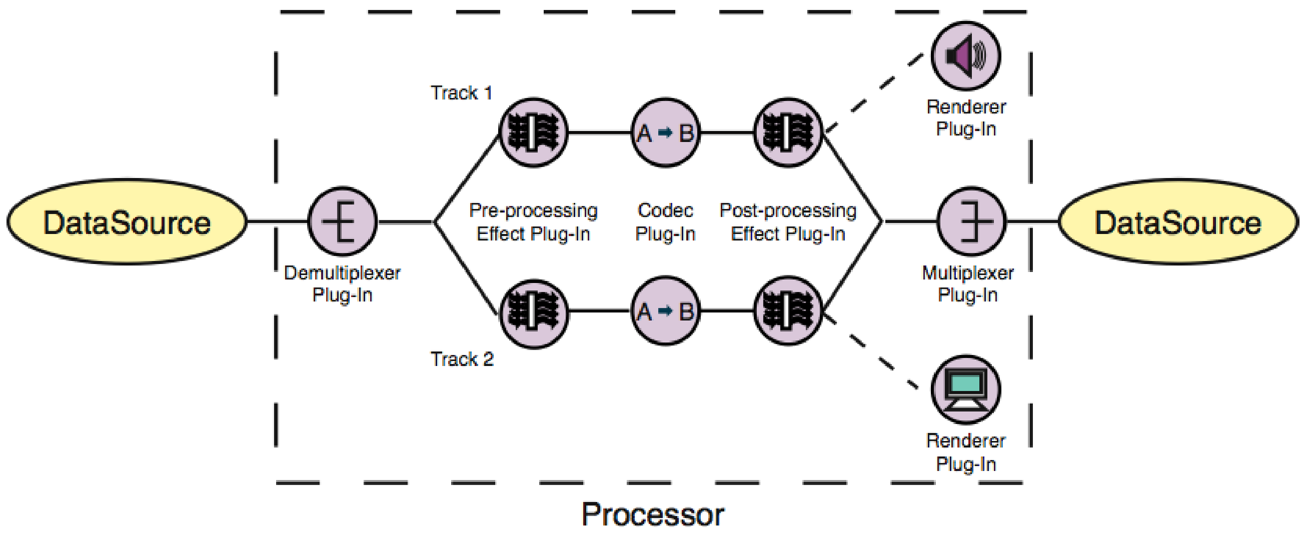


Figure 4.5: JMF Processor stages

- **Effect:** modify the track data in some way, often creating some special effect.
- **Codec:** perform media data encoding and decoding.
- **Multiplexer:** join individual tracks into a single stream of data.
- **Renderer:** delivers the media data in a track to presentation device depending on the type of data, as video or audio.

Processor, as a Player, can be in several states. In this case, there are two additional standby states which occur before the Processor enters the *Realizing* state, Configuring and Configured.

4.3.3 JMF RTP API

JMF enables the playback and transmission of RTP streams through the APIs defined in the javax.media.rtp.event, and javax.media.rtp.rtcp packages. The RTP APIs can be used to transmit captured or stored media streams across the network which can be originated from a file or a capture device.

JMF RTP Architecture

The JMF RTP API are designed to work seamlessly with the capture, presentation, and processing capabilities of JMF. RTP media streams can be manipulated and presented by players and processors. It is possible to transmit media streams that have been captured from a local capture device using a capture DataSource.

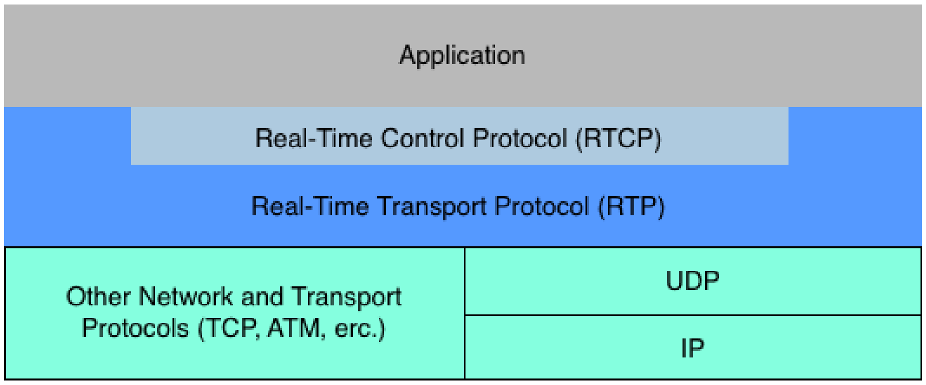

Application
Real-Time Control Protocol (RTCP)
Real-Time Transport Protocol (RTP)
Other Network and Transport Protocols (TCP, ATM, erc.)
UDP
IP

Figure 4.6: JMF RTP API

Session Manager

A SessionManager interface is used to coordinate an RTP, it defines methods that enable an application to initialize and start participating in a session. The session manager keeps track of the session participants and the streams that are being transmitted. A session manager is a local representation of the RTP session. In addition, it also handles the RTCP control channel, and supports RTCP for both transmitters and receivers.

- **Session Statics:** The session manager maintains statics on all of RTP and RTCP packets transmit and received in the session.
- **Session Participants:** The session manager keeps track of all of the participants in the session. SessionManager create a Participant, represented by the instance class that implements, whenever an RTCP packet arrives that contains a source description (SDES) with a canonical name (CNAME) that has not been seen before in the session. A participants can own mote than one stream, each of which is identified by the synchronization source identifier (SSRC) used by the source of stream.
- **Session Streams:** The SessionManager maintains an RTPStream object for each stream of RTP data packets in the session.

  There are two types of RTP streams:

  - **ReceiveStream:** it represents a stream that is being received from a remote participant. It is created automatically whenever a session manager detects a new source of RTP data.
  - **SendStream:** it represents a stream of data coming from the Processor or input DataSource that is being send over the network. It is created by the SessionManager method createSendStream.

RTP Events

The RTP-specific events are used to report on the state of the RTP session and streams. To receive their notifications, it is necessary to implement the appropriate RTP listener and register it with the session manager.

There are several listeners interfaces that lets us to handle event notifications:

- **SessionListener:** It receives notifications of changes in the state of the session, such as the addition of new participants.
- **SendStreamListener:** It receives notifications of changes in the state of an RTP streams that's being transmitted.
- **ReceiveStreamListener:** It receive notifications whenever a new receive streams are created or the transfer of data has been start or stop, when an RTCP packet is received or the receive stream's format or payload has changed. This interface also provide handling on the stream and get access to the RTP DataSource to create a MediaHandler.
- **RemoteListener:** It receive notifications related to RTCP control messages received from remote participants. This interface is used to monitor the session, enabling the RTCP reception without having to receive data on each stream.

RTP Data

The streams within an RTP session are represented by RTPStream objects. There are two sub-objects of RTPStream: ReceiveStream and SendStream. All of them has a buffer source associated with it, and for the ReceiveStream, this DataSource is always a PushBufferDataSource.

SessionManager creates ReceiveStream objects automatically when a new receiving streams are detected, and it also creates SendStream objects when the method *createSendStream()* is called.

Reception

JMF Players and Processors provide the presentation, capture, and data conversion mechanism for RTP streams.

JMF also provide two ways to construct a Player:

- **MediaLocator:** it has the parameters of the RTP session and it can only present the first RTP stream that's detected in the session. Construct a Player is possible by calling *Manager.createPlayer(MediaLocator)*.
- **SessionManager:** it is necessary to play back multiple RTP streams in a session. SessionManager gets construct a Player for each ReceiveStream. Construct a Pleyer for a particular ReceiveStream is possible by retrieving the DataSource from the stream and passing it to *Manager.createPlayer(DataSource)*.

To be able receiving RTP streams, first a SessionManager object has to be created, initialized by calling the method *initSession()*, and started by calling the method *startSession()*. It is also necessary to register a listener by calling the method *addReceiveStreamListener()*.

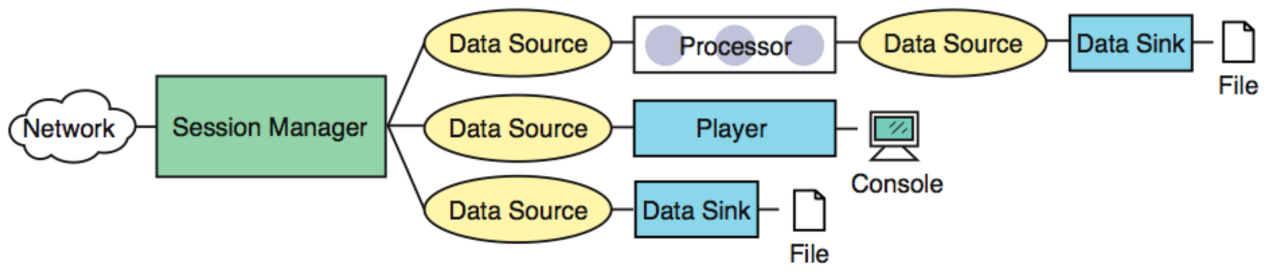


Figure 4.7: JMF RTP reception

When a SessionManager has been started, any receiving stream will generate a NewReceiveStreamEvent. So the update method in the ReceiveStreamListener interface can detect this event and obtain the ReceiveStream object by calling *getReceiveStream()*. By retrieving the DataSource from the ReceiveStream object and passing it to the Manager, it can construct a Player or Processor by calling *Manager.createPlayer()*, and *Manager.createProcessor()*.

For presenting the RTP media streams, Sun supplies the class PlayerWindow, which receives a Player, plays back the media, and creates a control window (which is also provided by Sun as PlayerPanel).

Transmission

A session manager can also be used to initialize and control a RTP session to stream data across the network. This data to be streamed is acquired from the output of a Processor by retrieving a DataSource object and passing it to the session manager using the method *createSendStream()*. The Processor has to generate RTP-specific format data because Session Manager does not perform this action. The method *setFormat()* of the TrackControl interface allows a Processor to set the format of a track. The send stream methods as *start()* and *stop()*, are used to control the transmission.

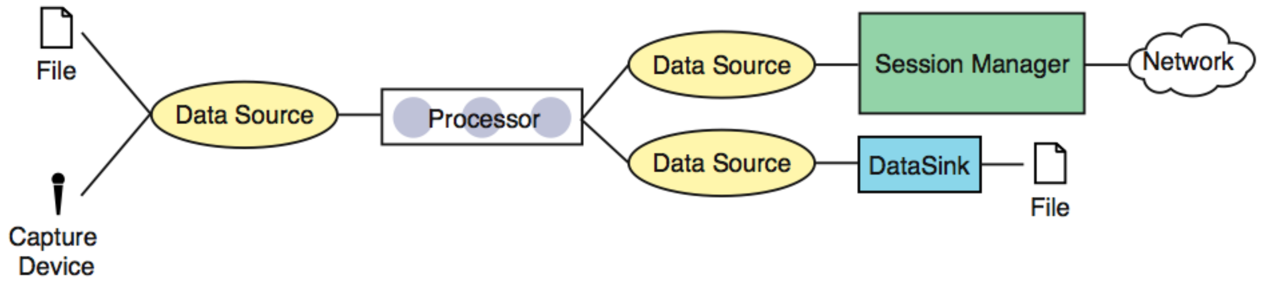


Figure 4.8: JMF RTP transmission

If the data to be streamed is intended to be a physical source as a microphone, a CaptureDeviceInfo object has to be created by calling the method getDevice of CaptureDeviceManager interface. This interface searches for a device in the system, allowing the CaptureDeviceInfo returns a MediaLocator when the method getLocator is called.

4.3.4 Summary

The Java Media Frameworks (JMF) architecture provide support for capturing and storing media data, controlling the type of processing, and performing custom processing on media data streams. It is based in a high-level API, containing the generic interfaces to capture and present the media, and a low-level API, to allow customization and extension. The RTP API support RTP transmission and reception.

4.4 Summary

The main Java APIs have been described to help for going deeper into the Technology used in the implementation of SIP VoIP Adapter. This chapter is intended as a guide to help the reader understand the SIP VoIP Adapter class design as well as the description of each class that makes it up.

Chapter 5

Design and Implementation of SIP VoIP Adapter

5.1 Introduction

This chapter covers the class design and functionality of the application developed as the main goal of this thesis, called SIP VoIP Adapter. SIP VoIP Adapter is a Java application that works as a Sip User Agent. And, on one hand, it handles the Media Data received from a RTP and RTCP communication, established with another User Agent, over a specific Personal Area Network (PAN) interface keeping off the transport level protocol to work only with the transmission protocol UDP. On the other hand, it handles the media data received from a UDP connection and transmits the data over a RTP and RTCP communication to the other User Agent. With the aim to communicate an End Device, situated on the PAN side, as the Media Capture and Player Device of the SIP VoIP Adapter.

5.2 Requeriments

The SIP VoIP Adapter has been developed in Java, using the JAIN-SIP and JAIN-SDP API's as the main technology used for the SIP signaling, and the Java Media Frameworks (JMF) API for the audio treatment and transmission.

The communication between the SIP VoIP Adapter and Asterisk PBX has been developed using both IPv4 and IPv6. The audio data transmission between both end clients registered on the PBX, uses the RTP and RTCP protocol, also providing most of the main audio codifications as ULAW, GSM, etc. The communication between the SIP VoIP Adapter and the sound end device (6lowPAN device provided with a speaker and microphone) uses the User Datagram Protocol (UDP) as the transmission protocol. In addition SIP VoIP Adapter provides an independent audio codification for this transmission.

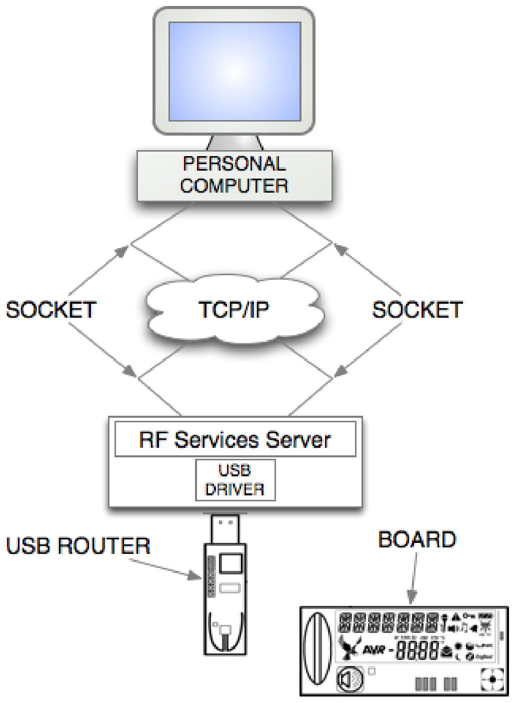


Figure 5.1: 6lowPAN system overview

The SIP VoIP Adapter communication through the end sound device is designed over UDP (without using the RTP protocol) due to the architecture of the intended devices to be used as a sound device: the AVR RZ Raven 2.4 GHz Wireless Evaluation Kit, which is composed by the USB router and the Board Device.
The SIP VoIP Adapter must transmit the audio received from the RTP communication to the USB router through an UDP socket, as well as it must receive the audio data from an UDP socket and transmit to the RTP communication. Thus, the USB Router connected on the same computer where the SIP VoIP Adapter is running, will be the responsible of handling the 6lowPAN wireless communication between him and the 6lowPAN Board Device. Figure 5.1 shows the communication between the computer and the 6lowPAN USB Router.

As Figure 5.2 shows, the SIP VoIP Adapter application has been designed in independent layers: The SIP signaling layer as been designed independently of the audio media treatment layer. The transmission between the sound device and the SIP VoIP Adapter has been also abstracted to the RTP/RTCP transmission between the Registrar Server (Asterisk) and the SIP VoIP Adapter to let adapt another end devices for a future work.

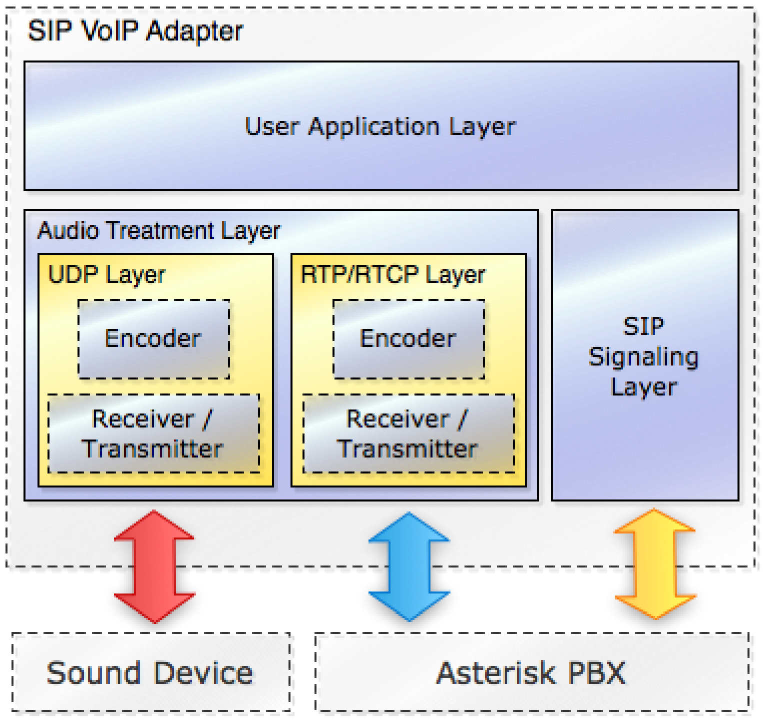


Figure 5.2: SIP VoIP Adapter application layers

5.3 SIP VoIP Adapter Design

The SIP VoIP Adapter design stage has been broken down in two: SIP components design and Media components design. The first design stage discusses the requirements needed for the Application to establish a SIP communication through a Proxy Registrar to another client also connected on the Proxy. Moreover, the second design stage discusses the management of the media being transmitted between clients.

5.3.1 SIP Components Design

First of all, it is necessary to understand the SIP signaling communication needed to register the application on the Proxy Register, so then, be able to establish a communication with another client for the exchange of media data. As we have seen in section 4.2.3, to create a SIP communication channel, the interface SipListener, provided by the JAIN-SIP API, must be implemented. This interface must create, configure and start the SIP Stack to allow the application talk with the Proxy. As well as implement the listener methods to control responses, requests and timeout messages received from the Proxy.

The response messages the application needs to care about to manage a communication through a Registrar Server are: OK 200, TRYING 100, RINGING 180, TEMPORY UNAVAILABLE, PROXY AUTHENTICATION REQUIRED 407, UNAUTHORIZED 401. An OK 200 message can be received as a response of several requests, so it is necessary to find from which request is associated to be processed. It can be REGISTER, INVITE, BYE, CANCEL, or MESSAGE request. PROXY AUTHENTICATION REQUIRED 407 and UNAUTHORIZED 40 can be processed together.

The request messages the application needs to care about to manage a communication through a Registrar Server are: INVITE, ACK, BYE, CANCEL, MESSAGE. The timeout messages the application needs to care about to manage a communication through a Registrar Server are if they come from a REGISTER or INVITE.

Figure 5.3 and Figure 5.4 shows the different states the application can be.

Another interface is needed to process the SIP messages handled by the SipListener interface, in order to answer according to the application's state. This work is made it by the MessageProcessor class. It manage responses, requests and timeout types chosen by the SipListener and process it to decide what the application needs to do. And it depends on the two clearly different actions the application does: register and call. So, it is necessary to introduce the Register Status class, that defines in which register state the application is, and the Call Status (defined in the Audio Call's implementation), that defines in which call

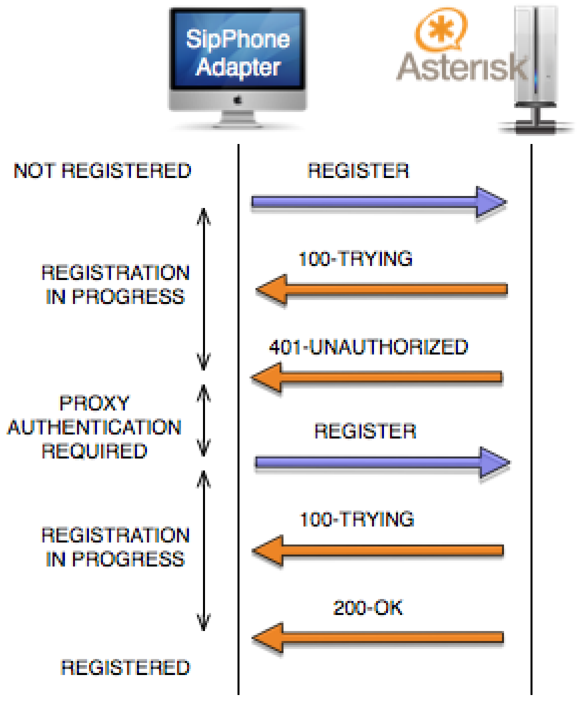


Figure 5.3: Register Status in a registration process

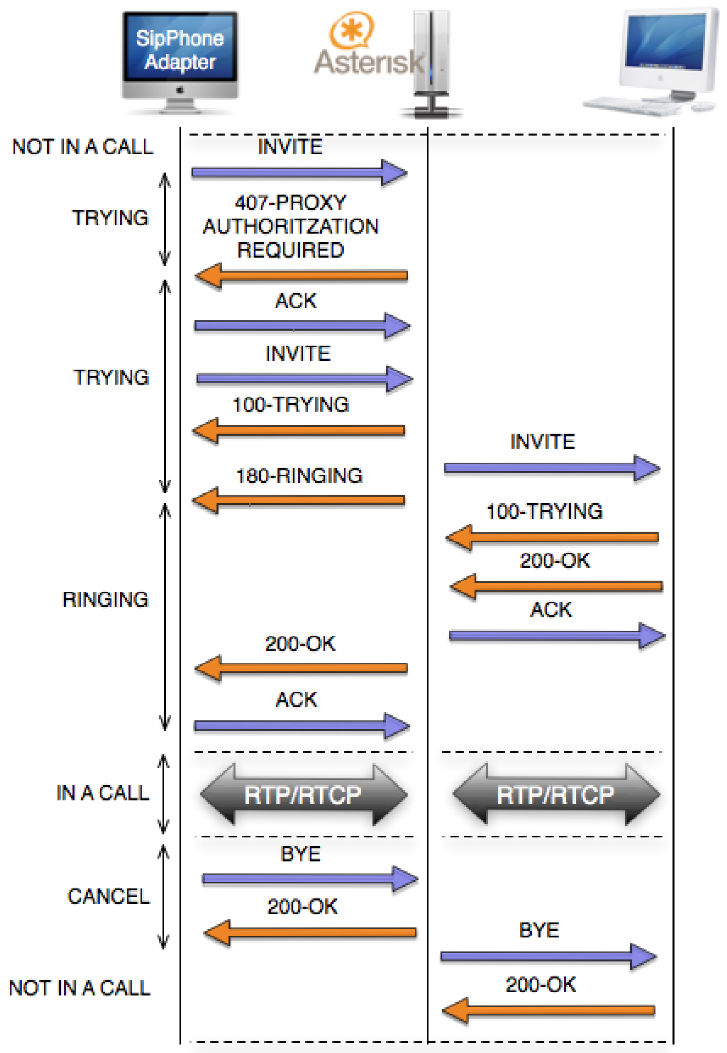


Figure 5.4: Call Status in a call process

state the application is.

Now, that we are able to receive SIP messages from the Proxy, and answer it according to the phone's state. It is necessary to build an interface for controlling this changes by updating the GUI's view. So, this interface called PhoneController, has to update the view of the GUI when one of the two status, register or call, change.

Finally, it is necessary to build an interface for controlling the application actions the user can do for registering or making a call. This interface is called SIPManager, and its methods allows the SIP VoIP Adapter's user transmit SIP requests, as INVITE and REGISTER among others. It also manage both status, register and call, and change their state according to the user actions, as register the application in the Proxy or make a call to another client also registered.

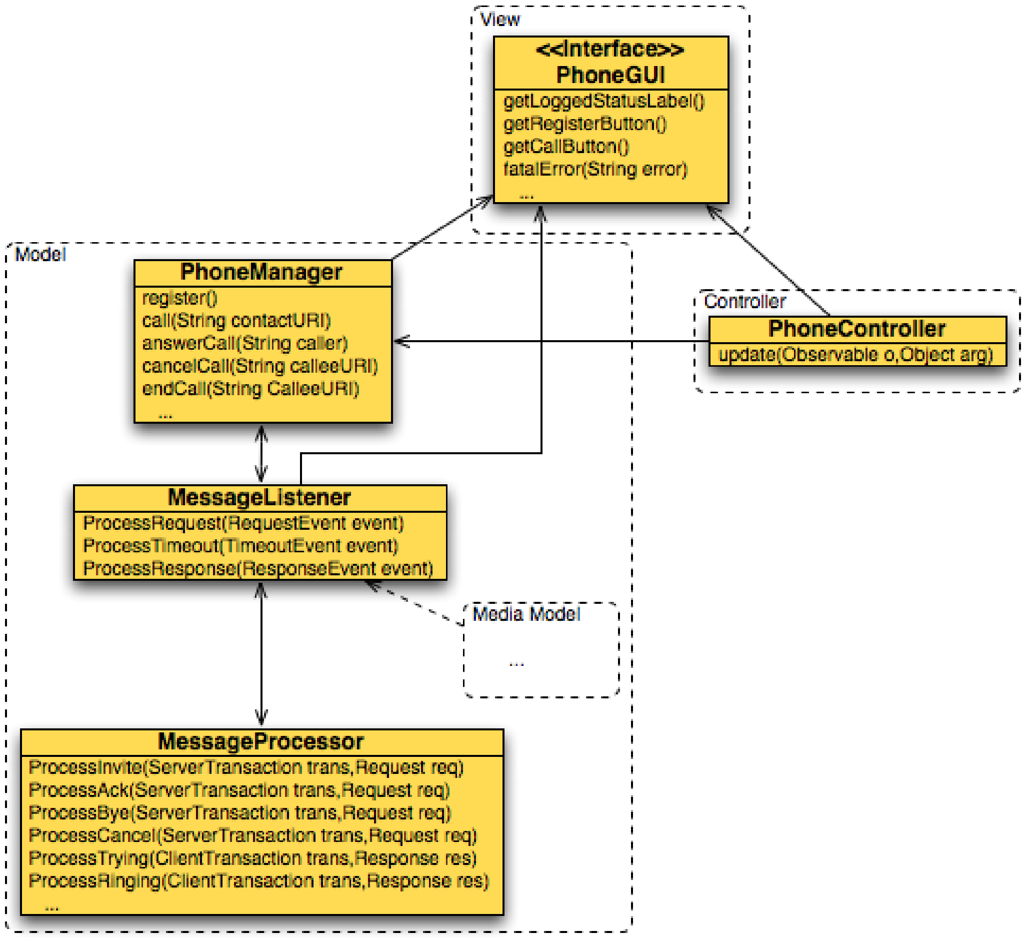

View
<<Interface>>
PhoneGUI
getLoggedStatusLabel()
getRegisterButton()
getCallButton()
fatalError(String error)
...
Model
PhoneManager
register()
call(String contactURI)
answerCall(String caller)
cancelCall(String calleeURI)
endCall(String CalleeURI)
...
Controller
PhoneController
update(Observable o,Object arg)
MessageListener
ProcessRequest(RequestEvent event)
ProcessTimeout(TimeoutEvent event)
ProcessResponse(ResponseEvent event)
Media Model
...
MessageProcessor
ProcessInvite(ServerTransaction trans,Request req)
ProcessAck(ServerTransaction trans,Request req)
ProcessBye(ServerTransaction trans,Request req)
ProcessCancel(ServerTransaction trans,Request req)
ProcessTrying(ClientTransaction trans,Response res)
ProcessRinging(ClientTransaction trans,Response res)
...

Figure 5.5: SIP VoIP Adapter Class Diagram - MVC pettern

5.3.2 Media Components Design

Once the SIP signaling exchange has designed, another interface is needed to handle the media transmission between users corresponding to a call. This interface is called MediaManager, besides opening the media reception and transmission, and closing them, it has another functions as detect the supported codecs by the sound devices connected to the computer where the application is running, as well as manage the media session information contained in the SDP messages exchanged by the users. This class also handles the media transmission between the SIP VoIP Adapter and the PAN interface connected in the same computer where the application is running. It has been designed using the Java Media Frameworks (JMF) API and the JAIN-SDP library.

Moreover, two other classes are needed to allow the application receive and transmit data over the RTP communication configured from the RTP parameters exchanged by the users in the SIP signaling. They are called Receiver and Transmitter.

The Receiver class has to initialize the RTP Manager interface, provided by the JMF API, to create the RTP channel for reception, and wait until an incoming stream arrives. Once the stream has arrived, it creates a DataSource from the incoming stream and parses the DataSource to create a Player.

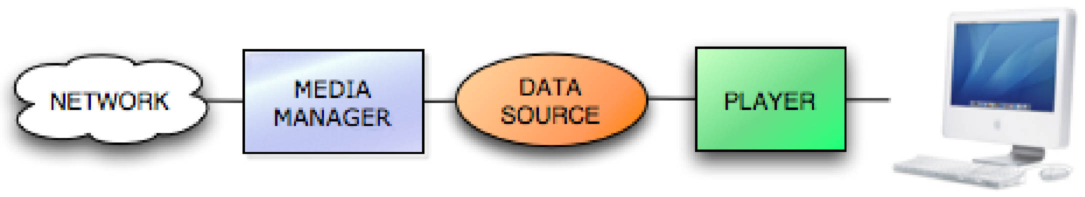


Figure 5.6: RTP reception

The Transmitter class has to create a MediaLocator to get the devices connected on the computer to capture the audio from it. Then, it has to create a DataSource from that MediaLocator to be parsed in the Processor's creation, configure the media with the appropriate parameters, and gets its data output. Finally, the Transmitter has to create the RTP channel for transmitting by initializing the RTP manager interface.

Another classes are needed to allow the application receive audio data over the RTP communication and transmit it to another network interface using UDP, as well as, receive audio data from the UDP network interface and transmit it over the RTP communication. They are called RTPtoUDP and UDPtoRTP.

Moreover, to implement both classes first we need to create a custom DataSource to read

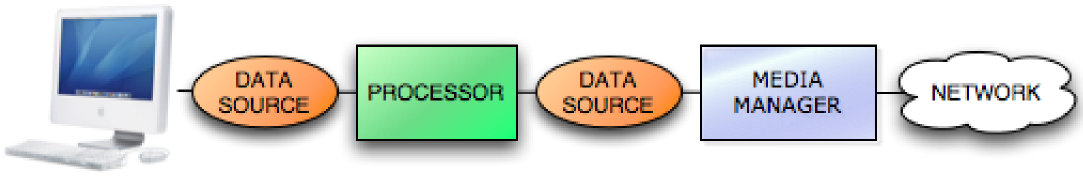


Figure 5.7: RTP transmission

media data from a UDP port, and a custom DataSink to transmit media data over a UDP port. On the first hand, two classes, UDPDataSource, and UDPSourceStream, has been created to lets a Processor handle an incoming streams acquired from a UDP connection. And on the other hand, DataSourceHandler class has been created to implement a DataSink to transmit the data output of a Processor, over a UDP connection.

For both classes, Receiver and Transmitter, the RTP audio transmission is able to codify the audio data with the following codecs:

- G.711 μ-law
- GSM
- G.723
- DVI

The RTPtoUDP class has to initialize the RTP Manager interface to create the RTP channel for reception, and wait until an incoming stream arrives. Once the stream has arrived, it creates a DataSource from the incoming stream and parses the DataSource to create a Processor. It configures the media with the appropriate parameters, and gets its data output to be parsed to the DataSink. Finally, the DataSink use the DataSourceHandler interface to transmit the media data over the UDP connection.

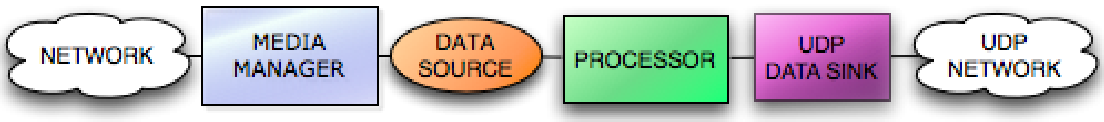


Figure 5.8: RTP reception - UDP transmission

The UDPtoRTP class has to create a Processor from the UDPDataSource and wait until the media data arrives from the UDP connection. Once the data has arrived, it is necessary to configure the media with the appropriate parameters to be send over RTP, gets its data output, and finally, create the RTP channel for transmitting by initializing the RTP manager interface.

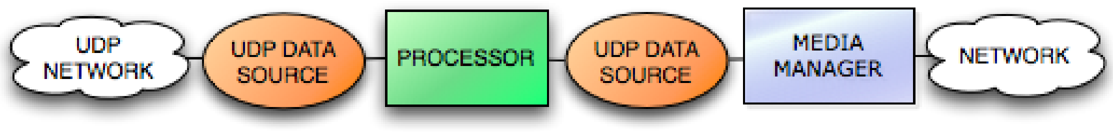


Figure 5.9: UDP reception - RTP transmission

For both classes, RTPtoUDP and UDPtoRTP, the audio data transmitted is able to be codified with the following formats:

- G.711 μ-law
- GSM
- G.723
- DVI ADPCM

Figure 5.10 shows the class diagram of the media components designed.

5.3.3 High Level View

Before start explaining in more detail the functionality and implementation of each class which conforms the application, it is interesting to understand how the main classes works together for the management of the application and its interaction between user and Registrar Server.

Figure 5.11 shows both, the inputs and the outputs the application has, and which class take care of each action, depending on who is generating that action.

The PhoneManager controls the input actions generated by the user and on the other hand, is the PhoneController class who manages the output actions of the application that must be shown on the GUI. The MessageListener handles both, inputs and outputs of the SIP messages exchange through the network where the SIP Registrar Server belong.

The classes that makes up the Sound Device are Receive.java and Transmit.java, and both

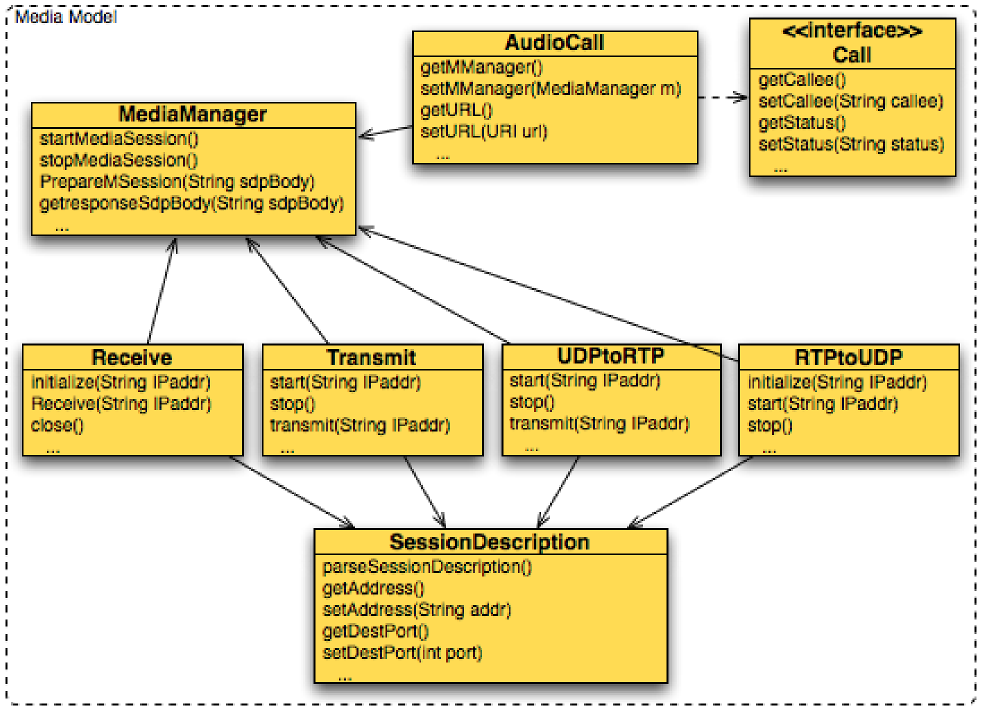


Figure 5.10: Media Model Class Diagram - MVC pattern

manage the audio data inputs and output between the audio computer device and the network governed by SIP Registrar Server. The classes that makes up the Adapter are RTPtoUDP.class and UDPtoRTP, and both handles the audio inputs and outputs between the UDP interface connected on the computer and the network governed by the Server. This classes are managed by the MediaManager.java.

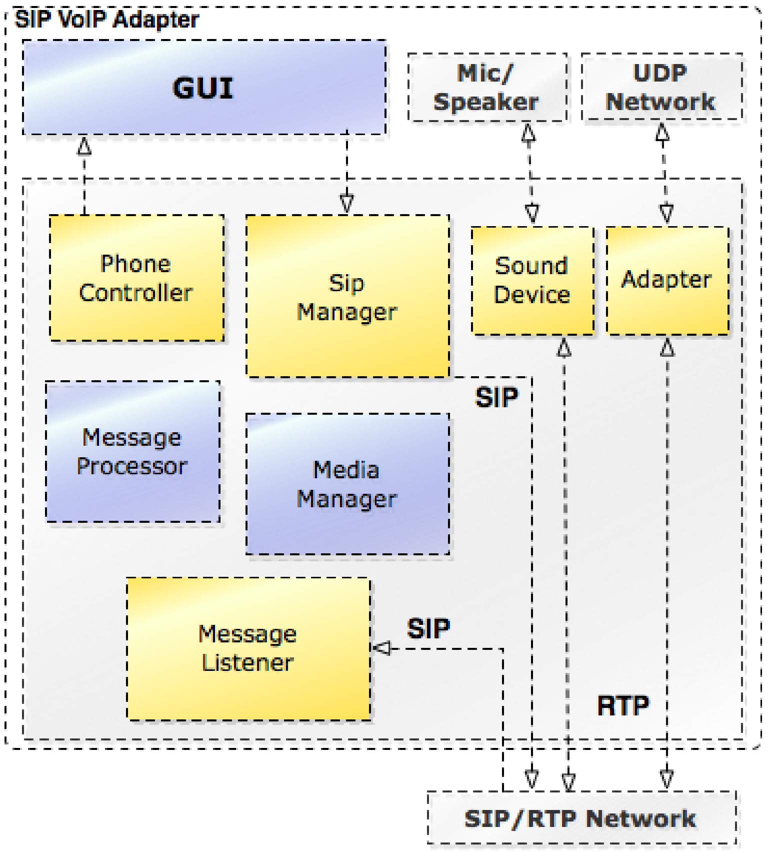


Figure 5.11: SIP VoIP Adapter inputs and outputs

5.4 SIP VoIP Adapter Functionality and Interface (High Level View)

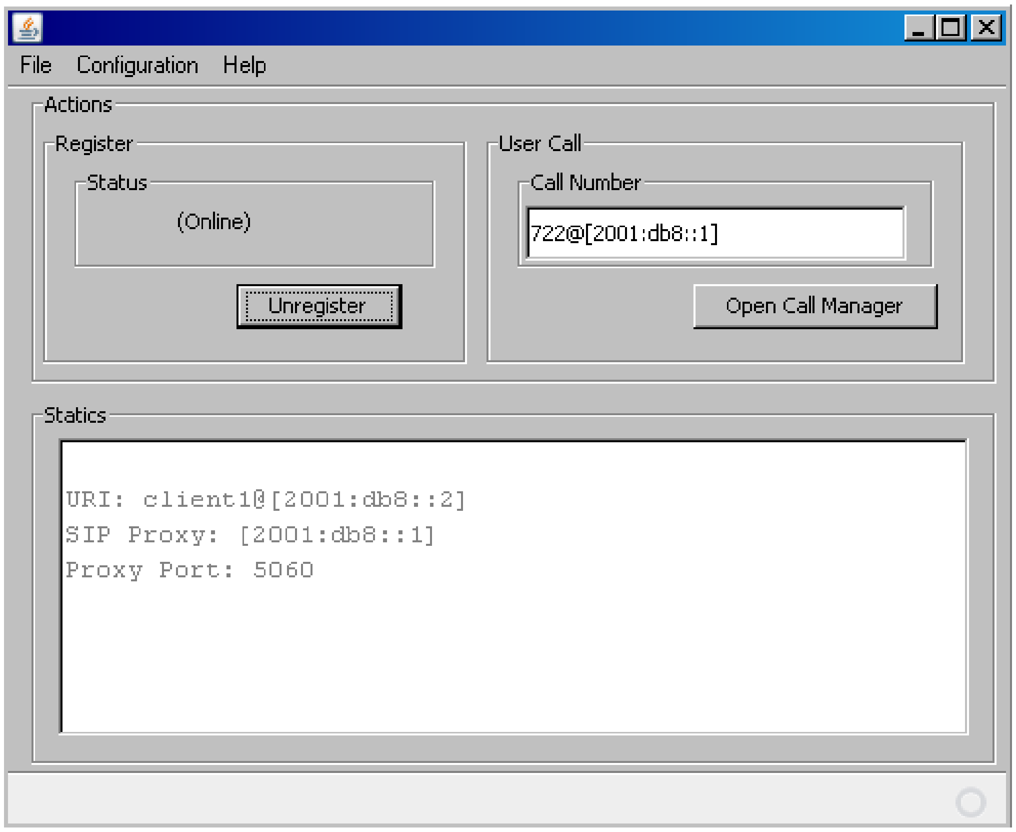
This section contains a high level view of the SIP VoIP Adapter in order to summarize itÕs functionality.

5.4.1 Graphical User Interface (GUI)

SIP VoIP Adapter main window (SIPVoIPView)

Figure 5.12 shows the SIP VoIP Adapter main window. In the "Configuration" menu the user can get access to the configuration window to specify the register and the Audio Capture and Player Device configurations. In the bottom of the application the user can see the user configuration realized. The button "Register" sends a Register Request to the Proxy Server Asterisk to establish Sip communication. Once the communication is established with the Proxy Server, the "Status" label "(Offline)" is going to change as "(Online)". Then, the user will be able to get access to the button "Open Call Manager" to call another Sip Phone registered in the Proxy Server Asterisk, which itÕs number is specified in the "Call Number" text field. In the "Help" menu the user can see the information about the SIP VoIP Adapter. In the "File" menu the user can select the option "Exit" to close the application.

This class implements the *Observer* interface and so, it has the method *update()* to connects the view to the model.



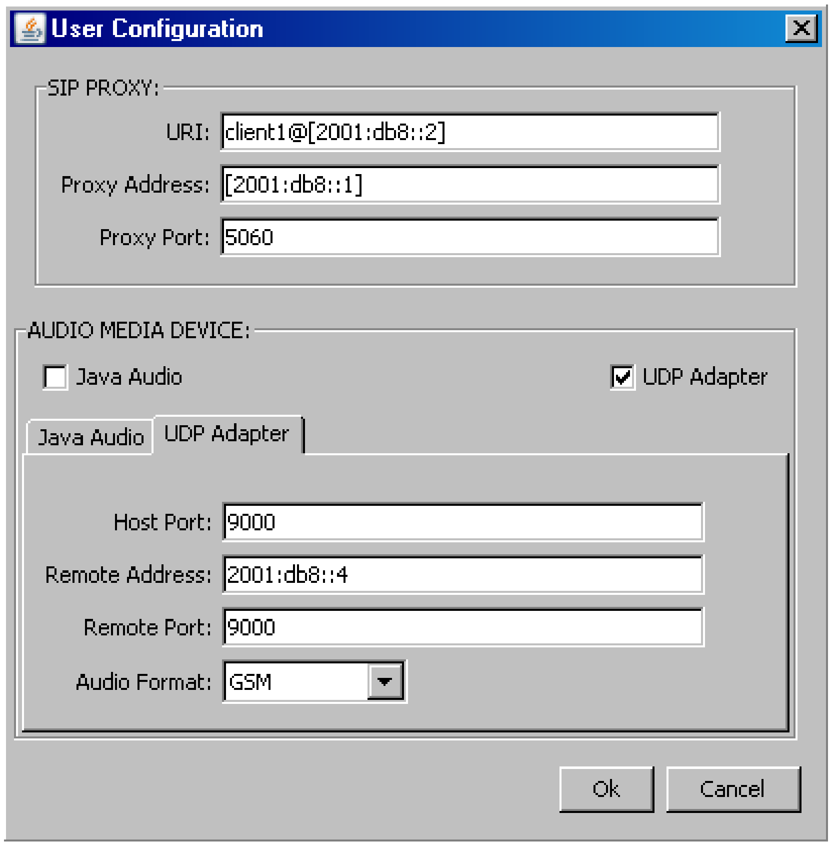
Call Number: 722@[2001:db8::1]

```
URI: client1@[2001:db8::2]
SIP Proxy: [2001:db8::1]
Proxy Port: 5060
```

Figure 5.12: SIP VoIP Adapter main window

Configuration window (ConfigDialog)

Figure 5.13 shows the Configuration window with the option UDP Adapter marked as Audio Media Device. The upper part of the window contains the SIP proxy configuration, where the user can introduce the Sip Proxy Settings to for connecting to Asterisk. The bottom part of the window contains the audio media device options, where the user can chose the Java Audio as a sound device or use the UDP Adapter connecting with another sound device in the PAN interface, introducing the port where to receive the audio data, the remote IP address and port where to send it, and the codec of this audio data to be transmitted.


User Configuration

SIP PROXY:

URI: client1@[2001:db8::2]

Proxy Address: [2001:db8::1]

Proxy Port: 5060

AUDIO MEDIA DEVICE:

[ ] Java Audio [✓] UDP Adapter

Host Port: 9000

Remote Address: 2001:db8::4

Remote Port: 9000

Audio Format: GSM

Figure 5.13: Adapter Configuration window

Call window (CallDialog)

Figure 5.14 shows the Call window. The user can get access to the Call window by two different ways, which in both, the user have to already be registered on the Proxy Server Asterisk. On the first case, the user press the button "Open Manager Call" in the SIP VoIP

Adapter main window, after to introduce the sip number to call in the "Call Number" text field. On the second case, a window is going to be opened asking if the user wants to receive a call from another user, also registered in the Proxy Server Asterisk, if the "yes" option button is pressed, a Call window is going to be opened establishing the communication between users. Once the Call window is opened, in both cases, the user can see all the communication statics in the text area, at the bottom of the window. The user can also press the button "Cancel Call" to finish the communication with the other user.

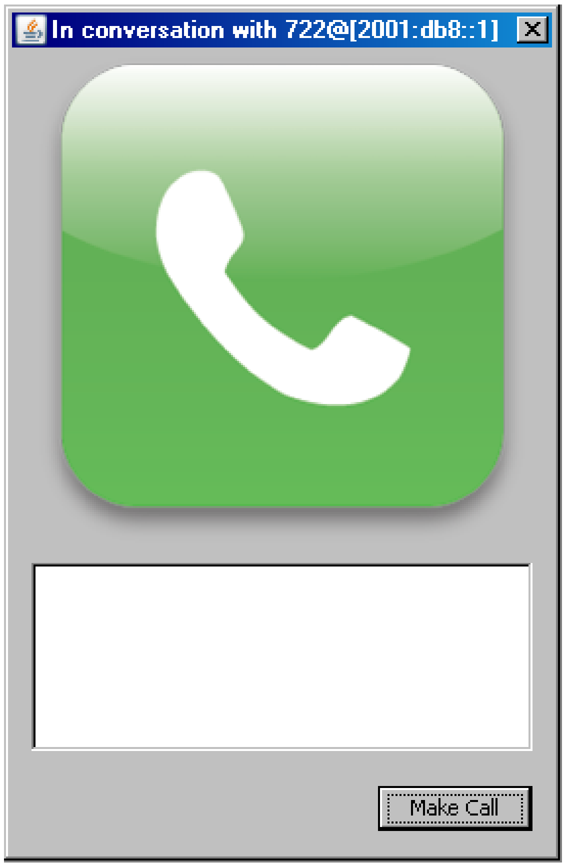


Figure 5.14: Audio Call window

Media Presentation

When a Call is established there are two options to be able to communicate with the other user, depending on which configuration have been chosen. If the Media Capture and Player Device chosen, in the Configuration window, is "Java Sound", the audio data is acquired (and played) from (over) the same computer where the application runs. Acquiring the audio data from the Microphone and playing the audio data received over the speakers. Otherwise, if the chosen option is ”UDP Adapter” the audio data is sent (and acquired) to (from) the specified IP Address and Port (PAN Interface).

5.5 SIP VoIP Adapter Class Description

The SipClient Adapter goal was to create a set of basic classes that could perform a SIP communication between two end users, which are connected to a private branch exchange (PBX) through a local area network (LAN), and handles the media transmitted between the users over an other personal area network (PAN) interface. This section contains a main description of each class to understand the principal elements of the SIP VoIP Adapter. This is divided in packages, each of them, manage different aspects for controlling the media communication audio call over the internet protocol (IP) besides the media transmitted over the PC side interfaces. Figure 5.15 shows how the packages are divided.

5.5.1 User-Agent package

This package is formed by the main classes that manage and control the Jain-Sip architecture layer to establish and control the sip phone communication call.

MessageListener.java

Interface that defines the methods to create a SIP communication channel. This class implements the SipListener stack and it’s notified of every change in the communication status by handling the SIP listener methods. It creates factories to build headers and send messages besides to start and configures the Jain-Sip stack and so the user agent.

public class MessageListener implements SipListener() {

public void ProcessRequest(RequestEvent evt) {}

Process the request events received by the SipListener stack. Analyze the message request type and call the consequent method of the MessageProcessor class to handle.

public void ProcessResponse(ResponseEvent evt) {}

Process the response events received by the SipListener stack. Analyze the message response and call the consequent method of the MessageProcessor class to handle it.

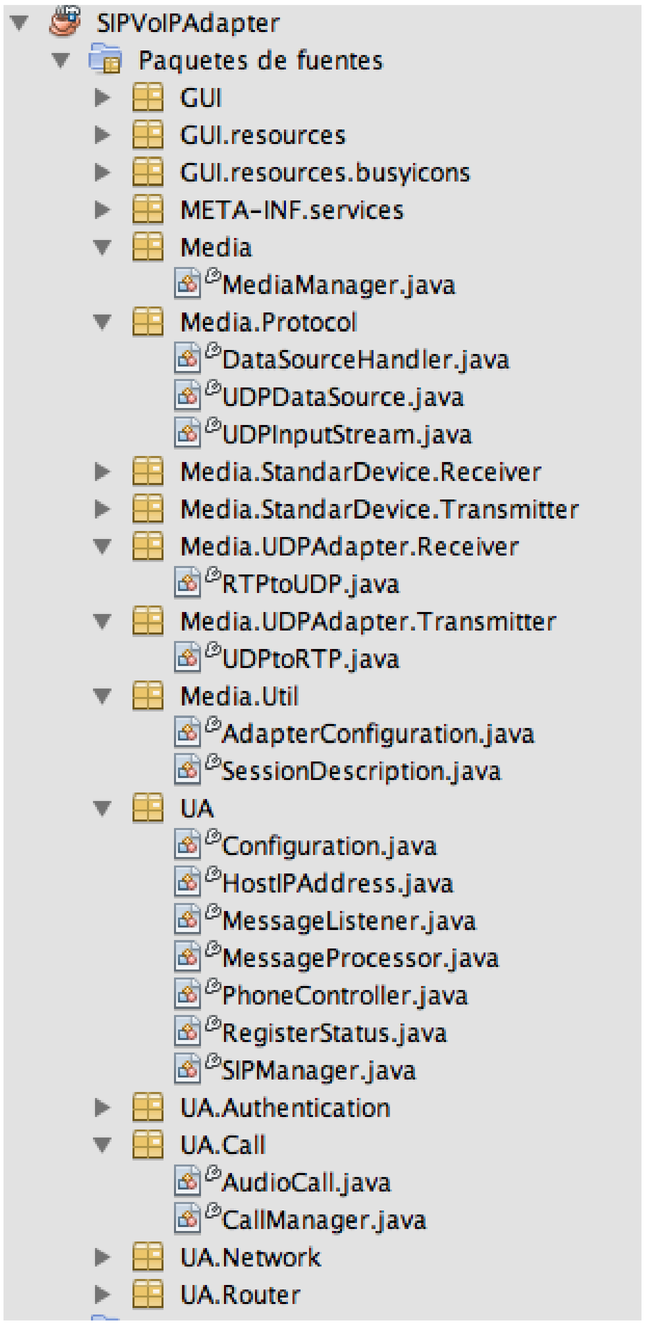
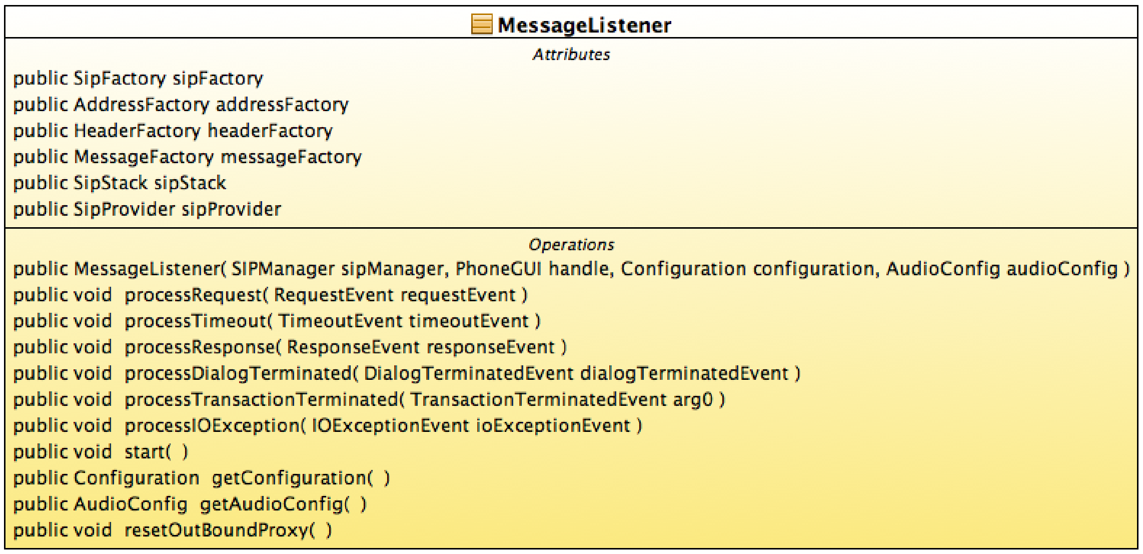


Figure 5.15: SIP VoIP Adapter package scheme


MessageListener

Attributes

```
public SipFactory sipFactory
public AddressFactory addressFactory
public HeaderFactory headerFactory
public MessageFactory messageFactory
public SipStack sipStack
public SipProvider sipProvider
```

Operations

```
public MessageListener( SIPManager sipManager, PhoneGUI handle, Configuration configuration, AudioConfig audioConfig )
public void  processRequest( RequestEvent requestEvent )
public void  processTimeout( TimeoutEvent timeoutEvent )
public void  processResponse( ResponseEvent responseEvent )
public void  processDialogTerminated( DialogTerminatedEvent dialogTerminatedEvent )
public void  processTransactionTerminated( TransactionTerminatedEvent arg0 )
public void  processIOException( IOExceptionEvent ioExceptionEvent )
public void  start(  )
public Configuration  getConfiguration(  )
public AudioConfig  getAudioConfig(  )
public void  resetOutBoundProxy(  )
```

Figure 5.16: MessageListener class overview

public void ProcessTimeout(TimeoutEvent evt) {}

Process the time out request events received by the SipListener stack. Analyze the message request type and call the MessageProcessor to handle it.

A Transaction is a request sent by a client transaction to a server transaction, along with all responses to that request sent from the server transaction back to the client transactions. It can be broken down in two different types depending on if it's originated from a server or a client.

A Request/Response object lets the application know the event type generated to handle a consequent answer.

public void start() {}

This method is needed to configure the specific properties of the SIP stack in addition to create the SIP communication channel. The SipStack interface defines the methods that are used by this class to control the architecture and setup of the SIP stack. So, the method *start()* set the stack properties and creates the stack, creates the ListeningPoint to an specific port to be able to initialize the SipProvider, and initialize the SipProvider interface registering in it the MessageListener.

The SipProvider interface represents the messaging entity of a SIP stack and lets the interface implementing SipListener to be notified of Events representing either Request, Response and Timeout messages.

A procedure example to register an object implementing the SipListener interface to the SipProvider, allowing the object listen for the SIP exchange events is as follows:

```
sipFactory = SipFactory.getInstance(); //create the SipFactory object
sipFactory.setPathName(“gov.nis“); //the Path Name is fixed

Properties properties = new Properties(); //Create the SipStack Properties
Properties.setProperty(“javax.sip.StackName“, name);
Properties.setProperty(“javax.sip.IPAddress“, hostIPAddress);

//... see annex 10.1 for all configurable properties...

//Creates the stack:
SipStack sipStack = sipFactory.createSipStack(properties);
ListeningPoint lp = sipStack.createListeningPoint(listeningPort, transportProtocol);

//creates the SipProvider object:
SipProvider sipProvider = sipStack.createSipProvider(lp);

//register SipListener interface to catch the events:
sipProvider.addSipListener(this);
```

}

MessageProcessor.java

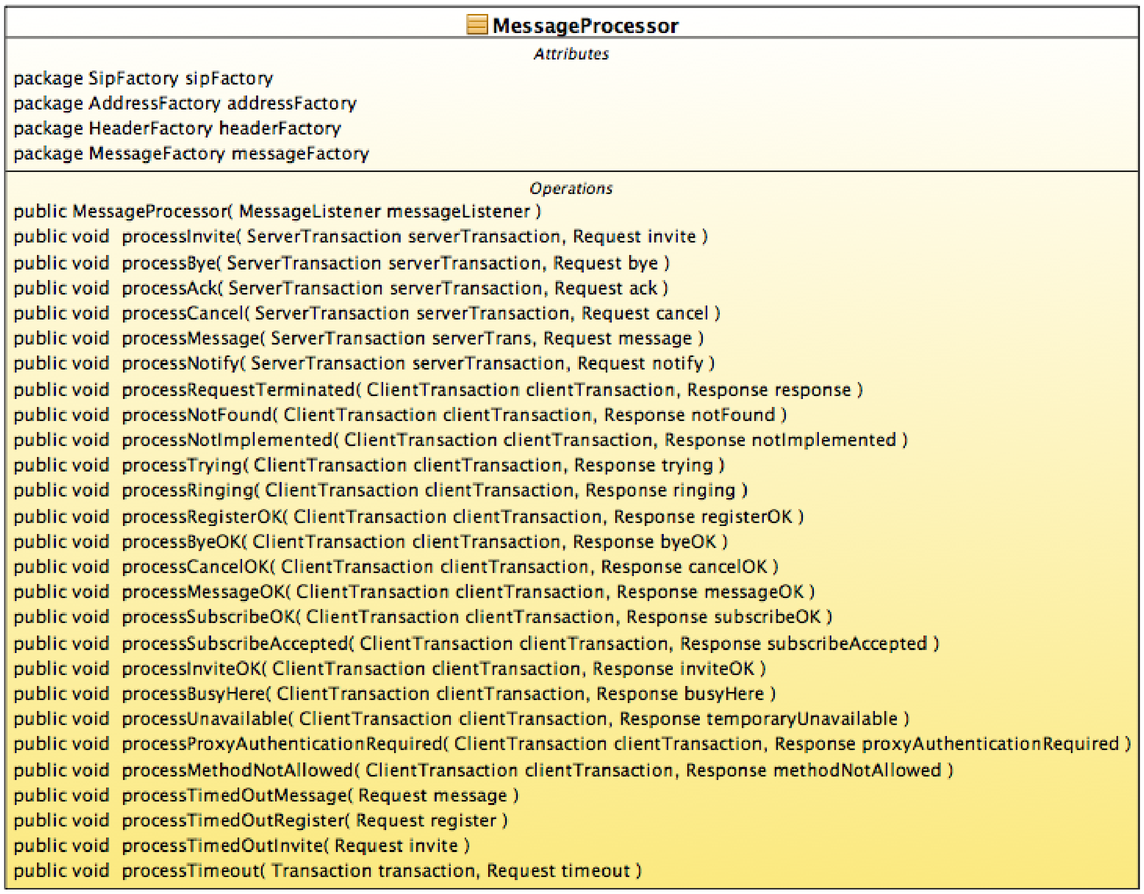
This class contain all necessary methods to process the messages handled by the message-Listener. Its main function is to govern the application creating different actions depending on the events caught by the messageListener and answer corresponding to this.

public class MessageProcessor() {

The main method in this class which handles the requested messages are:

public void ProcessInvite(ServerTransaction transaction, Request invite) {}

Process the INVITE received request. Call the CallManager object to see if the user is already in a call. If so, a BUSY HERE response message is created and send it back.



```
MessageProcessor
Attributes
package SipFactory sipFactory
package AddressFactory addressFactory
package HeaderFactory headerFactory
package MessageFactory messageFactory
Operations
public MessageProcessor( MessageListener messageListener )
public void  processInvite( ServerTransaction serverTransaction, Request invite )
public void  processBye( ServerTransaction serverTransaction, Request bye )
public void  processAck( ServerTransaction serverTransaction, Request ack )
public void  processCancel( ServerTransaction serverTransaction, Request cancel )
public void  processMessage( ServerTransaction serverTrans, Request message )
public void  processNotify( ServerTransaction serverTransaction, Request notify )
public void  processRequestTerminated( ClientTransaction clientTransaction, Response response )
public void  processNotFound( ClientTransaction clientTransaction, Response notFound )
public void  processNotImplemented( ClientTransaction clientTransaction, Response notImplemented )
public void  processTrying( ClientTransaction clientTransaction, Response trying )
public void  processRinging( ClientTransaction clientTransaction, Response ringing )
public void  processRegisterOK( ClientTransaction clientTransaction, Response registerOK )
public void  processByeOK( ClientTransaction clientTransaction, Response byeOK )
public void  processCancelOK( ClientTransaction clientTransaction, Response cancelOK )
public void  processMessageOK( ClientTransaction clientTransaction, Response messageOK )
public void  processSubscribeOK( ClientTransaction clientTransaction, Response subscribeOK )
public void  processSubscribeAccepted( ClientTransaction clientTransaction, Response subscribeAccepted )
public void  processInviteOK( ClientTransaction clientTransaction, Response inviteOK )
public void  processBusyHere( ClientTransaction clientTransaction, Response busyHere )
public void  processUnavailable( ClientTransaction clientTransaction, Response temporaryUnavailable )
public void  processProxyAuthenticationRequired( ClientTransaction clientTransaction, Response proxyAuthenticationRequired )
public void  processMethodNotAllowed( ClientTransaction clientTransaction, Response methodNotAllowed )
public void  processTimedOutMessage( Request message )
public void  processTimedOutRegister( Request register )
public void  processTimedOutInvite( Request invite )
public void  processTimeout( Transaction transaction, Request timeout )
```

Figure 5.17: MessageProcessor class overview

Otherwise an AudioCall object is created to initialize the incoming call, the call status is set it to IN A CALL and notified to the observers, and a TRYING response message is created and send it back.

public void ProcessBye(ServerTransaction transaction, Request invite) {}

Process the BYE received request. Stops the media session, change the call status to NOT IN A CALL and notifies this change. An OK response message is created and send it back.

public void ProcessAck(ServerTransaction transaction, Request ack) {}

Process the ACK received request. Finds the audio call and starts the media session, change the call status to IN A CALL and notifies this change.

public void ProcessCancel(ServerTransaction sTransaction, Request cancel) {}

Process the CANCEL received request. Remove the current audio call, change the call status to CANCEL and notifies this change. An OK response message is created and send it back.

All of this methods handles the request messages received, so each of them needs to sends back an answer to that request. The MessageFactory interface provides factory methods to create the responses messages and it can be called through the MessageListener object. Otherwise ServerTransaction interface provides methods to send the response to a request associated with this ServerTransaction.

So, all the methods in this class that handles request messages, creates and sends the responses as follows:

```
//Create the response Type to an specific request:
Response response = (Response) MessageListener.messageFactory.
Response(Response.Type,request)

//Sends the response through the ServerTransaction object:
serverTransaction.sendResponse(response);
```

The methods that manage call requests need to care about who is sending the request to manage the correct AudioCall. Therefore, it is necessary to extract the user's URI from the ToHeader argument received in the request message as follows:

```
//Create the response Type to an specific request:
SipURI calleeURI = (SipURI) ((FromHeader) request.getHeader(FromHeader.NAME)).
getAddress().getURI();
```

To notify the observers for a change in the register status or in the call status, MessageProcessor use the method provided by the SipManager interface as *notifyNewCallStatus(Call call)* and *setRegisterStatus(String status)* which change the register status as well as notifies its change.

The main methods in the MessageProcessor class which handles the responses messages are:

public void ProcessTrying(ClientTransaction trans, Response trying) {}

Process the Trying received response. Finds the audio call, change the call status to TRYING and notifies this change.

public void ProcessRinging(ClientTransaction trans, Response ringing) {}

Process the Ringing received response. Finds the audio call, change the call status to RINGING and notifies this change.

public void ProcessRegisterOK(ClientTransaction trans Response registerOk) {}

Process the OK received response for a REGISTER. Update the register status to REGISTERED or NOT REGISTERED (and notifies this change).

public void ProcessByeOK(ClientTransaction trans, Response byeOk) {}

Process the OK received response for a BYE. Finds and remove the audio call, change its call status to NOT IN A CALL and notify this change.

public void ProcessCancelOK(ClientTransaction trans, Response cancel) {}

Process the OK received response for a CANCEL. Finds and remove the audio call, change its call status to NOT IN A CALL and notifies this change.

public void ProcessInviteOK(ClientTransaction trans, Response inviteOk) {}

Process the OK received response for an INVITE. If the content type header received supports SDP, starts the media session, change the call status to IN A CALL and notify this change. An ACK message is send back.

public void ProcessBusyHere(ClientTransaction trans, Response busy) {}

Process the BUSY HERE received response for an INVITE. Check if it comes from a call instance and if so, finds and remove the audio call, change its call status and notifies this change.

ProcessProxyAuthenticationRequired(ClientTransaction trans Response authRequired) {}

Change the register status to PROXY AUTHENTICATION REQUIRED. It will be the PhoneController the responsible to ask for the userÕs authentication.

Only the method *processInviteOK()* needs to send back an answer, but, unlike the receiver requests, it has to be inside a Dialog. So, the answer message is created and send it as follows:

```
Request ack = (Request) clientTransaction.getDialog().createRequest(Request.ACK);

//Send response through the Dialog:
clientTransaction.getDialog().sendAck(ack);
```

The methods that manage call responses need to care about who is sending the response to manage the correct AudioCall. Therefore, it is necessary to extract the user's URI to find the corresponding call. If the user is in a communication with another end user, the call has to be found inside the Dialog between users as follows:

```
Call call = callManager.findCall(clientTransaction.getDialog().getDialogID());
```

If the call has finished or it has not already started, we are able to find it from the ToHeader argument received in the response message as follows:

```
SipURI calleeURI = (SipURI) ((FromHeader) request.getHeader(FromHeader.NAME)).
getAddress().getURI();
```

}

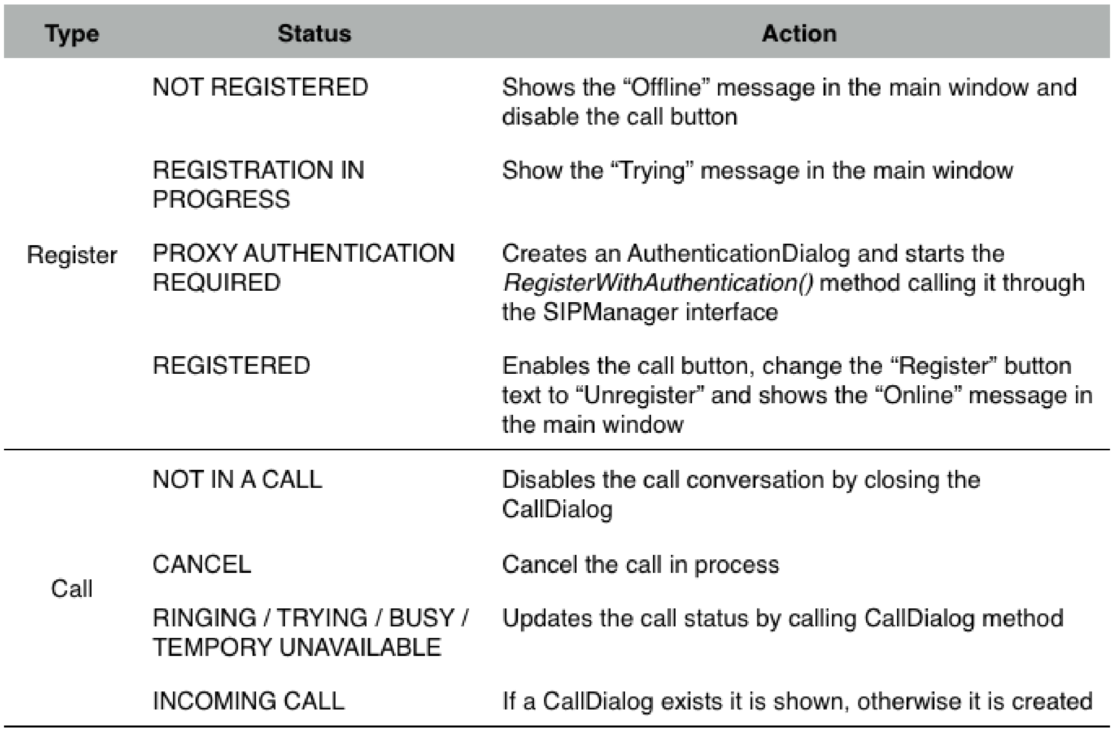
PhoneController.java

This class represents the control part in the MVC pattern for this application, so that when there is a change in the model it updates the view. This class implements the interface *Observer* pattern, therefore it only has the method *update()* for two different types of instances, *AudioCall* instances and *RegisterStatus* instances. Therefore, when one of these status change, the PhoneController generates the specific interface action depending on the change occurred.

public class PhoneController implements Observer() {

public void Update() {}

This method catch two kind of instances, *AudioCall* instances which depending on the call status, creates or close a *Call Dialog* window and writes statics updates. And *RegisterStatus* instances which updates the register status label view and enables or disables the *Make a Call* button.



Type	Status	Action
Register	NOT REGISTERED	Shows the "Offline" message in the main window and disable the call button
	REGISTRATION IN PROGRESS	Show the "Trying" message in the main window
	PROXY AUTHENTICATION REQUIRED	Creates an AuthenticationDialog and starts the *RegisterWithAuthentication()* method calling it through the SIPManager interface
	REGISTERED	Enables the call button, change the "Register" button text to "Unregister" and shows the "Online" message in the main window
Call	NOT IN A CALL	Disables the call conversation by closing the CallDialog
	CANCEL	Cancel the call in process
	RINGING / TRYING / BUSY / TEMPORY UNAVAILABLE	Updates the call status by calling CallDialog method
	INCOMING CALL	If a CallDialog exists it is shown, otherwise it is created

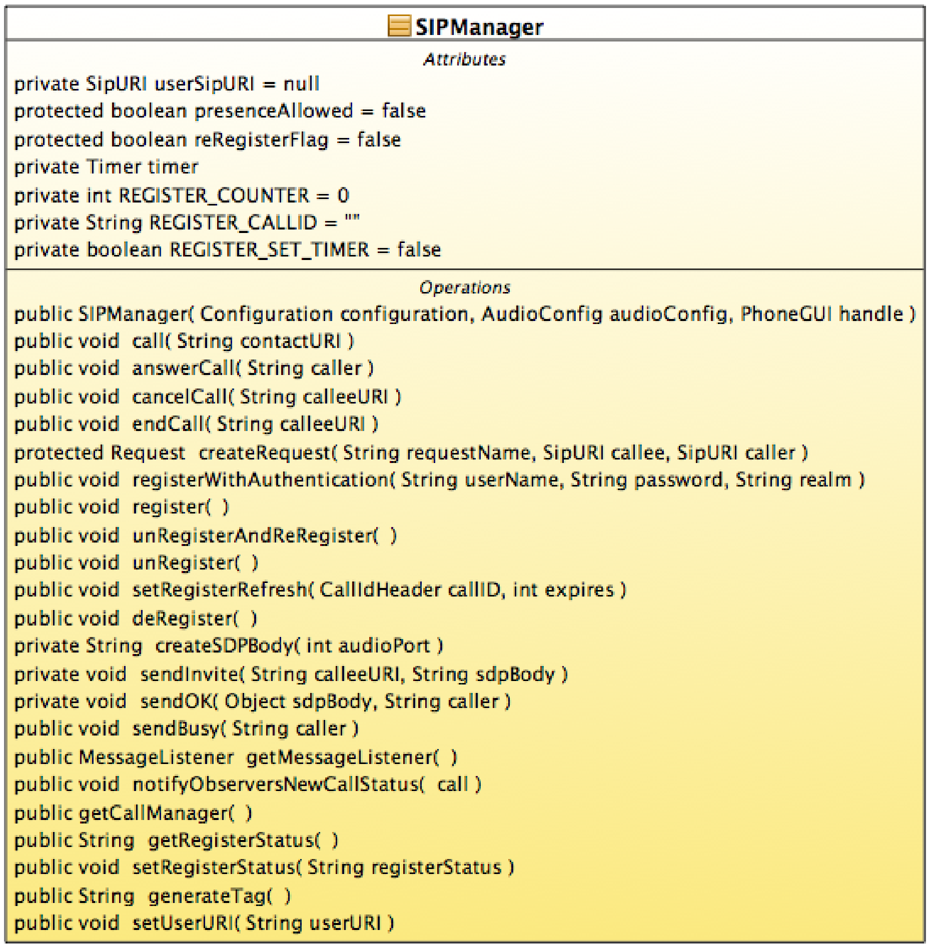
Figure 5.18: Shows the actions depending on the call or register status

SIPManager.java

This class represents the model part in the MVC pattern for this application, therefore it is the domain-specific representation of the data on which the application operates. This class implements the interface pattern *Observable* and manage the calls and their status. In a general point of view, this class manage all the actions the user can do depending on their register status and call status.

public class SIPManager {

public void call(String contactURI) {}



```
SIPManager
Attributes
private SipURI userSipURI = null
protected boolean presenceAllowed = false
protected boolean reRegisterFlag = false
private Timer timer
private int REGISTER_COUNTER = 0
private String REGISTER_CALLID = ""
private boolean REGISTER_SET_TIMER = false
Operations
public SIPManager( Configuration configuration, AudioConfig audioConfig, PhoneGUI handle )
public void  call( String contactURI )
public void  answerCall( String caller )
public void  cancelCall( String calleeURI )
public void  endCall( String calleeURI )
protected Request  createRequest( String requestName, SipURI callee, SipURI caller )
public void  registerWithAuthentication( String userName, String password, String realm )
public void  register(  )
public void  unRegisterAndReRegister(  )
public void  unRegister(  )
public void  setRegisterRefresh( CallIdHeader callID, int expires )
public void  deRegister(  )
private String  createSDPBody( int audioPort )
private void  sendInvite( String calleeURI, String sdpBody )
private void  sendOK( Object sdpBody, String caller )
public void  sendBusy( String caller )
public MessageListener  getMessageListener(  )
public void  notifyObserversNewCallStatus(  call )
public getCallManager(  )
public String  getRegisterStatus(  )
public void  setRegisterStatus( String registerStatus )
public String  generateTag(  )
public void  setUserURI( String userURI )
```

Figure 5.19: SIPManager class overview

Call the contact specified in the URI parameter. To perform this action a new Audio Call is created and parsed to the Media Manager, the SDP body of the INVITE message is created and send it as follows:

```
String sdpBody = createSDPBody(call.getMediaManager().getAudioPort())

sendInvite(contactURI, sdpBody);
```

public void answerCall(String caller) {}

Answer the call received. In oder words, answer OK to an incoming INVITE after it has configured the media session depending on the SDP description received. To perform this action it finds the Audio Call related to the caller, it gets the request messages associated to that call by getting the Dialog related. It check to see if the SDP content is set it, and gets its content to compare if the application can accept the media described in it. Once the response SDP message is created, it is sent by calling the method *sendOK(resonseSDP, caller)*.

The SDP message is obtained from the Dialog related to the incoming call, and a response SDP message is created as follows:

```
//Check if it there is an SDP message associated:
ContentTypeHeader header = (ContentTypeHeader) request.getHeader
(ContentTypeHeader.NAME);
String type = header.getContentType();
String subType = header.getContentSubType();

if (type.equals(“application“) && subType.equals(“application“) && subType.
equals(“sd“)){
    //Get the SDP body from the request:
    String content = new String(request.getRawContent());
    //Create the response SDP body:
    responseSdpBody = mediaManager.getResponseSdpBody(content);
}
sendOK(responseSdpBody, caller);
```

public void endCall(String calleeURI) {}

Ends the current call. To perform this action it finds the Audio Call, stops the media session and sends a BYE message.

public void register() {}

Sends a REGISTER request message to the proxy especified in the Configuration. First of all, it creates the proxy URI from the Configuration object by calling the method *messageListener.getConfiguration()*, increase the register counter and creates the request calling the *createsRequest()* method. The header values for a registration action are added, as the expires in the contact header, the CSeq header and the call ID header.

To send the request, a new ClientTransaction is created by calling the SipProvider. This action is implemented as follows:

ClientTransaction trans = messengerListener.sipProvider.getNewClientTransaction (request);

trans.sendRequest();

And finally, the register status is changed to REGISTRATION IN PROGRESS (notifying this change), and a timer is set.

public void registerWithAuthentication(String name, String pwd, String real) {}

Creates and sends a REGISTER request message to the proxy with the authentication acquired from the user. It gets the request sent as the first instance to add the authentication parameters on the header. The REGISTER COUNTER and the CSeq number are updated, and a new client transaction is created to send the request.

public void unregister() {}

Creates and sends a UNREGISTER request message to the proxy. To do this action it creates the unregister request as the *register()* method, but fixing the the expires header to zero.

The following inner methods are created to perform the request actions already explain it, as make a Proxy registration or a call:

public Request createRequest(String name, SipURI callee, SipURI caller) {}

Given a request type, the SIP URI of the callee, and the SIP URI of the caller, it creates the request to be sent. It builds the message header for the corresponding callee and caller, by getting the Sip Factories from the MessageListener interface. The request headers and the request by itself are created as follows:

requestURI = callee;
//Call ID
CallIdHeader callIdHeader =messageListener.sipProvider.getNewCallId();

//CSeq
CSeqHeader cSeqHeader = MessageListener.headerFactory.createCSeqHeader (1,name);

```
//From Header
SipURI callerURI = MessageListener.addressFactory.createSipURI(caller.getUser(),
caller.
getHost()));
Address fromAddress = MessageListener.addressFactory.createAddress(callerURI);
FromHeader fromHeader = MessageListener.headerFactory.createFromHeader
(fromAddress,generateTag());

//ToHeader
SipURI calleeURI = MessageListener.addressFactory.createSipURI(callee.getUser(),
callee.
getHost());
Address toAddress = MessageListener.addressFactory.createAddress(calleeURI);

// From and To are logical identifiers (should have no parameters):
ToHeader toHeader = MessageListener.headerFactory.createToHeader(toAddress,
null);

//ViaHeader
ArrayList viaHeaders = new ArrayList();
ViaHeader viaHeader = MessageListener.headerFactory.createViaHeader
(contactIPAddress,listeningPoint.getPort(), listeningPoint.getTransport(), null);
viaHeaders.add(viaHeader);

//Max Forward Header
MaxForwardsHeader maxForwardsHeader =MessageListener.headerFactory.createMax
ForwardsHeader(70);
Request request =MessageListener.messageFactory.createRequest(requestURI, re-
questName,
callIdHeader,cSeqHeader,fromHeader,toHeader,viaHeaders,max
ForwardsHeader);

//Contact Header
SipURI contactURI = MessageListener.addressFactory.createSipURI(caller.getUser(),
contactIPAddress);
Address contactAddress = MessageListener.addressFactory.createAddress(contactURI);
ContactHeader contactHeader = MessageListener.headerFactory.createContactHeader
(contactAddress);
contactURI.setPort(listeningPoint.getPort());
request.addHeader(contactHeader);
```

private String createSDPBody(int audio) {}

Given an audio port, it creates the SDP body of the INVITE message for the initialization of the media session. The Jain-SDP API provides methods to construct the message body. However, this method builds the SDP content ÔmanuallyÕ, using only the sdpFactory object to build the MediaDescription content. It has been developed by this way to make it compatible with IPv6. Although according to the Jain-SDP API specifications, it is compatible with IPv6, we have had problems to pass the address correctly to be understood by Asterisk, therefore, and thanks to the body content can be passed as a String, it has been written manually as follows:

```
SdpFactory sdpFactory = SdpFactory.getInstance();
//“v=0“ String version = “v=0“;

//Origin “o“
String addrType;
if(messageListener.getConfiguration().isIPv6)
    addrType=“IP6“;
else
    addrType=“IP4“;
long sdpSessionId = (long)(Math.random() * 1000000);
String origin = “o=“+userSipURI.getUser() + “ “ + sdpSessionId + “ “ +
(sdpSessionId+1369)
+“ IN “+ addrType + “ “+messageListener. getConfiguration(). stackIPAddress
+ “ “;

//Session Name “s=-“
String sessionName = “s=- “;
//Connection “c=“
String connection = “c=IN “+addrType+“ “+messageListener.getConfiguration()
.stackIPAddress + “ “;

//Time “t=0 0“
String time = “t=0 0 “;

MediaDescription mediaDescription =sdpFactory.createMediaDescription(“audio“,
audioPort, 1, “RTP/AVP“, MediaManager.getSdpAudioSupportedCodecs());
Vector mediaDescriptions = new Vector();
mediaDescriptions.add(mediaDescription);

String sdpData = version + origin + sessionName + connection + time + me-
diaDescriptions.element At(0).toString() +“ “;
```

private void sendInvite(String calleeURI, String sdpBody) {}

Given the SIP URI of the user to call, and the SDP content describing the media session to include to the message, it creates and sends the INVITE request message with the SDP body content added. So, this method builds the request by calling the inner method *createRequest(Request.INVITE, contactURI,userSipURI)*, it creates the content type header by calling the headerFactory object from the MessageListener, and adds both, header and content to the request. Finally, it creates a new ClientTransaction and sets the new Dialog to the call as follows:

```
ClientTransaction inviteTransaction = inviteTransaction = messageListener.sipProvider.
getNewClientTransaction(inviteRequest);

inviteTransaction.sendRequest();

AudioCall call = callManager.findAudioCall(”sip:” + calleeURI);
call.setDialog(inviteTransaction.getDialog());
```

private void sendOK(Object sdpBody, String caller)) {}

Given the SDP content describing the media session to include in the response, and the caller, it sends an OK message in response to an incoming INVITE. It gets the request sent in a first instance, and the ServerTransaction related to the caller call. And it creates the response OK message, adding the contact header, as well as the content type header and body of the SDP message This method have been implemented as follows:

```
//Find the Audio call
AudioCall call = callManager.findAudioCall(caller);

//Get the request
Request request = call.getDialog().getFirstTransaction().getRequest();

//Get the server Transaction
ServerTransaction serverTransaction =(ServerTransaction) call.getDialog().
getFirstTransaction();
Response ok =(Response) MessageListener.messageFactory.createResponse
(Response.OK, request);

//Put a tag on the To Header:
((ToHeader) ok.getHeader(ToHeader.NAME)).setTag(generateTag());

//Specify the contact Header
SipURI contactURI =MessageListener.addressFactory.createSipURI(userSipURI
.getUser(), contactIPAddress);
Address contactAddress = MessageListener.addressFactory.createAddress
(contactURI);
```

```
ContactHeader contactHeader =MessageListener.headerFactory.createContact
Header(contactAddress);
contactURI.setPort(listeningPoint.getPort());
ok.addHeader(contactHeader);

//Add the sdp Body describing the media session
ContentTypeHeader contentTypeHeader =(ContentTypeHeader) request.
getHeader(ContentTypeHeader.NAME);
ok.setContent(sdpBody, contentTypeHeader);

//Send the ok response
serverTransaction.sendResponse(ok);
```

public void sendBusy(String caller) {}

It sends a BUSY HERE response to an incoming INVITE request. It finds the call related to the Request and obtains the SeverTransaction and the Request from the Dialog stored in the call. It creates the response BUSY HERE and adds the URL defined by default in the Call Info Header. Finally, the response is transmitted through the ServerTransaction and the call is removed. This actions has been implemented as follows:

```
AudioCall call = callManager.findAudioCall(caller);

//Get the request
Request request = call.getDialog().getFirstTransaction().getRequest();

//Get the server Transaction
ServerTransaction serverTransaction = (ServerTransaction) call.getDialog()
.getFirstTransaction();
Response busy = (Response) MessageListener.messageFactory.createResponse
(Response.BUSY HERE,request);

//Adds the URL for the BUSY HERE response in the CALL-Info
CallInfoHeader callInfoHeader=MessageListener.headerFactory.createCallInfoHeader
(MessageListener.addressFactory.createURI(httpBusy));
busy.addHeader(callInfoHeader);

serverTransaction.sendResponse(busy);
callManager.removeAudioCall(call);
```

The following methods are created to retrieve some interfaces used in this class:

public void notifyObserversNewCallStatus(Call call) {}

Notify the GUI and the controller that the call status has changed.

public void setRegisterStatus(String status) {}

Change the register status and notifies this change to the GUI and the controller.

public MessageListener getMessageListener() {}

Retrieves the MessageListener interface to allow its accessibility through the SipManger.

public CallManager getCallManager() {}

Retrieves the CallManager interface to allow its accessibility through the SipManger.

public RegisterStatus getRegisterStatus() {}

Retrieves the RegisterStatus interface to allow its accessibility through the SipManger.

Configuration.java

This class represents the configuration of the user agent, with all the parameters necessaries to establish a SIP communication using the JAIN-SIP library for Java.

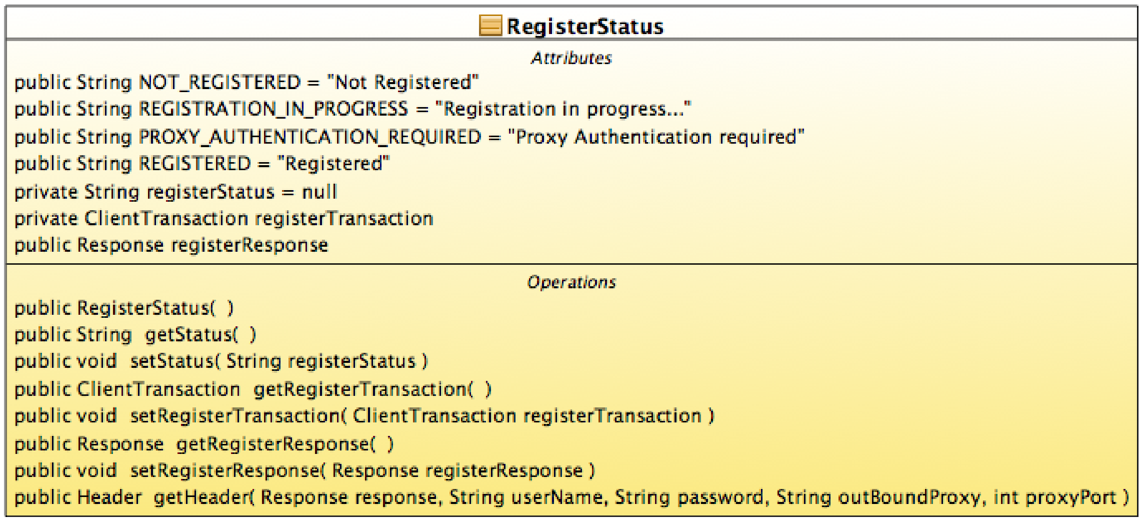
The following parameters defines the stack:

RegisterStatus.java

This class represents the registration status of the application. It contains the parameters needed for the registration process as the RegisterStatus, the current ClientTransaction, and the current registration response. It also provides the method to create the Proxy Authorization Header for the register request.

Parameter	Description
stackName	SipStack name
stackIPAddress	Local Host IP address for the SipStack initia-litzation
outboundProxy	IP Proxy address
domain	Without DNS it is the IP Proxy Address
proxyPort	Listening Proxy port
listeningPort	Listening application port
signalingTransport	Protocol used for the SIP signaling transport
mediaTransport	Protocol used for the media transport
userURI	User URI is formed as follows: <User>@<ProxyAddress>
userSipUri	User Sip URI is formed containing some atributs as user, host, port, password, TTL, transport pa-rameter, etc. See RFC3261 for more information
isIPv6	True if the application uses IPv6 addresses and false if uses IPv4.

Figure 5.20: SIP Stack patameters



```
RegisterStatus
Attributes
public String NOT_REGISTERED = "Not Registered"
public String REGISTRATION_IN_PROGRESS = "Registration in progress..."
public String PROXY_AUTHENTICATION_REQUIRED = "Proxy Authentication required"
public String REGISTERED = "Registered"
private String registerStatus = null
private ClientTransaction registerTransaction
public Response registerResponse
Operations
public RegisterStatus( )
public String  getStatus( )
public void  setStatus( String registerStatus )
public ClientTransaction  getRegisterTransaction( )
public void  setRegisterTransaction( ClientTransaction registerTransaction )
public Response  getRegisterResponse( )
public void  setRegisterResponse( Response registerResponse )
public Header  getHeader( Response response, String userName, String password, String outBoundProxy, int proxyPort )
```

Figure 5.21: RegisterStatus class overview

public class RegisterStatus {

public Header getHeader(Response response, String userName, String password, String outBoundProxy, int proxyPort) {}

This method is used to create the Proxy Authorization Header for the register request. This method has been implemented as follows:

```
// Proxy-Authorization header:
ProxyAuthenticateHeader authenticateHeader = (ProxyAuthenticateHeader) re-
sponse.getHeader (ProxyAuthenticateHeader.NAME);
WWWAuthenticateHeader wwwAuthenticateHeader=null;
CSeqHeader cseqHeader = (CSeqHeader) response.getHeader(CSeqHeader.NAME);

String cnonce=null;
String uri=”sip:” + outBoundProxy + ”:” + proxyPort;
String method=cseqHeader.getMethod();
String nonce=null;
String realm=null;
String qop=null;

if (authenticateHeader==null) {
    wwwAuthenticateHeader = (WWWAuthenticateHeader) response.getHeader
    (WWWAuthenticateHeader.NAME);
    nonce=wwwAuthenticateHeader.getNonce();
    realm=wwwAuthenticateHeader.getRealm();
    cnonce=wwwAuthenticateHeader.getParameter(”cnonce”);
    qop=wwwAuthenticateHeader.getParameter(”qop”);

} else{
    nonce=authenticateHeader.getNonce();
    realm=authenticateHeader.getRealm();
    cnonce=authenticateHeader.getParameter(”cnonce”);
    qop=authenticateHeader.getParameter(”qop”);

}
HeaderFactory headerFactory=MessageListener.headerFactory;
MessageDigestAlgorithm digest=new MessageDigestAlgorithm();
String responseAlgorithm = digest.generateResponse(realm, userName, uri, nonce,
password, method, cnonce, ”MD5”);

if (authenticateHeader==null)
```

```
        AuthorizationHeader header=headerFactory.createAuthorizationHeader("Digest");
    else
        ProxyAuthorizationHeader header = headerFactory.createProxyAuthorization
        Header("Digest");
    header.setParameter("username",userName);
    header.setParameter("realm",realm);
    header.setParameter("uri",uri);
    header.setParameter("algorithm","MD5");
    header.setParameter("opaque","");
    header.setParameter("nonce",nonce);
    header.setParameter("response",responseAlgorithm);
    header.setParameter("qop",qop);

    return header;
```

5.5.2 UserAgent.Call package

This package is formed by the main classes that performs the audio call. The interface AudioCall represents a call that can be received by the application and keeps the information about the audio call. The CallManager class manage a vector of AudioCalls and its related dialogs to show the call on a window.

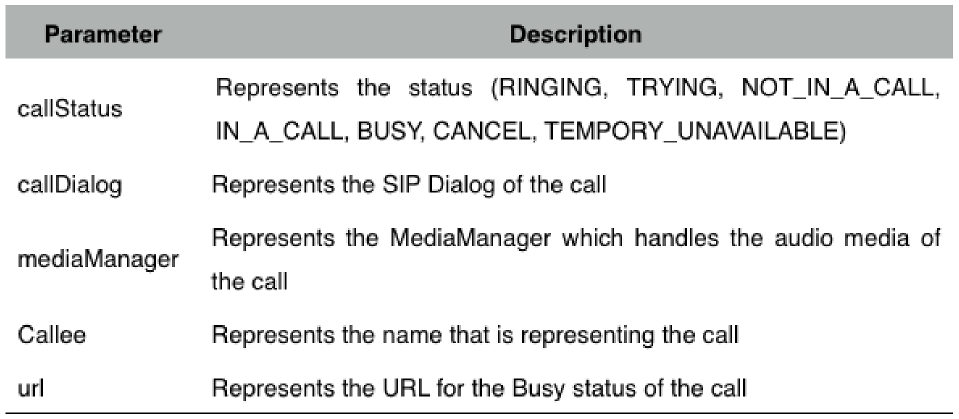
AudioCall.java

This interface represents a call that the application is able to handle, it contains the parameters related to an specific call. It also defines the possible status of a call. Therefore it only has methods get and set.

The main parameters of an AudioCall are:

CallManager.java

This class manage all calls in the application. It contains the methods to allow the application control more than only one call, however only one active. It also handles the Dialogs related to an specific call and provides a method to tell if a media session is already active.



Parameter	Description
callStatus	Represents the status (RINGING, TRYING, NOT_IN_A_CALL, IN_A_CALL, BUSY, CANCEL, TEMPORY_UNAVAILABLE)
callDialog	Represents the SIP Dialog of the call
mediaManager	Represents the MediaManager which handles the audio media of the call
Callee	Represents the name that is representing the call
url	Represents the URL for the Busy status of the call

Figure 5.22: AudioCall patameters

5.5.3 UserAgent.Authentication package

This package is formed by the class that performs the authentication method.

MessageDigestAlgorithm.java

The MessageDigestAlgorithm class takes standard Http Authentication details and returns a response according to the MD5 algorithm as described in *RCF 2617*.

5.5.4 UserAgent.Router package

CustomRouter.java

This interface represents the router of the application. When the implementation want to forward a request and had run out of other options, then it calls this method to figure out where to send the request. The default router implements a simple "default routing algorithm" which just forwards to the configured proxy address.

public class CustomRouter implements Router {

private boolean hopsBackToMe(String host, int port) {}

Return true if there is a listener listening here.

public ListIterator getNextHops(Request request){}

Return addresses for default proxy to forward the request to. The list is organized in the following priority:

1. If the requestURI refers directly to a host, the host and port information are extracted from it and made the next hop on the list.
2. If the default route has been specified, then it is used to construct the next element of the list.

This method has been developed as follows:

```
LinkedList ll = null;
if (! checked) {
    checked = true;
    if ( defaultRoute != null)
        localLoopDetected = hopsBackToMe(defaultRoute.getHost(),
        defaultRoute.getPort());
} if (defaultRoute != null && !localLoopDetected ) {
    ll = new LinkedList();
    ll.add(defaultRoute);
}
return ll == null ? null : ll.listIterator();
```

public Hop getNextHop(Request request){}

Return the default route given by the object CustomHop.

javax.sip.address.Hop getOutboundProxy(){}

Return the default route given by the object CustomHop.

public Hop getDefaultRoute(){}

Return the default route given by the object CustomHop.

CustomHop.java

This class represents a Hop, which is an string in the form of host:port/transport. This class extracts this parameters in its initialization and it also contains the methods to set or get this communication parameters individually.

5.5.5 UserAgent.Network package

NetworkAddressManager.java

This class handles network address selection to let the application be compatible with both versions of IP, IPv4 and IPv6.

5.5.6 Media package

This package is formed by the classes that manage and control the media session. The classes containing this package implements the media session using the Java Media Frameworks (JMF) library which implements the RTP and RTCP stack as well as the audio treatment.

MediaManager.java

This class handles the media part of the audio call, starts the media transmission and reception and manage the supported codecs to use for the communication.

public class MediaManager {

public static void detectSuportedCodecs() {}

Detects the supported codecs where the application is running, depending on the devices connected to the computer. This function is called by the SipManager when it is initialized and stores the supported codecs information to be able to chose one of them for the media communication. First of all, this method uses the interface CaptureDeviceManager to find the devices available on the computer to create an AudioLocator. Once it is detected it creates a DataSource and also a Processor to handle it. The output ContentDescriptor of the Processor is set to RAW RTP to know the only valid format for the RTP communication, and finally, the tracks are extracted from the Processor to find and store the usefully codecs on a static List called audioCodecSupportedList.

public static String[] getSdpAudioSupportedCodecs() {}

Return the audio codecs supported by the computer where the application is running , in the SDP format. Once the method detectSupportedCodecs has been called (in the initialization of SipManager) and so, the audioCodecSupportedList filled. This method gets its corresponding value in the SDP format.

public void prepareMediaSession(String incomingSdpBody) {}

Extracts from the SDP all the information to initiate the media session, as the remote IP address, the remote port, the local port and finally, the supported codecs for then, negotiate a compatible audio codec. This method creates a SessionDescription object from the SDP content given by calling the function *sdpFactory.createSessionDescription (incomingSdpBody)*, and gets the connection related to extract the remote IP address by calling the method *sessionDescription.getConnection().getAddress()*. To find the remote port and the audio codecs related to the incoming SDP information, two inner method has been developed to extract it from the SessionDescription created: *extractAudioCodecs(sessionDescription)* and *getAudioPort(sessionDescription)*. Once all information has been extracted from the incoming SDP string, the method *negotiateAudioCodecs()* is called to find the codecs availables for both clients, caller and callee.

public Object getResponseSdpBody(String incomingSdpBody) {}

Given the SDP body of the incoming call, this method builds the SDP body that will presents what codec has been chosen for the media transmission, and in which port the media will be received. The object built will be be transmitted in the SIP OK response for an incoming call.

The Jain-SDP API provides methods to construct the SDP message body. However, this method builds the SDP content "manually", using only the sdpFactory object to build the MediaDescription content. It has been developed by this way to make it compatible with IPv6. Although according to the Jain-SDP API specifications, it is compatible with IPv6, we have had problems to pass the address correctly to be understood by Asterisk, therefore, and thanks to the body content can be passed as a String, it has been written manually. See the method description *createSDPBody(int audio)* in SipManager.class to understand its function.

public void startMediaSession() {}

Depending on the audio configuration given, this method starts the communication by call the transmission and reception inner methods for the standard device or the adapter.

public void stopMediaSession() {}

Depending on the audio configuration given, this method stops the communication by call the transmission and reception inner methods for the standard device or the adapter.

The following methods has been created to perform the actions already described:

private List extractAudioCondecs(SessionDescription session) {}

Extracts all the audio codecs from the description of the media session of the incoming request and return a list containing all of them.

private String negociateAudioCodecs(List audioCodecList) {}

This method finds the best codec between our own supported codecs (*audioSupportedCodecList*) and the remote supported codecs given as input argument (*audioCodecList*), to initialize the media session. The algorithm applied to find the best codec is simply the first which belongs to both lists.

private int getAudioPort(SessionDescription session) {}

Extracts the audio port from the description of the media session of the incoming request and return the port number.

protected void startReceiving() {}

This method creates a new Session Description which is configured with the parameters already extracted by the methods handling the SDP messages. This parameters are the remote IP address, the local and remote port, and the negotiate audio codec. Once the Session Description has been configured it creates and starts a new Receive object for starting the media reception.

protected void startTransmitting() {}

This method creates a new Session Description which is configured with the parameters already extracted by the methods handling the SDP messages. This parameters are the remote IP address, the local and remote port, and the negotiate audio codec. Once the Session Description has been configured it creates and starts a new Transmit object for starting the media transmission.

protected void startReceivingAdapter(AudioConfig config) {}

This method creates a new Session Description which is configured with the parameters already extracted by the methods handling the SDP messages. This parameters are the remote IP address, the local and remote port, and the negotiate audio codec. Once the Session Description has been configured it creates and starts a new RTPtoUDP object for starting the media bridge between the RTP and UDP network interfaces. In this case the AudioConfig object has to be passed in the RTPtoUDP initialization.

protected void startTransmittingAdapter(AudioConfig config) {}

This method creates a new Session Description which is configured with the parameters already extracted by the methods handling the SDP messages. This parameters are the remote IP address, the local and remote port, and the negotiate audio codec. Once the Session Description has been configured it creates and starts a new UDPtoRTP object for starting the media bridge between the UDP and RTP network interfaces. In this case the AudioConfig object has to be passed in the UDPtoRTP initialization.

}

5.5.7 Media.Util package

This package contains the necessary classes to parse the media configuration, as the remote and host IP addresses, remote and host ports, the audio codecs, etc.

SessionDescription.java

This is an utility class created to parse the session addresses and stores the information to configure the receiver and transmitter.

public class SessionDescription() {

> **public void parseSessionDescription(String session) {}**
> Parse the session in a Session Label which contains the address, the remote port, the local port and the time to live separated by slashes.

This class also provides methods to store and get all communication parameters individually.

}

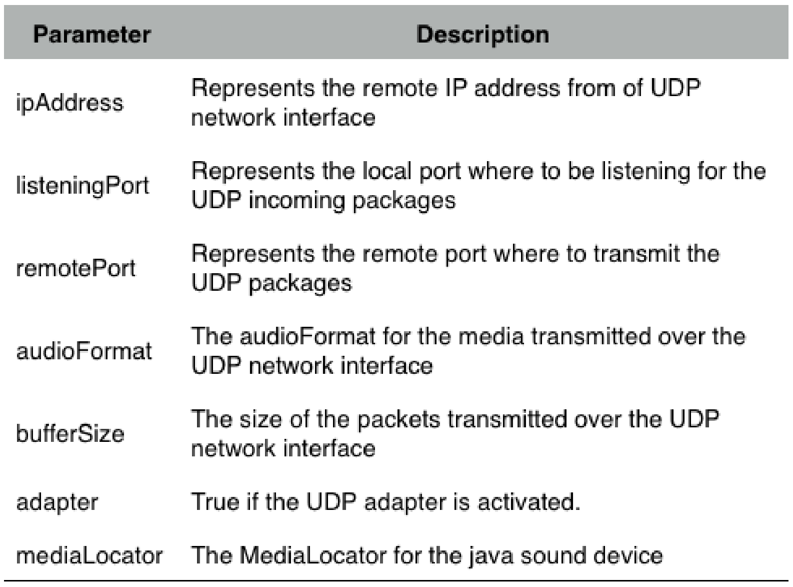
AdapterConfiguration.java

This class represents the configuration of the phone adapter, with all the parameters necessaries to establish a UDP communication over the PAN interface connected on the computer where the application is running.

The main parameters of the adapter configuration are:

5.5.8 Media.StandarDevice.Receiver package

This package is formed by the classes that implements the RTP and RTCP receiver using the Java Media Frameworks (JMF) API, and the player to listen the audio received.



Parameter	Description
ipAddress	Represents the remote IP address from of UDP network interface
listeningPort	Represents the local port where to be listening for the UDP incoming packages
remotePort	Represents the remote port where to transmit the UDP packages
audioFormat	The audioFormat for the media transmitted over the UDP network interface
bufferSize	The size of the packets transmitted over the UDP network interface
adapter	True if the UDP adapter is activated.
mediaLocator	The MediaLocator for the java sound device

Figure 5.23: SessionDescription patameters

Receive.java

This class creates a RTP communication channel configured with the parameters exchange over the SDP protocol, starts the audio data reception over RTP and decode the audio received to play it on the computer where the application is running.

public class Receive() implements ReceiveStreamListener, SessionListener,

ControllerListener {

protected boolean initialize(String localIPAddress) {}

Using the RPTManager API from JMF, it opens a RTP session with the IP address given. To maintain and close the RTP session a new instance of RTPManager is created and initialized to receive the audio data. This action is performed as follows:

```
mgrs[i] = (RTPManager) RTPManager.newInstance();

// create the local and remote endpoint port
int localPor = sessionDescription.getLocalPort() + 2*i;
int destPort = sessionDescription.getDestinationPort() + 2*i;

SessionAddress localAddr = new SessionAddress(InetAddress.getByName
(localIpAddress), localPort);

mgrs[i].addSessionListener(this);
// add the ReceiveStreamListener to receive data
mgrs[i].addReceiveStreamListener(this);
// initialize the RTPManager, so the session
mgrs[i].initialize(localAddr);

InetAddress remoteIPAddress = InetAddress.getByName(sessionDescription
.getAddress());

SessionAddress remoteDestinationAddressAndPort = new SessionAddress
(remoteIPAddress, destPort);
SessionAddress destAddr = new SessionAddress(InetAddress.getByName
(sessionDescription.getAddress()), destPort);

mgrs[i].addTarget(destAddr);
```

protected void close() {}

Close the current RTP session and close the player.

public synchronized void update(SessionEvent evt) {}

This method it is necessary to be able to catch SessionEvent to know when a new participant has joined a session.

public synchronized void update(ReceiveStreamEvent evt) {}

This method react depending on which event has been generated by the RTPSession-Listener, if the event corresponds to a NewReceiveStream, it is necessary to get the streams to create a DataSource and so, a player for the new incoming streams. Besides if the event corresponds to a ByeEvent, it has to close the player already created. Another events can be caught to know information about the media being received. Next code shows how this method has been implemented:

```
if (evt instanceof NewReceiveStreamEvent) {


   stream = ((NewReceiveStreamEvent)evt).getReceiveStream();
   DataSource ds = stream.getDataSource();

   // create a player by passing datasource to the Media Manager
   Player p = javax.media.Manager.createPlayer(ds);
   p.addControllerListener(this);
   p.realize();
```

```
        }

        else if (evt instanceof ByeEvent) {

            PlayerWindow pw = find(stream);
            pw.close();
            playerWindows.removeElement(pw);

        }
```

public synchronized void controllerUpdate(ControllerEvent evt) {}

This method is necessary to manage and control the players where the audio data received is played. It handles the media to configure and initialize the player depending on the events caught.

}

PlayerWindow.java

This class implements a minimalist GUI frame class for the Player.

PlayerPanel.java

This class implements a minimalist GUI panel class for the Player.

5.5.9 Media.StandarDevice.Transmiter package

This package contains the classes necessaries to create the RTP communication for transmitting the audio data acquired.

Transmit.java

This class creates a RTP communication channel with the other user, it acquires the audio data from the sound device where the application is running, it encodes the audio and starts the audio data transmutation over RTP.

public class Transmit {

protected boolean initialize() {}

It get the audio devices availables on the computer where the application is running, for the media capture. This action is realized as follows:

```
CaptureDeviceInfo audioCDI=null;
Vector captureDevices=null;
captureDevices= CaptureDeviceManager.getDeviceList(null);

for (int i = 0; i <captureDevices.size(); i++) {
    cdi = (CaptureDeviceInfo) captureDevices.elementAt(i);
    Format[] formatArray=cdi.getFormats();

    for (int j = 0; j <formatArray.length; j++) {
    Format format = formatArray[j];
    if (format instanceof AudioFormat)
        audioCDI=cdi;
}
audioLocator = audioCDI.getLocator();
```

public synchronized String start(String localIPAddress) {}

It creates a Processor for the specified media locator find it on the computer by the method initialize and program it to output RTP. Then, it creates the transmitter.

public void stop() {}

It stops the transmission if already started by stopping the processor already created as well as all RTPManagers.

private String createProcessor() {}

It creates the audio data source from the audio locator and it is parsed to the manager to create the processor for controlling the time-based media data acquired, and get access to the output data streams. A new instance of the StateListener is created to control the processor states. This method has been developed as follows:

```
DataSource audioDS=null;
StateListener stateListener=new StateListener();

//create the DataSource
audioDS= javax.media.Manager.createDataSource(audioLocator);
```

```
//Create the processor from the DataSource
processor = javax.media.Manager.createProcessor(audioDS);

// Wait for it to configure
stateListener.waitForState(processor, Processor.Configured);
// Set the output content descriptor to RAW RTP
ContentDescriptor cd = new ContentDescriptor(ContentDescriptor.RAW RTP);
processor.setContentDescriptor(cd);

// Program the tracks
// Get the tracks from the processor
TrackControl [] tracks = processor.getTrackControls();
for (int i = 0; i <tracks.length; i++) {
    supported = tracks[i].getSupportedFormats(); {
    for(int j = 0; j<supported.length; j++) {
    if ((supported[j] instanceof AudioFormat) && (supported[j].toString().
    toLowerCase().indexOf(sessionDescription.getAudioFormat().toLowerCase())!=-
    1))
        tracks[i].setFormat(supported[j]);
    }
}
stateListener.waitForState(processor, Controller.Realized);
dataOutput = processor.getDataOutput();
```

private String createTransmitter(String localIpAddress) {}

Getting the audio data source output from the configured processor, it uses the RTP-Manager API to create a session for the media track, and it starts the transmission to the specified IP address and port given. This method has been developed as follows:

```
PushBufferDataSource pbds = (PushBufferDataSource)dataOutput;
PushBufferStream pbss[] = pbds.getStreams();
rtpMgrs = new RTPManager[pbss.length];

for (int i = 0; i <pbss.length; i++) {
    rtpMgrs[i] = RTPManager.newInstance();

    destPort = sessionDescription.getDestinationPort() + 2*i;
    ipAddr = InetAddress.getByName(sessionDescription.getAddress());
    destAddr = new SessionAddress( ipAddr, destPort);
    localPort = sessionDescription.getLocalPort() + 2*i;
    localAddr = new SessionAddress(InetAddress.getByName(localIpAddress), lo-
    calPort);
```

```
        SessionAddress localAddress = new SessionAddress(InetAddress.
        getByName(localIpAddress),destPort);
        rtpMgrs[i].initialize(localAddress);

        SessionAddress destAddress = new SessionAddress(InetAddress.
        getByName(sessionDescription.getAddress()),destPort);

        rtpMgrs[i].addTarget(destAddress);

        sendStream = rtpMgrs[i].createSendStream(dataOutput, i);
        sendStream.start();
    }
```

StateListener.java

public class StateListener implements ControllerListener{

This interface provides a way for the application to determine what state a processor is in and to respond appropriately. Therefore, this interface has been developed to control the change states of the processor.

public void controllerUpdate(ControllerEvent evt) {}

If there is an error during the configure or the realize method of the processor, it is closed. In addition this method controls a notification to the waiting thread in *waitForState()*.

```
    if (ce instanceof ControllerClosedEvent)
        setFailed();
    if (ce instanceof ControllerEvent) {
        synchronized (getStateLock())
            getStateLock().notifyAll();
    }
```

public synchronized boolean waitForState(Processor p, int state) {}

This method is developed to configure the processor given depending on the state also given, and block the processor until it gets an event that confirms the success of the method, or a failure event.

```
        p.addControllerListener(this);
        failed = false;

        // Call the required method on the processor
        if (state == Processor.Configured)
            p.configure();
        else if (state == Processor.Realized)
            p.realize();
        while (p.getState() <state && !failed){
            synchronized (getStateLock())
                getStateLock().wait();
        }
    }
```

5.5.10 Media.UDPAdapter.Receiver package

This package contains the classes necessaries to create and maintain an RTP communication, as well as its treatment and its consequent transmission over the UDP communication.

RTPtoUDP.java

This class creates a RTP communication channel with the other user, it acquires the audio data from the listening UDP port, decodes this data and it is encoded to the specified format to finally, transmit this audio data over the RTP channel opened to the other user.

public class RTPtoUDP implements ReceiveStreamListener, SessionListener,

ControllerListener, DataSinkListener {

public boolean initialize(String localIPAddress) {}

Using the RPTManager API from JMF, it opens a RTP session with the IP address given. To maintain and close the RTP session a new instance of RTPManager is created and initialized to receive the audio data. Its implementation is realized as the same method described in the Receive.java.

public void close() {}

Close the Processors and the RTPManagers to finish the session.

public synchronized void update(SessionEvent evt) {}

The method update is necessary to implement the SessionListener and it catches the session events generated by the RTPSessionManager. The only one event that we need to care about is when a new participant is joined.

public synchronized void update(ReceiveStreamEvent evt) {}

The method update is necessary to implement the ReceiveStreamListener for the RTP channel management, and it catches the stream events generated by the channel. When a new ReceiveStreamEvent is generated this method gets the DataSource from the event, it creates a Processor to configure the audio content received to a RAW format and sets the audio data output from the Processor to the input of the DataSourceHandler object created to be transmit over the UDP communication. This method have been implemented as follows:

```
//Pull out the stream
stream = ((NewReceiveStreamEvent)evt).getReceiveStream();
DataSource ds = stream.getDataSource();

p = Manager.createProcessor(ds);
p.addControllerListener(this);
// Put the Processor into configured state.
p.configure();
waitForState(p.Configured);

// Get the tracks from the processor
TrackControl [] tracks = p.getTrackControls();
// Get the raw output from the processor.
p.setContentDescriptor(new ContentDescriptor(ContentDescriptor.RAW));

// Program the tracks.
for (int i = 0; i <tracks.length; i++) {
    if (tracks[i].isEnabled())
        tracks[i].setFormat (config.getAudioFormat());
}
p.realize();
waitForState(p.Realized);

// Get the output DataSource from the processor and
// hook it up to the DataSourceHandler.
```

```
DataSource ods = p.getDataOutput();
DataSourceHandler handler = new DataSourceHandler(config);

handler.setSource(ods);
handler.addDataSinkListener(this);
handler.start();

// Prefetch the processor.
p.prefetch();

waitForState(p.Prefetched)

// Start the processor.
p.start();
```

public void controllerUpdate(ControllerEvent evt) {}

This method is necessary to manage and control the Processor state transitions. Therefore this method controls the notification to the waiting thread in *waitForState()*. It is implemented as follows:

```
if (evt instanceof ConfigureCompleteEvent ||evt instanceof RealizeCompleteEvent
||evt instanceof PrefetchCompleteEvent) {
    synchronized (waitSync) {
        stateTransitionOK = true;
        waitSync.notifyAll();
    }
} else if (evt instanceof ResourceUnavailableEvent) {
    synchronized (waitSync) {
        stateTransitionOK = true;
        waitSync.notifyAll();
    }
} else if (evt instanceof EndOfMediaEvent)
    p.close();
```

public void dataSinkUpdateData(DataSinkEvent evt) {}

This method is necessary to implement the DataSinkLsitener, and it catches the events generated by the DataSink interface. When a EndOfStream event is received it close the DataSink source.

```
if (evt instanceof EndOfStreamEvent)
```

```
        evt.getSourceDataSink().close();
```

boolean waitForState(int state) {}

This method is developed to block the Processor until it gets an event that confirms the success of the method, or a failure event. The Processor stages are managed by the method *controllerUpdate()*.

```
    synchronized (waitSync) {
        while (p.getState() <state && stateTransitionOK)
            waitSync.wait();
    }
```

}

5.5.11 Media.UDPAdapter.Transmiter package

UDPtoRTP.java

This class creates a RTP communication channel with the other user, it acquires the audio data from the listening UDP port, decodes this data and it is encoded to the specified format to finally, transmit this audio data over the RTP channel opened to the other user.

public class UDPtoRTP implements ControllerListener {

public synchronized String start(String localIpAddress) {}

Creates a Processor by calling the method *createProcessor()*, a RTP session by calling the method *createTransmitter()* and starts the transmission.

public void stop() {}

Stops the Processor and the RTPManager if already started.

private String createProcessor() {}

This method creates and connects the audio data source from the UDPDataSource interface through a Processor, and it configures the output content of the Processor to be able to send it over RTP. Its implementation is exactly the same explained in the Transmit.java but, instead of creating the DataSource from an AudioLocator, now, this method creates a processor from the custom UDPDataSource which reads the audio data received from an specific port. Therefore, to create a Processor, first the UDPDataSource must be created and connected.

```
//create the 'audio' DataSource
UDPDataSource ds = new UDPDataSource(config);
// connects the UDPDataSource
ds.connect();

//Create the processor from the 'audio' DataSource
processor = javax.media.Manager.createProcessor(ds);
```

private String createTransmitter(String localIPAddress) {}

Getting the audio data source output from the configured processor, it uses the RTP-Manager API to create a session for the media track, and it starts the transmission to the specified IP address given. Once the processor has been created from the custom UDPDataSource, the next steps to initialize and starts transmitting it over RTP are exactly the same as described in the Transmit.java class description.

public void transmit(String localIpaddress) {}

Starts the transmission by calling the method start, in addition if there was an error it will return a string describing the possible error.

public void controllerUpdate(ControllerEvent ce) {}

This method is necessary to manage and control the Processor state transitions. If there was an error during the configure or realize transition, the Processor will be closed. On the other hand, this method controls the notification to the waiting thread in *waitForState()*. It has been implemented as follows:

```
if (ce instanceof ControllerClosedEvent)
    failed = true;
if (ce instanceof ControllerEvent) {
    synchronized (stateLock)
        stateLock.notifyAll();
}
```

public synchronized boolean waitForState(Processor p, int state) {}

This method is developed to configure the processor given depending on the state also given, and block the processor until it gets an event that confirms the success of the method, or a failure event. Its implementation is already explained in the StateListener.java.

}

5.5.12 Media.Protocol package

This interfaces has been developed to allow the SIP VoIP Adapter be able to use an End Device, situated in another interface, as a Sound Device to play and acquire the SIP conversation. In our case, this classes implements a communication through a network over UDP.

If the communication between the End Device and the SIP VoIP Adapter is conducted through a different interface as the communication port, this classes must be re-written.

DataSourceHandler.java

This class reads the data-streams from an output of a Processor and transmits this audio data over a network interface connected on the same computer where the application is running, using the transport protocol UDP. DataSourceHandler implements the interface DataSink which reads the data-streams given and stores it into a Buffer. In addition, to be able to send the data stored in the Buffer over the network, it also implements the interface BufferTransferHandler.

public class DataSourceHandler implements DataSink, BufferTransferHandler{

public void setSource(DataSource source) {}

It sets the media source used to obtain the content. This method handles only the source instances PushBufferDataSource and it sets the transfer handler to receive pushed data from the push DataSource. This method has been developed as follows:

```
// Different types of DataSources need to handled differently
if (source instanceof PushBufferDataSource) {
    pushStrms = ((PushBufferDataSource)source).getStreams();
    unfinishedStrms = new SourceStream[pushStrms.length];

    // Set the transfer handler to receive pushed data from
    // the push DataSource.
    for (int i = 0; i <pushStrms.length; i++) {
        pushStrms[i].setTransferHandler(this);
        unfinishedStrms[i] = pushStrms[i];
    }
} else
```

```
        throw new IncompatibleSourceException();
    this.source = source; readBuffer = new Buffer();
```

public void start() {}

This method starts the transfer of the data source set it.

```
    source.start()
```

public void stop() {}

This method stops the transfer of the data source set it.

```
    source.stop()
```

protected void sendEvent(DataSinkEvent event) {}

This method is developed to perform the event communication with the interface that implements the DataSinkListener (RTPtoUDP.java).

```
    if (!listeners.isEmpty()) {
        synchronized (listeners) {
            Enumeration list = listeners.elements();
            while (list.hasMoreElements()) {

            DataSinkListener listener = (DataSinkListener) list.nextElement();
            listener.dataSinkUpdate(event);
            }
        }
```

public void transferData() {}

This method is called when there is data pushed from the PushBufferDataSource. It reads the streams pushed, stores this streams into a Buffer object, and call the method *sendOverUDP()*. Besides, this method controls and generates an event when an end of stream is received. This method has been developed as follows:

```
    stream.read(readBuffer);
    sendOverUDP(readBuffer);

    // Check to see if we are done with all the streams.
    if (readBuffer.isEOM() && checkDone(stream))
        sendEvent(new EndOfStreamEvent(this));
```

public void sendOverUDP() {}

Given a Buffer object, this method converts a given Buffer object to a byte array, it creates a datagram packet from the byte array and sends the datagram over the socket.

```
        if(buffer.getData() instanceof byte[]) {
            buf=(byte[])buffer.getData();
            //packets transmitted counter
            count++;
            //bytes transmitted counter
            datacount+=buf.length;

            //Send buf over UDP to the UDPAddress and UDPport
            DatagramPacket datagram = new DatagramPacket(buf, buf.length, InetAd-
            dress.getByName(config.getIPAddress()), config.getRemotePort());

            socket.send(datagram);
        }

}
```

UDPDataSource.java

This class is the responsible to manage the transfer of media-content, it encapsulates both the location of media and the protocol and software used to deliver the media. Once obtained, a Processor is able to be configured with it to starts the RTP communication.

public class UDPDataSource extends PushBufferDataSource{

public void connect() {}

This method connects the DataSource allowing a Processor to start getting the audio-content that this source manages.

public void disconnect() {}

Disconnects the DataSource after the Processor who manages it, has already ended.

public void start() {}

This method starts reading the incoming UDPInputStreams.

```
 if (started)
     return;
started = true;
stream.start(true);
```

public void stop() {}

This method stops reading the incoming UDPInputStreams.

```
if (!connected) ||(!started))
    return;
started = false;
stream.start(false);
```

public PushBufferStreams getStreams() {}

This method gets a SourceStreams by creating a UDPInputStream interface object to receive the audio data from the UDP socket.

```
if (streams == null) {
    streams = new UDPInputStream[1];
    stream = streams[0] = new UDPInputStream(config);
}
return streams;
```

}

UDPInputStream.java

This class is the responsible to manage the audio data received from a socket UDP and stores the data received in datagrams into a Buffer object as streams.

public class UDPInputStream implements PushBufferStreams, Runnable{

public void read(Buffer buffer) {}

It creates a datagram where receive the audio data from a socket UDP, and stores the data received into the Buffer object handled by this method, besides it is necessary to set the buffer format and length as well as the header and the flags of the Buffer object. This method has been implemented as follows:

```
// Create a packet to receive data into the buffer
packet = new DatagramPacket(data, maxDataLength);

dsocket.receive(packet);
```

```
//UDP transmition statics
counter++;
size+=packet.getData().length;

//Add buffer object properties
buffer.setData(packet.getData());
buffer.setFormat(audioFormat);
buffer.setLength(packet.getData().length);
buffer.setFlags(0);
buffer.setHeader(null);
```

public void setTransferHandler(BufferTransferHandler transferHandler) {}

This method is needed to notify the data handler when data is available to be pushed.

```
synchronized (this) {
    this.transferHandler = transferHandler;
    notifyAll();
}
```

void start(boolean started) {}

This method start or stops the reading thread depending on the given started.

```
synchronized (this) {
    this.started = started;
    if (started && !thread.isAlive()) {
        thread = new Thread(this);
        thread.start();
    }
    notifyAll();
}
```

public void run() {}

While the UDPInputStream has already started, a thread is created to manage when the data is available to be read from stream. The runnable has been developed as follows:

```
while (started) {
synchronized (this) {
    while (transferHandler == null && started)
        wait(1000);
}
if (started && transferHandler != null) {
```

```
                transferHandler.transferData(this);
                Thread.currentThread().sleep( 10 );
            }
            }


}
```

5.6 Summary

A project class description was given for going deeper into the SIP VoIP Adapter implementation. This chapter has also discussed the main methods of each class to understand its functionallity.

Chapter 6

Design and Implementation of EndDevice Test

6.1 Introduction

This chapter covers the functionality and class design of the program developed to test the SipPhone Adapter. EndDevice Test is a Java application that handles the media data received from specific IP interface (it uses the transmission protocol UDP), decodes and plays the audio data. It also acquire the data from the computer audio device where the application is running, encodes and sends this audio data over the UDP interface.

This application has been created to simulate a sound device connected on the PAN interface of the computer where the SipPhone Adapter is running. The Transmition protocol UDP is used to communicate the PAN interface and the SipPhone Adapter, without implementing the RTP layer.

6.2 EndDevice Test Application Functionality and Interface (High Level View)

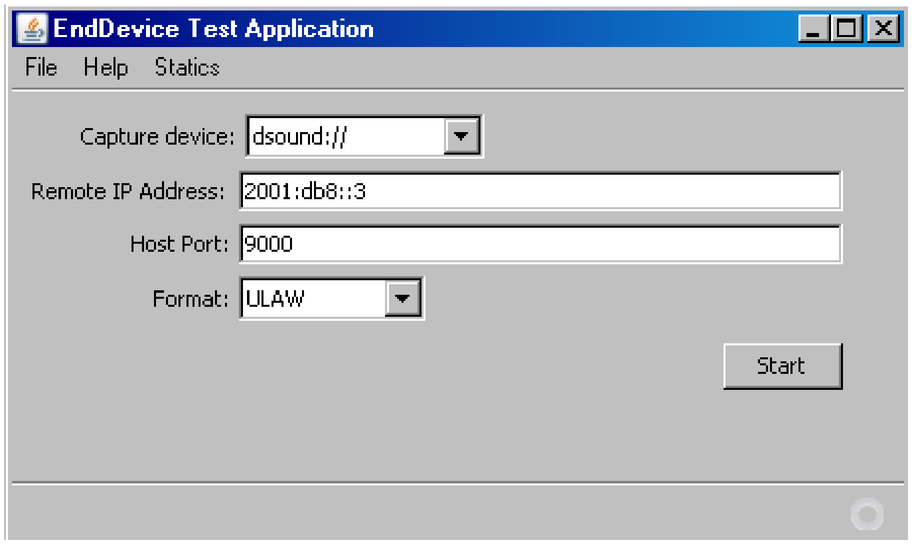
This section contains a high level view of the EndDevice Test in order to summarize their functionality.

6.2.1 Graphical User Interface (GUI)

EndDevice Test main window (enddevice)

Figure 6.1 shows the EndDevice Test main window. Because of the minimalist design, the main window only content the EndDevice configuration necessary to be able to establish a UDP communication over the network interface.

The input parameter necessary to start a full-duplex UDP communication are the port where the application is going to receive the audio data from the SipPhone Adapter and the IP address and port where the audio data acquired is going to be transmitted. The Capture Device to use on the computer where the application is running (EndDevice Test is going to acquire all devices available on the computer) and the audio codec used in the UDP communication. The EndDevice Test application is going to detect the host IP address.



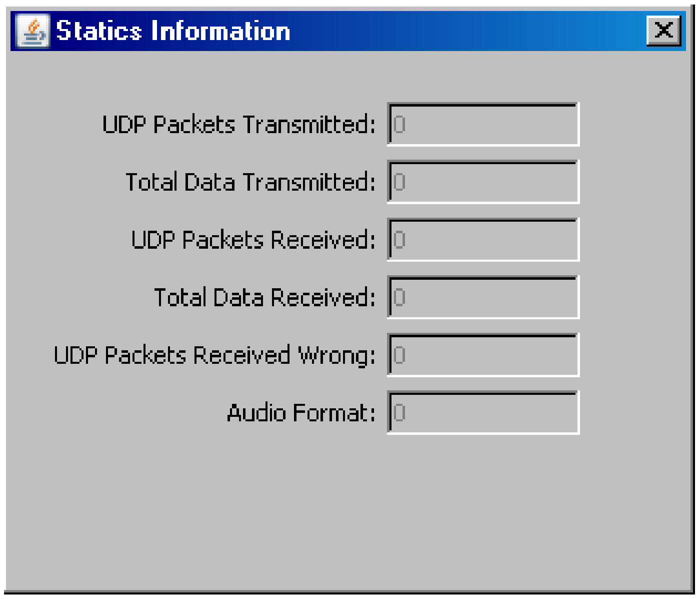
Capture device: dsound://
Remote IP Address: 2001:db8::3
Host Port: 9000
Format: ULAW

Figure 6.1: EndDevice Test application main window

Statics Information window (StaticsDialog)

Figure 6.2 shows the Statics Information window where it is possible to see information about the UDP communication between the EndDevice Test Application and the SipPhone Adapter. This class also implements the *Observer* interface, which uses the method *update()* to set the changes happened on the model to update the windows view.

The user can get access to the statics information by selecting the option "Show Statics" in the statics menu in the main window.


Statics Information

UDP Packets Transmitted:	0
Total Data Transmitted:	0
UDP Packets Received:	0
Total Data Received:	0
UDP Packets Received Wrong:	0
Audio Format:	0

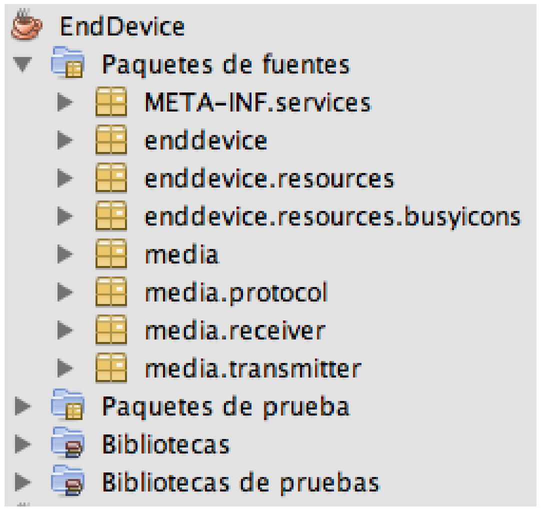
Figure 6.2: EndDevice Test application statics window

6.3 EndDevice Test Class Design

The EndDevice Test goal was to create a set of basic classes that could perform a minimalist media communication with the SipPhone Adapter, using only the standard UDP transmition protocol. Listening a defined port to play the data received from and transmitting the audio data acquired from the microphone over an specific IP address and port. Figure 6.3 shows how the packages are divided.

6.3.1 Media package

This package contains the necessary classes to parse the media configuration, as the remote and home IP addresses, remote and host ports, the audio codecs, etc. It also contents the class which stores the data statistics.



```
EndDevice
▼ Paquetes de fuentes
    ▶ META-INF.services
    ▶ enddevice
    ▶ enddevice.resources
    ▶ enddevice.resources.busyicons
    ▶ media
    ▶ media.protocol
    ▶ media.receiver
    ▶ media.transmitter
▶ Paquetes de prueba
▶ Bibliotecas
▶ Bibliotecas de pruebas
```

Figure 6.3: EndDevice Test Application package organization

AudioConfig.java

This class represents the configuration of the phone adapter, with all the parameters necessaries to establish a UDP communication over the PAN interface connected on the computer where the application is running.

Statics.java

This class represents the model part in the MVC pattern for this application, therefore when there is a change in the model (some "set" method is called) it notifies the registered view to be updated. So, this class implements the interface pattern *Observable* and manage all the statics of the UDP communication.

6.3.2 Media.receiver package

This package includes all necessary classes to receive audio data from a socket UDP and play the data received.

UDPreceiver.java

This class, as well as the transmitter Adapter class from the SipPhone Adapter, it uses the inner class UDPDataSource which given a computer port where to be listening for the incoming data, it creates a DataSource. In addition, JMF technology lets us to create a processor for this data source, to be able to play it on the computer where the application is running. Therefore, the UDPreceiver class has only to use the Manager, given by the JMF library, and the GUI player interface created, to be able to reproduce the audio data received.

public class UDPreceiver implements ControllerListener(){

> **public void initialize(){}**
>
> Initialize the UDPDataSource and creates a Player to process the input streams of media-stream delivered by the UDPDataSource. A controllerListener is added to the processor Player for handling the events generated by the object and it is controlled by the method *controllerUpdate()*.
>
> **public void controllerUpdate(){}**
>
> UDPreceiver implements the interface Controllerlistener, therefore the update method has to be implemented to update the GUI in response to changes in the Player object's created.
>
> **public void close(){}**
>
> it is necessary to play back multiple RTP streams in a session. SessionManager gets construct a Player for each ReceiveStream. Construct a Pleyer for a particular ReceiveStream is possible by retrieving the DataSource from the stream and passing it to *Manager.createPlayer(DataSource)*.

}

PlayerWindow.java

This class implements a minimalist GUI frame class for the Player.

PlayerPanel.java

This class implements a minimalist GUI panel class for the Player.

6.3.3 Media.transmitter package

This package includes all necessary classes to acquire audio data from the microphone where the application is running and send the audio acquired over a socket UDP.

UDPtransmitter.java

This class, as well as the receiver Adapter from the SipPhone Adapter, it uses the inner class DataSourceHandler, which given the DataSource output from a Processor it transmit the audio data over the network using UDP transport protocol. In addition, it encodes the audio acquired from the microphone where the EndDevice Test application is running to the specified format in the main window.

public class UDPtransmitter implements ControllerListener(){

public void start(){}

This method is used to starts the transmission and handles the error notifications.

public void stop(){}

Stops the transmition if already started by closing the Processor.

public void createTransmitter(){}

Given a MediaLocator where acquiring the audio data from a sound device selected on the main window, this class creates a Processor object to perform a format conversion specifying the format of the data output by the Processor. In addition, it handles this data output and hook it up to the DataSourceHandler to transmit the audio data over UDP.

public void controllerUpdate(){}

UDPtransmitter implements the interface Controllerlistener, therefore the update method has to be implemented to control the transition states of the Processor.

public void dataSinkUpdate(){}

UDPtransmitter also implements the interface DataSinkListener, therefore the update method has to be implemented to control the DataSink stop when an EndOfStream is received.

}

6.3.4 Media.protocol package

DataSourceHandler.java

To summarize, this class reads the data-streams from an output of a Processor and transmits this audio data over a UDP socket. For more information see chapter 5.5.12. On the other hand, although all methods are exactly the same as described on chapter 5, in this case there is a variation in method *sendOverUDP()* which uses the object Statics to maintain the statics of the communication.

UDPDataSource.java

To summarize, this class translate the data received from an UDP socket to a format that JMF can recognize. For more information see chapter 5.5.12.

UDPInputStream.java

This class is the responsible to manage the audio data received from a socket UDP and stores the data received in datagrams into a Buffer object as streams. For more information see chapter 5.5.12. On the other hand, although all methods are exactly the same as described on the chapter 5, in this case there is a variation in the method *read()* which uses the object Statics to maintain the statics of the communication.

6.4 Summary

A project class description was given for going deeper into the EndDevice Test implementation. This chapter has also discussed the main methods of each class to understand its functionallity.

Chapter 7

Asterisk PBX

7.1 Introduction

This chapter covers the installation of Asterisk as well as its configuration to be able to establish a SIP communication between the SipPhone Adapter and other VoIP phone situated on the same local area network (LAN). It also covers the configuration of the LAN. A brief introduction of Asterisk PBX and its potencial it is also presented and.

7.2 Asterisk: A brief introduction

Asterisk is an open source software implementation of a telephony private branch exchange (PBX) created in 1999 by Marc Spencer. Like any PBX, it allows attached telephones to make calls to another, and connect to other telephone services including the public switched telephone network (PSTN) and Voice over Internet Protocol (VoIP) services.

Asterisk supports a wide range of Voice over IP protocols, including SIP, MGCP and H.323. And it can interoperate with most SIP phones, acting both as registrar and as a gateway between IP phones and the PSTN. Asterisk also provides standard feature applications such as voicemail, hosted conferencing, call queuing and agents, music on hold, and call parking.

7.3 Overview

The created system to establish a SIP communication between the SipPhone Adapter and a VoIP phone over an Asterisk PBX is formed by:

- Computer with Fedora 10 as Operative System (OS) working as a server, and Asterisk PBX installed.
- 2 computer with Windows XP as OS and a sound card installed, workings as client and SIP VoIP Adapter and SIP Communicator installed.
- A router with 4 or more connections to build the LAN.
- End Device connected on the personal area network (PAN) where the application SipPhone Adapter is running.

Figure 7.1 shows the design of the system configuration.

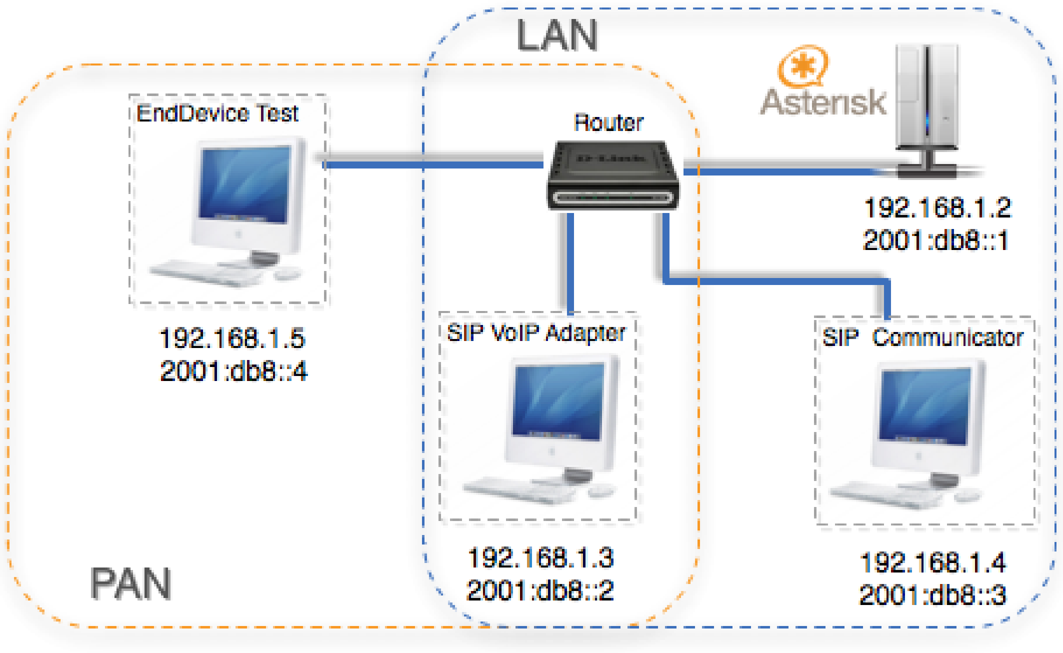


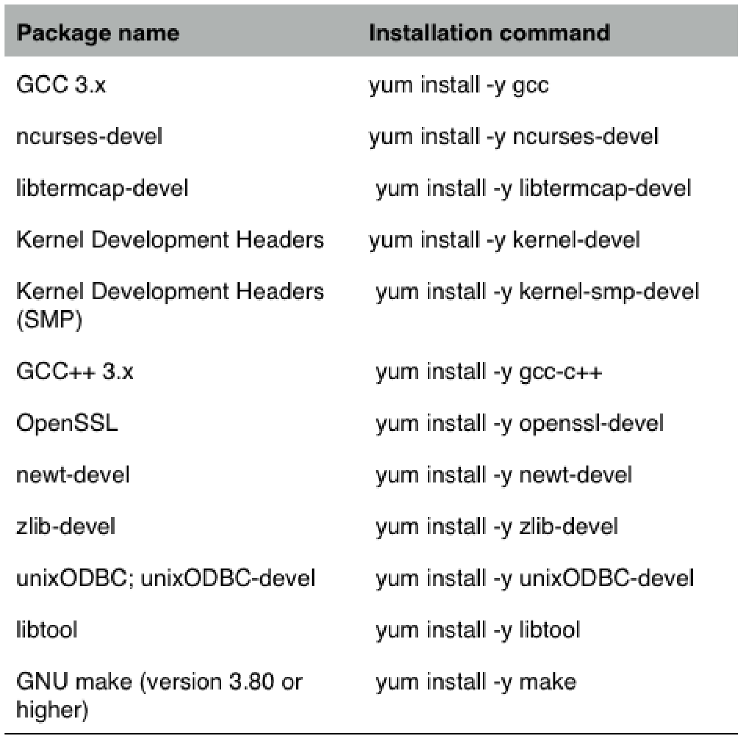
Figure 7.1: Network configuartion

7.4 Asterisk Installation

The computer where Asterisk is installed works with a Operating System Fedora 10 kernel Linux 2.6.27.5-17.fc10.i686, with GNOME version 2.24.1.

7.4.1 Packages Installed

The figure 7.2 shows the packages necessaries to be installed on the computer to be able to compile Asterisk, as well as the instruction commands to do it.



Package name	Installation command
GCC 3.x	yum install -y gcc
ncurses-devel	yum install -y ncurses-devel
libtermcap-devel	yum install -y libtermcap-devel
Kernel Development Headers	yum install -y kernel-devel
Kernel Development Headers (SMP)	yum install -y kernel-smp-devel
GCC++ 3.x	yum install -y gcc-c++
OpenSSL	yum install -y openssl-devel
newt-devel	yum install -y newt-devel
zlib-devel	yum install -y zlib-devel
unixODBC; unixODBC-devel	yum install -y unixODBC-devel
libtool	yum install -y libtool
GNU make (version 3.80 or higher)	yum install -y make

Figure 7.2: List of packages required

7.4.2 Downloading Asterisk

To download the latest distribution of Asterisk, go to The Asterisk port to IPv6 project website in: http://www.asteriskv6.org. Click on Download the latest asterisk-v6, a new web page is going to be opened, fill the form and an e-mail containing a download link is going to be send to the address given. Click on the link to download the compressed file.

7.4.3 Extracting the Source Code

The Asterisk package downloaded is a compressed file containing the source code so the user needs to extract it before compiling. If the package has not been download to /usr/src/, either move it there now or specify the full path to its location. It is necessary to use the GNU tar applications to extract the source code from the compressed archive. This is a simple process that can be achieved through the use of the following commands:

```
# cd /usr/src/
# tar zxvf asterisk-xxxx
```

7.4.4 Menuselect

In the IPv6 version of Asterisk and its related packages has a new build system implemented, autoconf, which lets more flexibility to control what modules are being built at built time. This has an advantage in that we only have to build the modules we want and need instead of building everything.

You can get access to the menuselect screen by running the command:

```
# make menuselect
```

After you have finished making changes to menuselect, type x to save and quit. q will also quit out of menuselect, but it will not save the changes.

7.4.5 Compiling and Installing Asterisk

Asterisk is compiled with gcc through the use of the GNU make program. To get started compiling Asterisk, simply run the following commands:

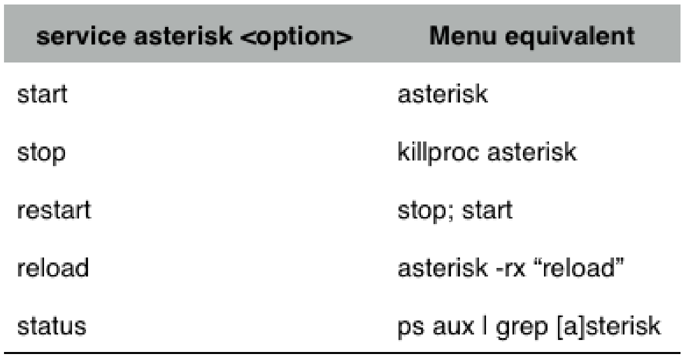
```
# cd /usr/src/asterisk-xxxx
# make clean
# ./configure
# make menuselect
# make install
# make samples
# make config
```

The command make samples install the default configuration files. This will allow you to get your Asterisk system up and running much faster instead of manual configuration. On the other hand, the make config command is necessary to be applied if the system makes use of the /etc/rc.d/init.d or /etc/init.d/ directories, and Fedora 10 do it.

7.4.6 Loading Asterisk

Once the command make config has been ran in the Asterisk source directories, the initialization scripts used to control Asterisk will be copied to /etc/rc.d/init.d/. The scripts can be used to easily load and unload Asterisk. It will also run *chkconfig* command for you so Asterisk will be started automatically upon system boot.

Figure 7.3 shows the several option that can be utilized to control the PBX.



service asterisk <option>	Menu equivalent
start	asterisk
stop	killproc asterisk
restart	stop; start
reload	asterisk -rx “reload”
status	ps aux l grep [a]sterisk

Figure 7.3: Asterisk initialization scripts options

7.5 Asterisk Configuration

7.5.1 Adding Clients

The new clients SIP configuration in the Asterisk system is done it by the sip.conf file. This file lets us to create a sip register for each client. The sip.conf file is composed by two specific fields, it starts by section [general], which contains the channel parameters (as the listening port, the codecs, etc.) and the default options in common by all users. The second section defines different sip account parameters for the users.

The first part informs the system on the default settings to all accounts:

- **Port:** defines the port used by the signaling protocol sip.
- **Disallow:** defines the codecs that it will not be used.
- **Allow:** defines the codecs that it will be used.
- **Callerid:** defines the default sip register identity.
- **Context:** defines the default content.
- **Bindadr:** IP address on which Asterisk is attached.

The second part lets define each sip account attributes. Many options lets define and characterize a client:

- **Type:** it lets gets the rights to the user. Three types are defined: peer, user or friend.
- **Username:** user name associate.
- **Secret:** user password.
- **Host:** client finder method (dynamic, host name or IP address).
- **Callerid:** user identification which it has to be attached to do a call.
- **Language:** user default language.
- **Nat:** the user is behind a NAT.

It is important to explain some parameters description so, for each element above, several values are available depending on the configuration that users wants.

Type:

- **Peer:** the Peer type can receive calls but it can not make it.
- **User:** the User type can make calls but it can not receive it.
- **Friend:** the Friend type includes both, User and Peer: it can make and receive calls.

Host:

- **Dynamic:** the client is registered to the server.
- **Host name:** client host name.
- **Static:** client IP address.

The Figure 7.4 shows the sip.conf file used in our system, it registers two clients on the network builded to test the SipPhone Adapter.

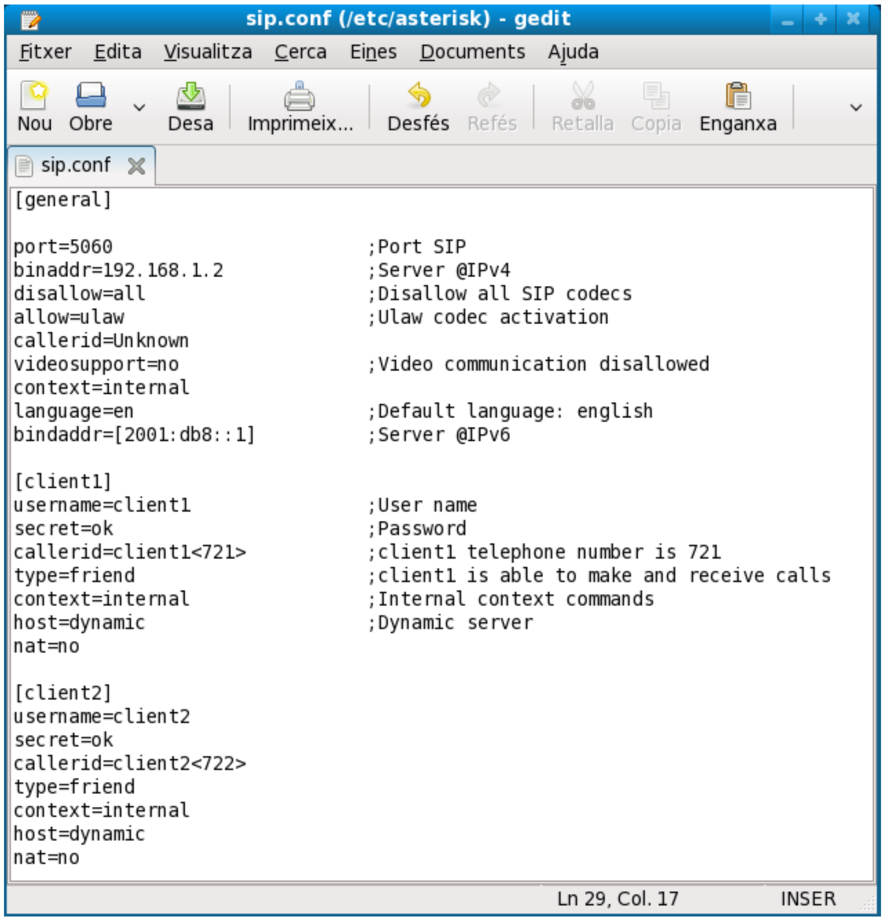
7.5.2 Telephone Numbers Allocation

The fact we have created the users on the server, does not mean that we are able to establish a communication. It is necessary to define the behavior of Asterisk when a number is typed. The rules to follow when a call is treated is defined on *extensions.conf* file.

The syntax used by extensions.conf is:

Exten =><extension>,<priority number>,<command>

- **<extension>:** is the label of the extension, and can be either a literal constant string (alphanumeric plus a few special symbols allowed) or a dynamically evaluated pattern.
- **<priority number>:** It is just a sequence number for each command line of an extension. The first executable command of an extensions has the priority ”1”, so when Asterisk transfers a call to an extension, it will look for a command with priority 1. If there is not a line with a priority of 1, then the extension will not match the dialed number. After executing the priority 1 command, Asterisk will increment the priority to ”2” unless the command itself determines the next priority to be executed.


sip.conf (/etc/asterisk) - gedit
Fitxer Edita Visualitza Cerca Eines Documents Ajuda
Nou Obre Desa Imprimeix... Desfés Refés Retalla Copia Enganxa
sip.conf
Ln 29, Col. 17 INSER

```
[general]

port=5060                       ;Port SIP
binaddr=192.168.1.2             ;Server @IPv4
disallow=all                    ;Disallow all SIP codecs
allow=ulaw                      ;Ulaw codec activation
callerid=Unknown
videosupport=no                 ;Video communication disallowed
context=internal
language=en                     ;Default language: english
bindaddr=[2001:db8::1]          ;Server @IPv6

[client1]
username=client1                ;User name
secret=ok                       ;Password
callerid=client1<721>           ;client1 telephone number is 721
type=friend                     ;client1 is able to make and receive calls
context=internal                ;Internal context commands
host=dynamic                    ;Dynamic server
nat=no

[client2]
username=client2
secret=ok
callerid=client2<722>
type=friend
context=internal
host=dynamic
nat=no
```

Figure 7.4: sip.conf

- **<command>:** It is the name of the command (also called an "application") to execute.

The following configured options are the chosen ones to manage our system:

- **Verbose([<level>|]<message>):** Send arbitrary text to verbose output. Level must be an integer value. If not specified, defaults to 0.
- **Dial(type/identification[timeout][options][url]):** Attempts to establish a new outgoing connection on a channel, and then link it to the existing input channel. Type specifies the channel type and identifier specifies the "phone number" to dial on that channel. The timeout specifies a maximum time, in seconds, that the Dial command is to wait for a channel to answer. The options parameter, which is optional, is a string containing zero or more of the following flags and parameters:
  - **t:** Allow the called user to transfer the call by hitting the blind xfer keys (features.conf).
  - **T:** Allow the calling user to transfer the call by hitting the blind xfer keys (features.conf).
- **Hand up():** Unconditionally hangs up a given channel by returning -1 always.

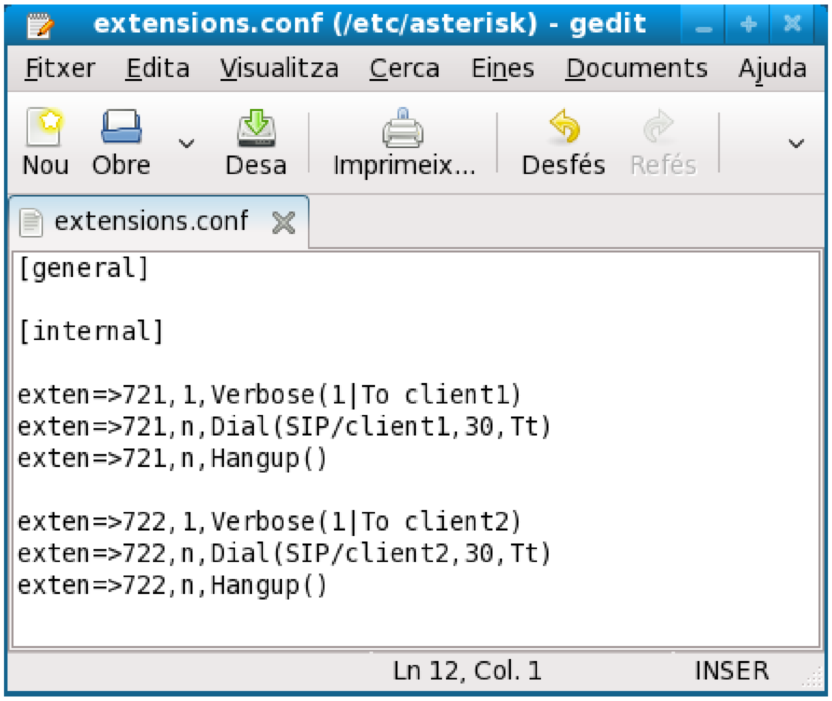
Figure 7.5 shows the extensions.conf file used in our system.

7.6 Network Configuration

An Asterisk installation and configuration has been given for handling a SIP communication between two SIP end clients (aka User Agent). Thus, due to the configuration already explained, the computer running Fedora where Asterisk is running must have the following IP configuration: address 192.168.1.2 for IPv6 and address 2001:db8::1 for IPv6. The computer acting as a SIP clients must have configured the IP addresses describes in Figure 7.6:

7.7 Summary

A primary description of Asterisk was given to help the reader before going deeper into the installation and its subsequent configuration. This chapter discusses the configuration settings to be applied in Asterisk for managing a VoIP communication between two end clients (aka User Agents).


extensions.conf (/etc/asterisk) - gedit
Fitxer Edita Visualitza Cerca Eines Documents Ajuda
Nou Obre Desa Imprimeix... Desfés Refés
extensions.conf
Ln 12, Col. 1 INSER

```
[general]

[internal]

exten=>721,1,Verbose(1|To client1)
exten=>721,n,Dial(SIP/client1,30,Tt)
exten=>721,n,Hangup()

exten=>722,1,Verbose(1|To client2)
exten=>722,n,Dial(SIP/client2,30,Tt)
exten=>722,n,Hangup()
```

Figure 7.5: extensions.conf

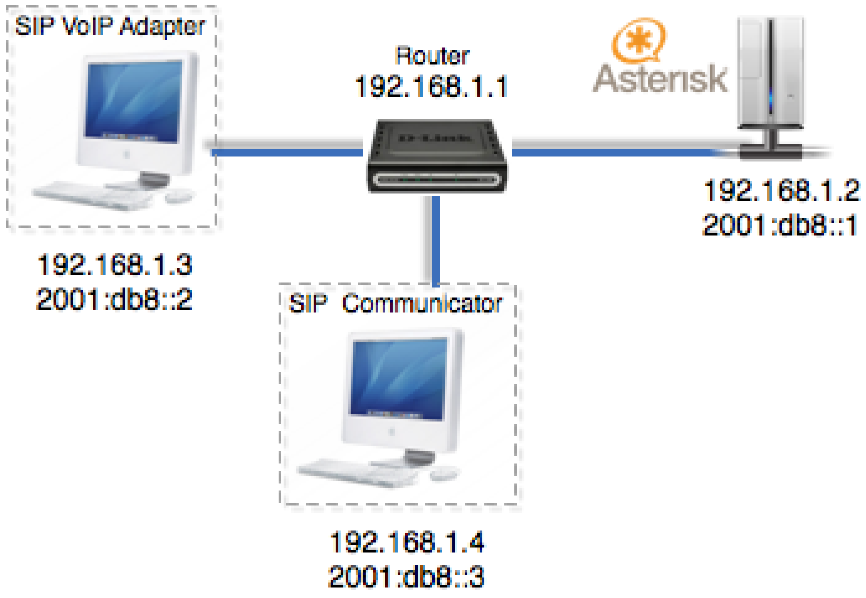


Figure 7.6: Network Configuration

Chapter 8

Experimental Results

8.1 Introduction

This chapter describes the system configuration for testing the SIP VoIP Adapter, in a communication through Asterisk PBX in both protocols IPv4 and IPv6. In addition, it shows the test results of mainteining an audio call with another SIP client, as well as the correct audio transmission to the EndDevice Test used as a sound device.

8.2 Overview

As we discussed in the previous chapter, see 7.6, the LAN configuration used to test the SIP VoIP Adapter is shown Figure 8.1. In addition, a system configuration's abstraction is shown in Figure 8.2 to understand the data exchange between computers and the protocols involved in that transmissions.

To be able to carefully analyze the packets exchanged between the 4 elements involved in the system: SIP VoIP Adapter EndDevice Test, Asterisk and other SIP client (SIP Communicator). The Wireshark network protocol analyzer has been used.

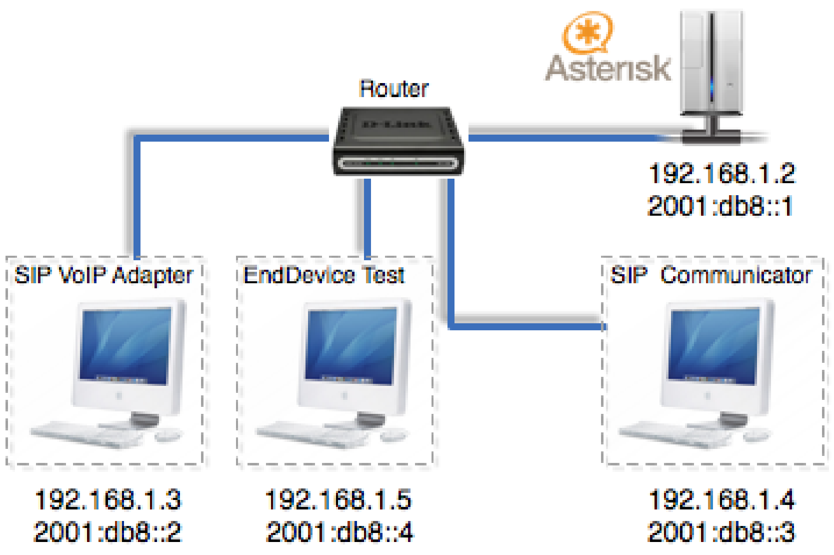


Figure 8.1: Local Area Network configuartion

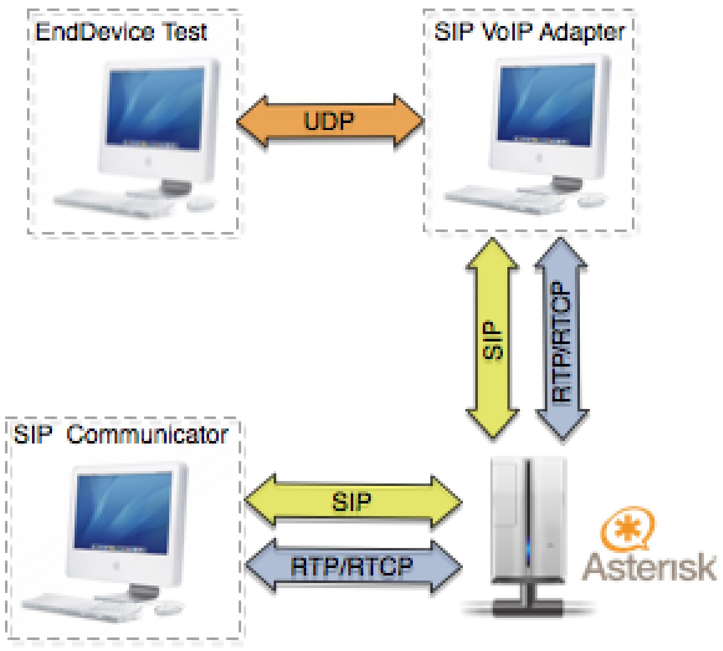


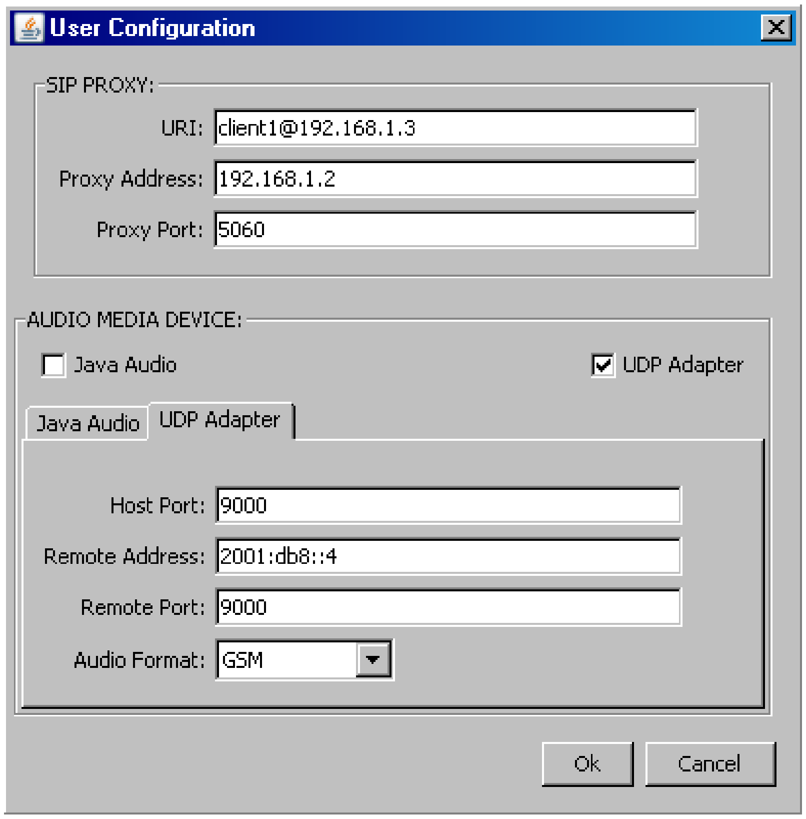
Figure 8.2: System data exhange over the network

8.3 Configuration

In this section, the reader is introduced to the parameters configuration of each application to test the Sip VoIP Adapter.

8.3.1 SIP VoIP Adapter

The SIP VoIP Adapter has been installed in the computer with address IPv4 192.168.1.3 and address IPv6 2001:db8::2. Figure 8.3 and Figure 8.4 shows the parameters used to configure the SIP VoIP Adapter to be registered on Asterisk PBX, in IPv4 and IPv6 respectively. In addition, it also shows the parameters configuration to use the EndDevice Test application as external sound device.



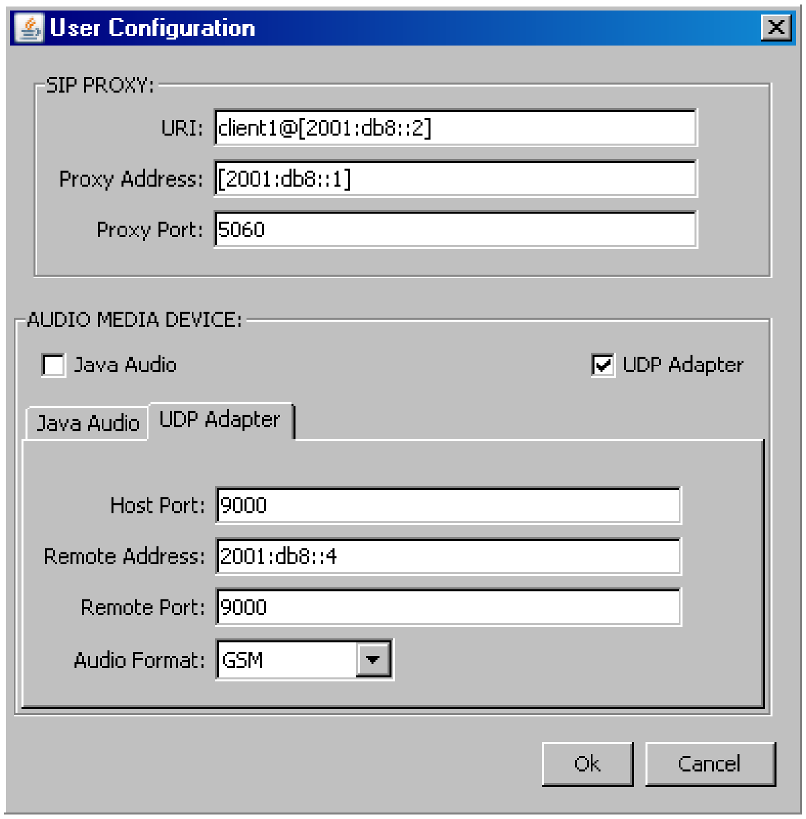
URI: client1@192.168.1.3
Proxy Address: 192.168.1.2
Proxy Port: 5060

☐ Java Audio ☑ UDP Adapter

Host Port: 9000
Remote Address: 2001:db8::4
Remote Port: 9000
Audio Format: GSM

Figure 8.3: SIP VoIP Adapter configuration over IPv4

It is important to note that in both cases, the remote IP address in the UDP Adapter configuration is defined in IPv6. This is because the end device chosen for a future work, 6lowPAN device, works in IPv6.



SIP PROXY:

URI: client1@[2001:db8::2]

Proxy Address: [2001:db8::1]

Proxy Port: 5060

AUDIO MEDIA DEVICE:

☐ Java Audio ☑ UDP Adapter

Host Port: 9000

Remote Address: 2001:db8::4

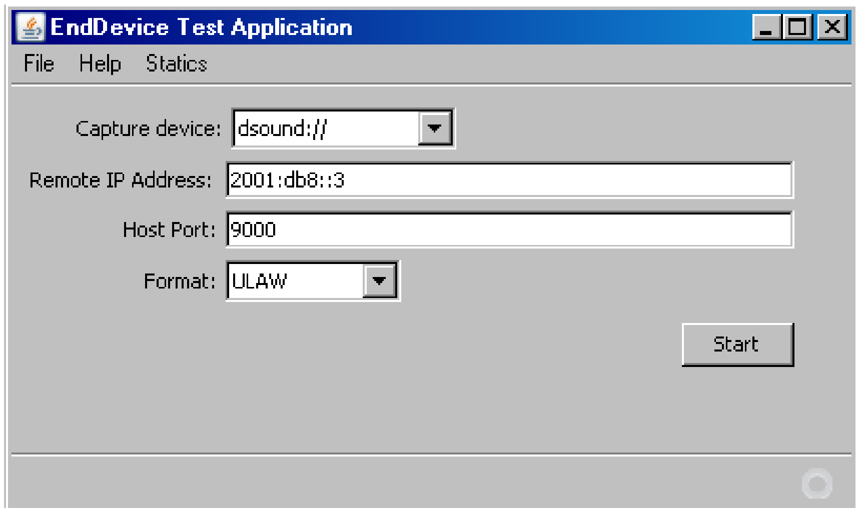
Remote Port: 9000

Audio Format: GSM

Figure 8.4: SIP VoIP Adapter configuration over IPv6

8.3.2 EndDevice Test

The EndDevice Test application has been installed on the computer with address IPv6 2001:db8::4. Figure 8.5 shows the parameters used to configure the EndDevice Test application to act as the sound device of the SIP VoIP Adapter in IPv6.



Capture device: dsound://
Remote IP Address: 2001:db8::3
Host Port: 9000
Format: ULAW

Figure 8.5: EndDevice Test configuration

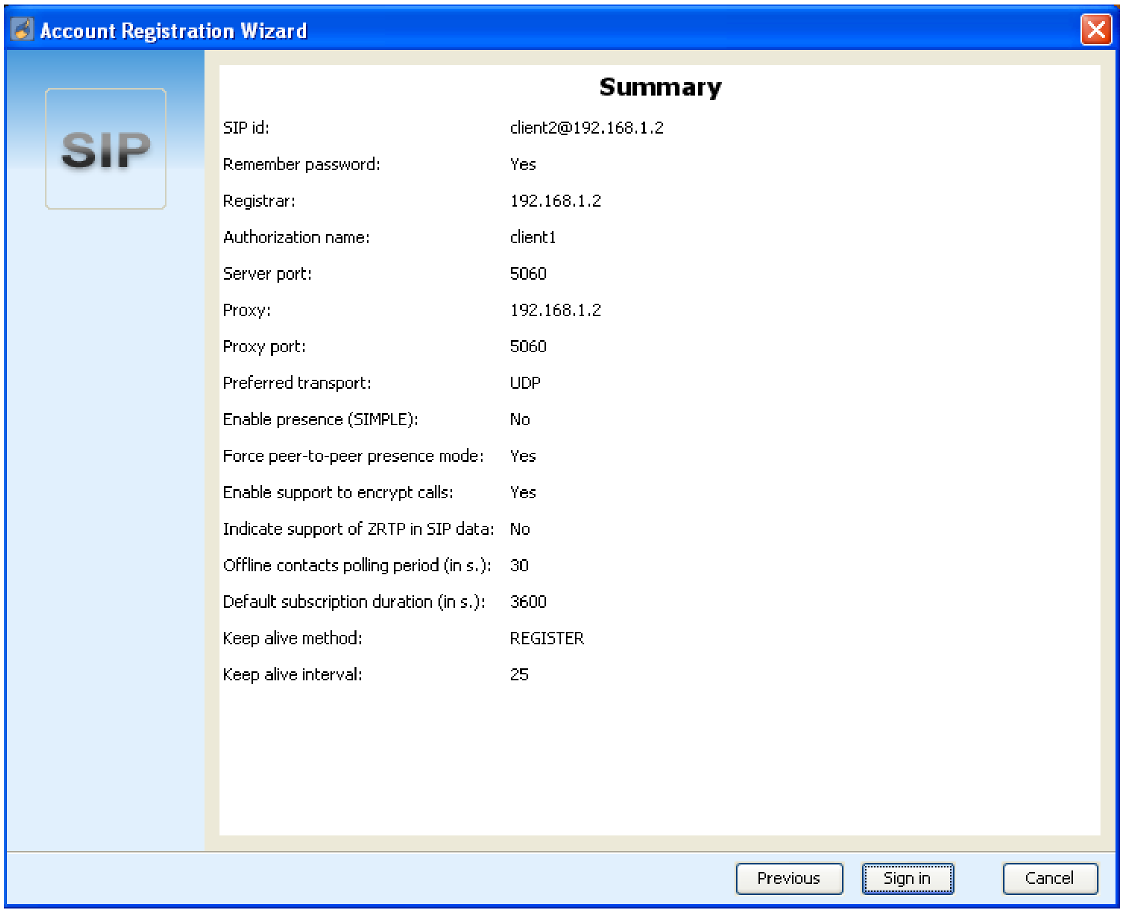
8.3.3 SIP Communicator

SIP Communicator is an audio/video Internet phone and instant messenger that supports some of the most popular instant messaging and telephony protocols such as SIP, Jabber, AIM/ICQ, MSN, Yahoo! Messenger, Bonjour, IRC, RSS and counting. This software has been chosen to be the other SIP end client, to establish a communication with the SIP VoIP Adapter through Asterisk.

Figure 8.6 and Figure 8.7 shows the SIP Communicator parameters configured for establishing a communication through Asterisk PBX, in IPv4 and IPv6 respectively.

8.3.4 Asterisk PBX

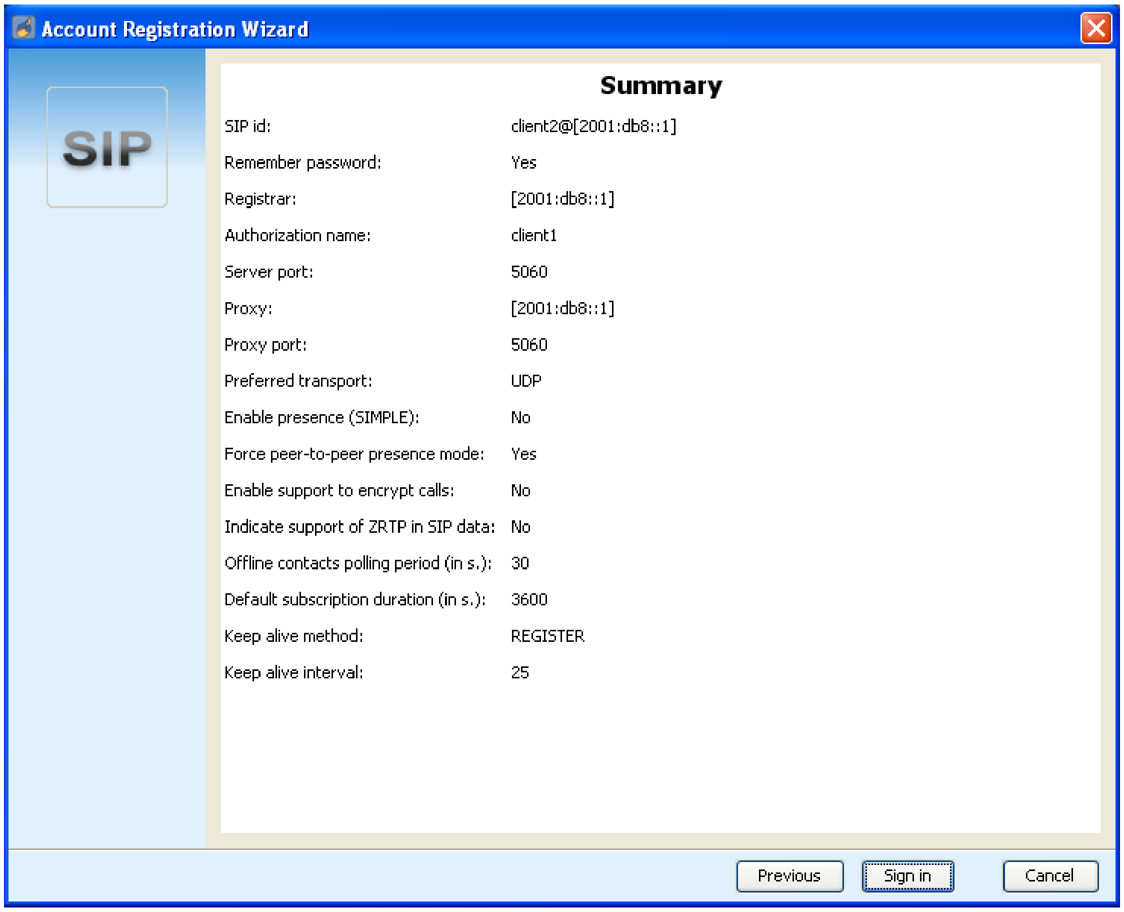
The Asterisk configuration has already described in chapter 7.



Summary

SIP id:	client2@192.168.1.2
Remember password:	Yes
Registrar:	192.168.1.2
Authorization name:	client1
Server port:	5060
Proxy:	192.168.1.2
Proxy port:	5060
Preferred transport:	UDP
Enable presence (SIMPLE):	No
Force peer-to-peer presence mode:	Yes
Enable support to encrypt calls:	Yes
Indicate support of ZRTP in SIP data:	No
Offline contacts polling period (in s.):	30
Default subscription duration (in s.):	3600
Keep alive method:	REGISTER
Keep alive interval:	25

Figure 8.6: SIP Communicator configuration in IPv4



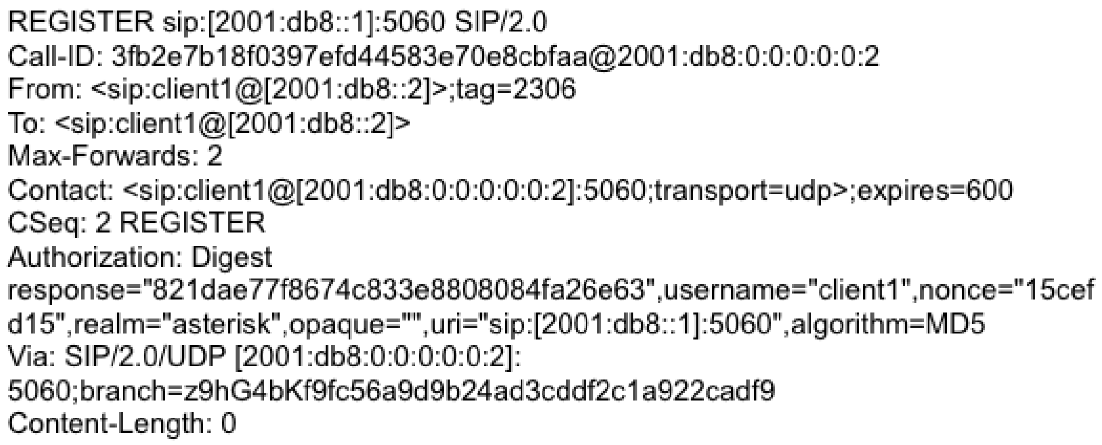
SIP id:	client2@[2001:db8::1]
Remember password:	Yes
Registrar:	[2001:db8::1]
Authorization name:	client1
Server port:	5060
Proxy:	[2001:db8::1]
Proxy port:	5060
Preferred transport:	UDP
Enable presence (SIMPLE):	No
Force peer-to-peer presence mode:	Yes
Enable support to encrypt calls:	No
Indicate support of ZRTP in SIP data:	No
Offline contacts polling period (in s.):	30
Default subscription duration (in s.):	3600
Keep alive method:	REGISTER
Keep alive interval:	25

Figure 8.7: SIP Communicator configuration in IPv6

8.4 Registering Process

Once the system has been configured as described before, the user is able to register the SIP VoIP Adapter to Asterisk PBX.

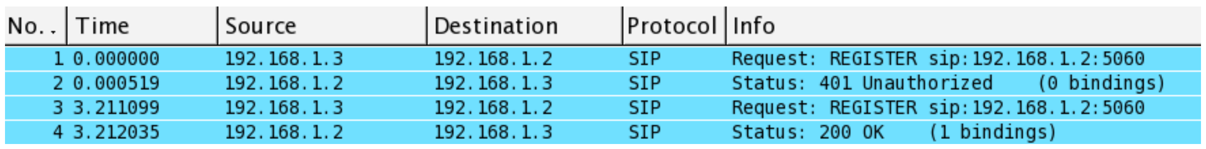
Pushing the button ”Register” the user will see how the register status label change from ”(Offline)” to ”Trying...”, which it means that a SIP REGISTER request has been sent. Asterisk answers the request by sending a SIP UNAUTHORIZED response due to the client in Asterisk has been configured with password registration. So, when SIP VoIP Adapter will receive this response, a new window is opened asking the user for the user name and password. Once the parameters has been filled, and the button ”Ok” has been pushed, another SIP REGISTER request is sent, but now with a header authentication. If Asterisk accepts the registration it will send a SIP Ok response, and the SIP VoIP Adapter will change the ”Register Status” from ”Trying...” to ”(Online)”. Figure 8.8 shows the REGISTER request message with authentication sent.



```
REGISTER sip:[2001:db8::1]:5060 SIP/2.0
Call-ID: 3fb2e7b18f0397efd44583e70e8cbfaa@2001:db8:0:0:0:0:0:2
From: <sip:client1@[2001:db8::2]>;tag=2306
To: <sip:client1@[2001:db8::2]>
Max-Forwards: 2
Contact: <sip:client1@[2001:db8:0:0:0:0:0:2]:5060;transport=udp>;expires=600
CSeq: 2 REGISTER
Authorization: Digest
response="821dae77f8674c833e8808084fa26e63",username="client1",nonce="15cef
d15",realm="asterisk",opaque="",uri="sip:[2001:db8::1]:5060",algorithm=MD5
Via: SIP/2.0/UDP [2001:db8:0:0:0:0:0:2]:
5060;branch=z9hG4bKf9fc56a9d9b24ad3cddf2c1a922cadf9
Content-Length: 0
```

Figure 8.8: REGISTER request message with Authentication sent

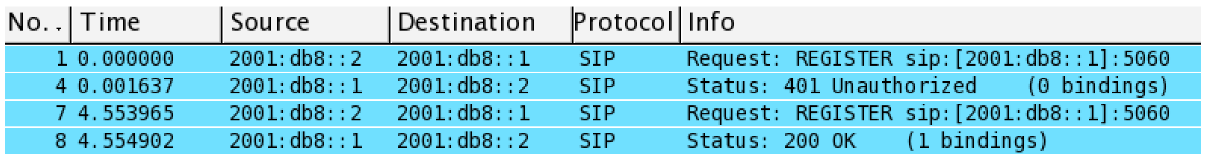
Figure 8.9 and Figure 8.10 shows the packets exchanges between the SIP VoIP Adapter and Asterisk, for the SIP VoIP Adapter register process in IPv4 and IPv6 respectively.



No..	Time	Source	Destination	Protocol	Info
1	0.000000	192.168.1.3	192.168.1.2	SIP	Request: REGISTER sip:192.168.1.2:5060
2	0.000519	192.168.1.2	192.168.1.3	SIP	Status: 401 Unauthorized (0 bindings)
3	3.211099	192.168.1.3	192.168.1.2	SIP	Request: REGISTER sip:192.168.1.2:5060
4	3.212035	192.168.1.2	192.168.1.3	SIP	Status: 200 OK (1 bindings)

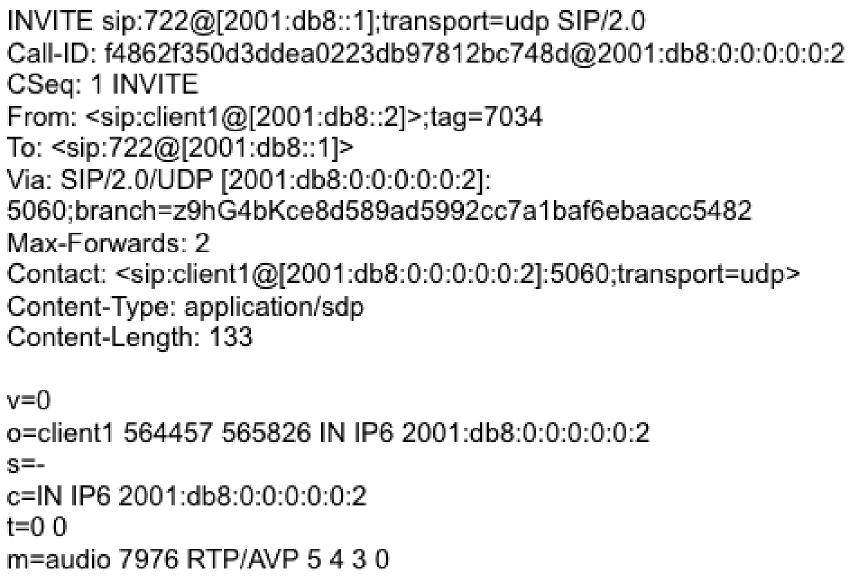
Figure 8.9: SIP Registrar Signaling between SIP VoIP Adapter and Asterisk in IPv4

Now the user is able to call another client also registered in Asterisk. Figure 8.11 shows the second SIP REGISTER request message sent, so with authentication header.



```
No.. Time      Source       Destination  Protocol Info
   1 0.000000  2001:db8::2  2001:db8::1  SIP      Request: REGISTER sip:[2001:db8::1]:5060
   4 0.001637  2001:db8::1  2001:db8::2  SIP      Status: 401 Unauthorized    (0 bindings)
   7 4.553965  2001:db8::2  2001:db8::1  SIP      Request: REGISTER sip:[2001:db8::1]:5060
   8 4.554902  2001:db8::1  2001:db8::2  SIP      Status: 200 OK    (1 bindings)
```

Figure 8.10: SIP Registrar Signaling between SIP VoIP Adapter and Asterisk in IPv6



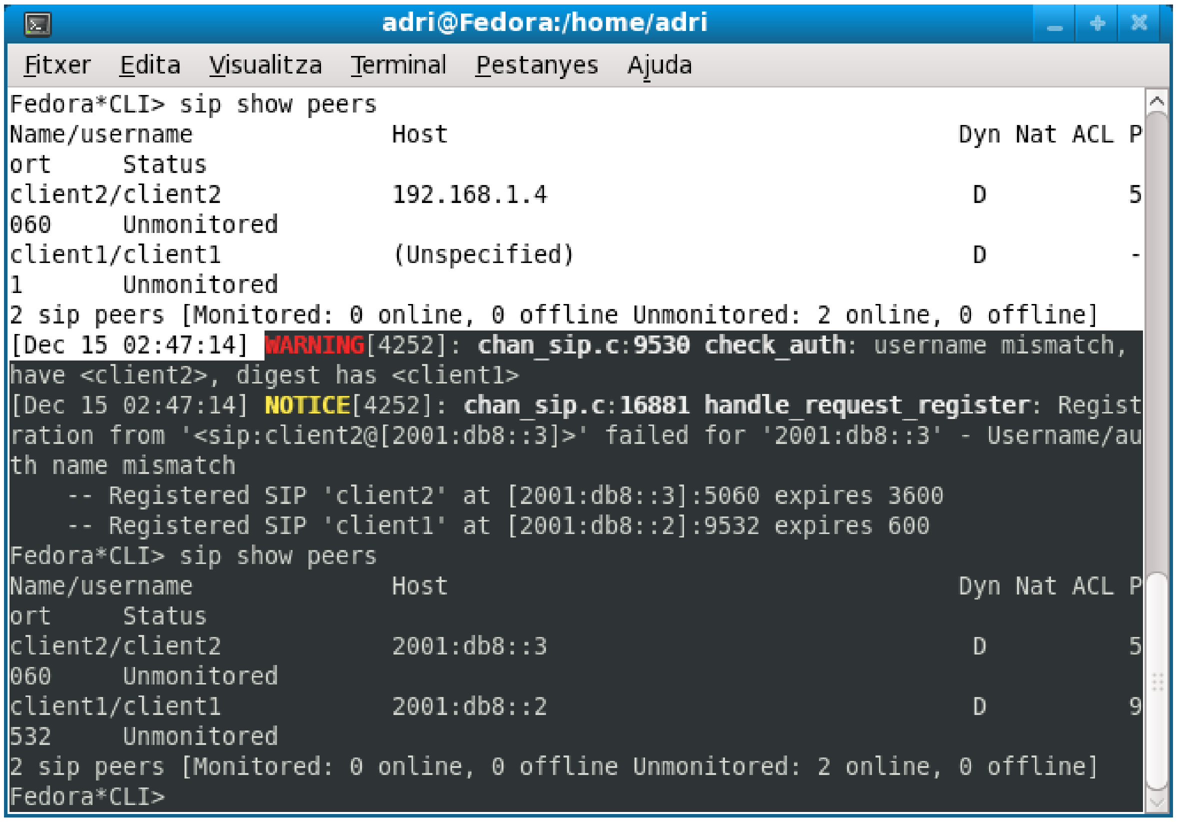
```
INVITE sip:722@[2001:db8::1];transport=udp SIP/2.0
Call-ID: f4862f350d3ddea0223db97812bc748d@2001:db8:0:0:0:0:0:2
CSeq: 1 INVITE
From: <sip:client1@[2001:db8::2]>;tag=7034
To: <sip:722@[2001:db8::1]>
Via: SIP/2.0/UDP [2001:db8:0:0:0:0:0:2]:
5060;branch=z9hG4bKce8d589ad5992cc7a1baf6ebaacc5482
Max-Forwards: 2
Contact: <sip:client1@[2001:db8:0:0:0:0:0:2]:5060;transport=udp>
Content-Type: application/sdp
Content-Length: 133

v=0
o=client1 564457 565826 IN IP6 2001:db8:0:0:0:0:0:2
s=-
c=IN IP6 2001:db8:0:0:0:0:0:2
t=0 0
m=audio 7976 RTP/AVP 5 4 3 0
```

Figure 8.11: INVITE request message with Authentication sent

All SIP messages exchanged has been attached on the Appendix in Appendix B.1 for IPv4 and Appendix B.2 for IPv6.

On the other hand, the user is able to see in Figure 8.12 the Asterisk information shown on the computer where it is running. To see this information, it is needed to open a terminal on the computer where Asterisk is running and tap the following instructions:

# Asterisk -rvvv
# sip show peers



```
Fedora*CLI> sip show peers
Name/username              Host                                    Dyn Nat ACL P
ort      Status
client2/client2            192.168.1.4                              D          5
060      Unmonitored
client1/client1            (Unspecified)                            D          -
1        Unmonitored
2 sip peers [Monitored: 0 online, 0 offline Unmonitored: 2 online, 0 offline]
[Dec 15 02:47:14] WARNING[4252]: chan_sip.c:9530 check_auth: username mismatch,
have <client2>, digest has <client1>
[Dec 15 02:47:14] NOTICE[4252]: chan_sip.c:16881 handle_request_register: Regist
ration from '<sip:client2@[2001:db8::3]>' failed for '2001:db8::3' - Username/au
th name mismatch
    -- Registered SIP 'client2' at [2001:db8::3]:5060 expires 3600
    -- Registered SIP 'client1' at [2001:db8::2]:9532 expires 600
Fedora*CLI> sip show peers
Name/username              Host                                    Dyn Nat ACL P
ort      Status
client2/client2            2001:db8::3                              D          5
060      Unmonitored
client1/client1            2001:db8::2                              D          9
532      Unmonitored
2 sip peers [Monitored: 0 online, 0 offline Unmonitored: 2 online, 0 offline]
Fedora*CLI>
```

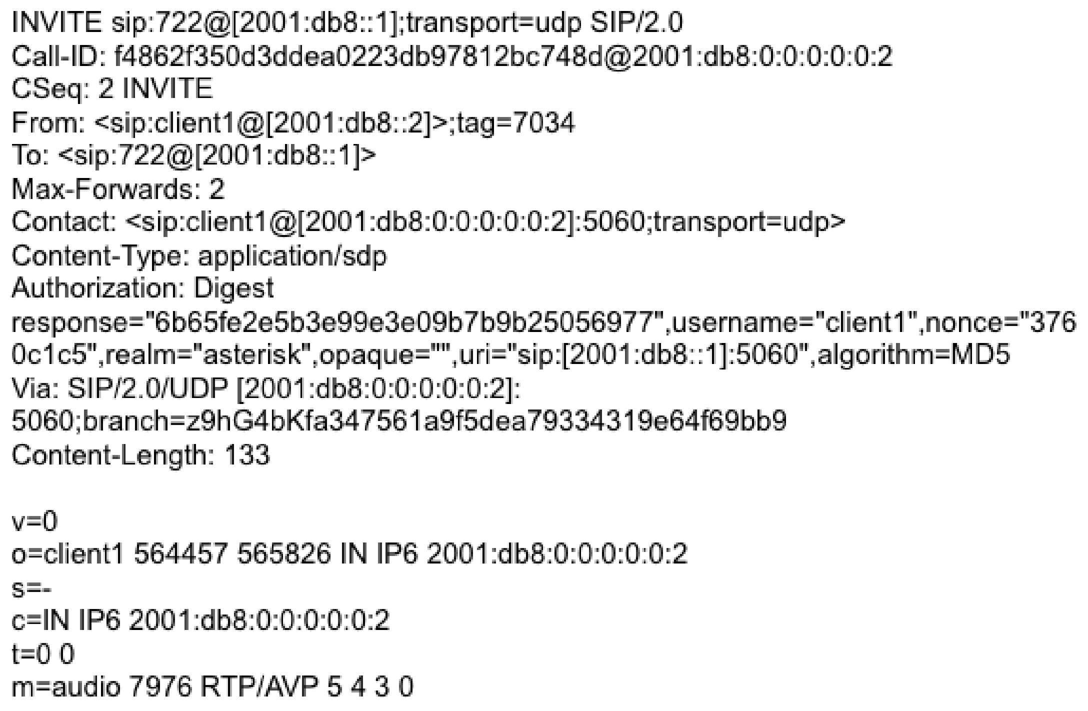
Figure 8.12: Asterisk screen: sip show peers comand

8.5 Call Process

8.5.1 Make a Call

Once the SIP VoIP Adapter is registered in Asterisk, the user is able to make a call to the other client also registered in Asterisk. First of all, the SIP URI of the client to call must be introduced in ”Call Number”. Then, pushing the button ”Open Call Manager” a new call window is opened. Now the user is able to make the call by pushing the button ”Make Call”. In addition, the call window shows some information about the current call, in the text area located on the bottom of the window.

When the button ”Make Call” is pushed, the SIP VoIP Adapter send an INVITE request with an SDP message to Asterisk, which answer sending a PROXY AUTHORIZATION REQUIRED response. When this response is received, a new window is opened asking the user for the authorization parameters. When the user pushes the button ”Ok”, after being introduced the user name and password, another INVITE request is sent back, now with the authorization header. Now Asterisk will answer first sending a TRYING response, and then, if the INVITE request received is correct, and the client to be called is registered, it will send an INVITE request to the other client and a TRYING response to the SIP VoIP Adapter. Figure 8.13 shows the INVITE request message sent.

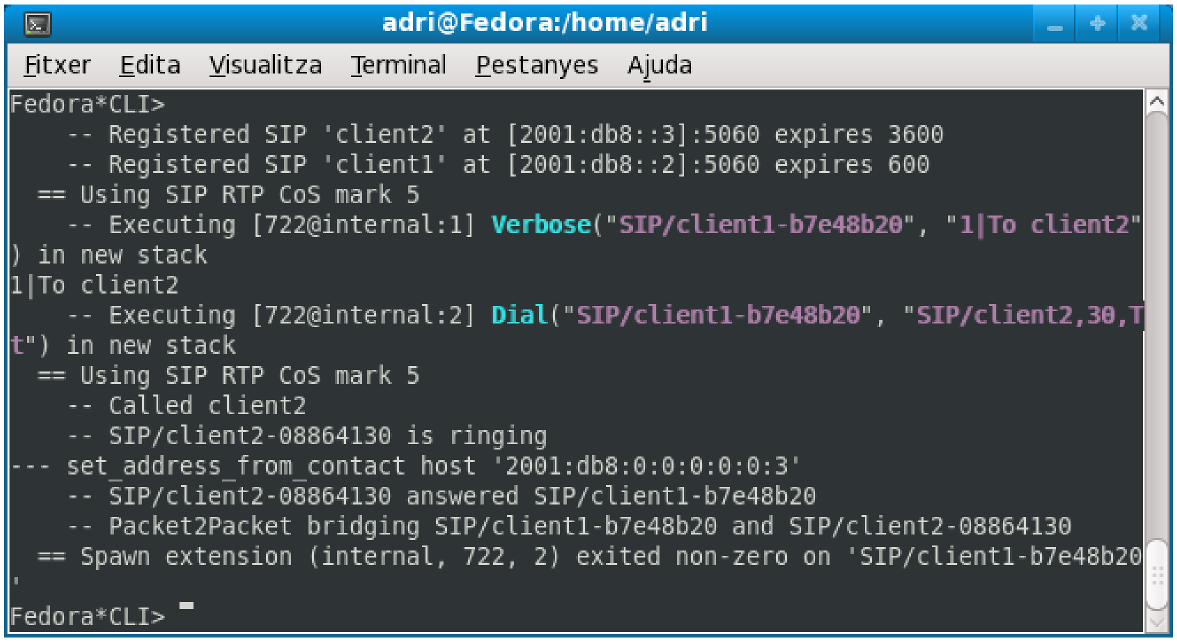


```
INVITE sip:722@[2001:db8::1];transport=udp SIP/2.0
Call-ID: f4862f350d3ddea0223db97812bc748d@2001:db8:0:0:0:0:0:2
CSeq: 2 INVITE
From: <sip:client1@[2001:db8::2]>;tag=7034
To: <sip:722@[2001:db8::1]>
Max-Forwards: 2
Contact: <sip:client1@[2001:db8:0:0:0:0:0:2]:5060;transport=udp>
Content-Type: application/sdp
Authorization: Digest
response="6b65fe2e5b3e99e3e09b7b9b25056977",username="client1",nonce="376
0c1c5",realm="asterisk",opaque="",uri="sip:[2001:db8::1]:5060",algorithm=MD5
Via: SIP/2.0/UDP [2001:db8:0:0:0:0:0:2]:
5060;branch=z9hG4bKfa347561a9f5dea79334319e64f69bb9
Content-Length: 133

v=0
o=client1 564457 565826 IN IP6 2001:db8:0:0:0:0:0:2
s=-
c=IN IP6 2001:db8:0:0:0:0:0:2
t=0 0
m=audio 7976 RTP/AVP 5 4 3 0
```

Figure 8.13: INVITE request message sent

Once the other end client has answered and Asterisk has confirm that the communication can be established, it responses with an OK response. When the OK response is received by the SIP VoIP Adapter it sends an ACK and starts the RTP/RTCP communication.

Figure 8.14 shows the Asterisk information related to the communication between two end clients.



```
Fedora*CLI>
    -- Registered SIP 'client2' at [2001:db8::3]:5060 expires 3600
    -- Registered SIP 'client1' at [2001:db8::2]:5060 expires 600
  == Using SIP RTP CoS mark 5
    -- Executing [722@internal:1] Verbose("SIP/client1-b7e48b20", "1|To client2"
) in new stack
1|To client2
    -- Executing [722@internal:2] Dial("SIP/client1-b7e48b20", "SIP/client2,30,T
t") in new stack
  == Using SIP RTP CoS mark 5
    -- Called client2
    -- SIP/client2-08864130 is ringing
--- set_address_from_contact host '2001:db8:0:0:0:0:0:3'
    -- SIP/client2-08864130 answered SIP/client1-b7e48b20
    -- Packet2Packet bridging SIP/client1-b7e48b20 and SIP/client2-08864130
  == Spawn extension (internal, 722, 2) exited non-zero on 'SIP/client1-b7e48b20
'
Fedora*CLI>
```

Figure 8.14: INVITE request message sent

When the RTP/RTCP communication starts, the SIP VoIP Adapter tries to acquire the audio from the port specified in the SIP VoIP Adapter configuration, and it is blocked until it receive the first audio stream from the EndDevice Test application, or in other words, when it is started.
On the other, the audio received from the RTP communication is sent to the IP address and port also specified in the SIP VoIP Adapter configuration, being or not received by the end device. Therefore, if the EndDevice Test application is started, by pushing the button ”Start”, it starts receiving and playing the audio received through the specified port, in addition to start transmitting the audio acquired from the microphone to the address and port also specified.

Figure 8.15 and Figure 8.16 shows the packets exchanges between SIP VoIP Adapter, EndDevice Test application, Asterisk and the other end client in IPv4 and IPv6 respectively. In it, the reader can observe the protocols involved in each communication:

- The protocols involved between the SIP VoIP Adapter and Asterisk PBX are:
  - SIP for the signalization
  - SDP for the audio media negotiation
  - RTP/RTCP for the audio communication
- The protocols involved between the SIP VoIP Adapter and the EndDevice Test application are:

  - UDP for the audio data transmission

- The protocols involved between the SIP Client (SIP Communicator) and Asterisk PBX are:
  - SIP for the signalization
  - SDP for the audio media negotiation
  - RTP/RTCP for the audio communication

It is important to pare attention in the black color packets, which are defined by Wireshark as "Fragmented IP protocol". This packets are the audio being transmitted between the SIP VoIP Adapter and the EndDevice Test Aplication. This is why it is necessary to pare attencion in the origin and destination of the packets. Figure 8.17 shows a packet send from the SIP VoIP Adapter to the EndDevice Test application. Figure 8.18 shows a packet send from the EndDevice Test application to SIP VoIP Adapter.

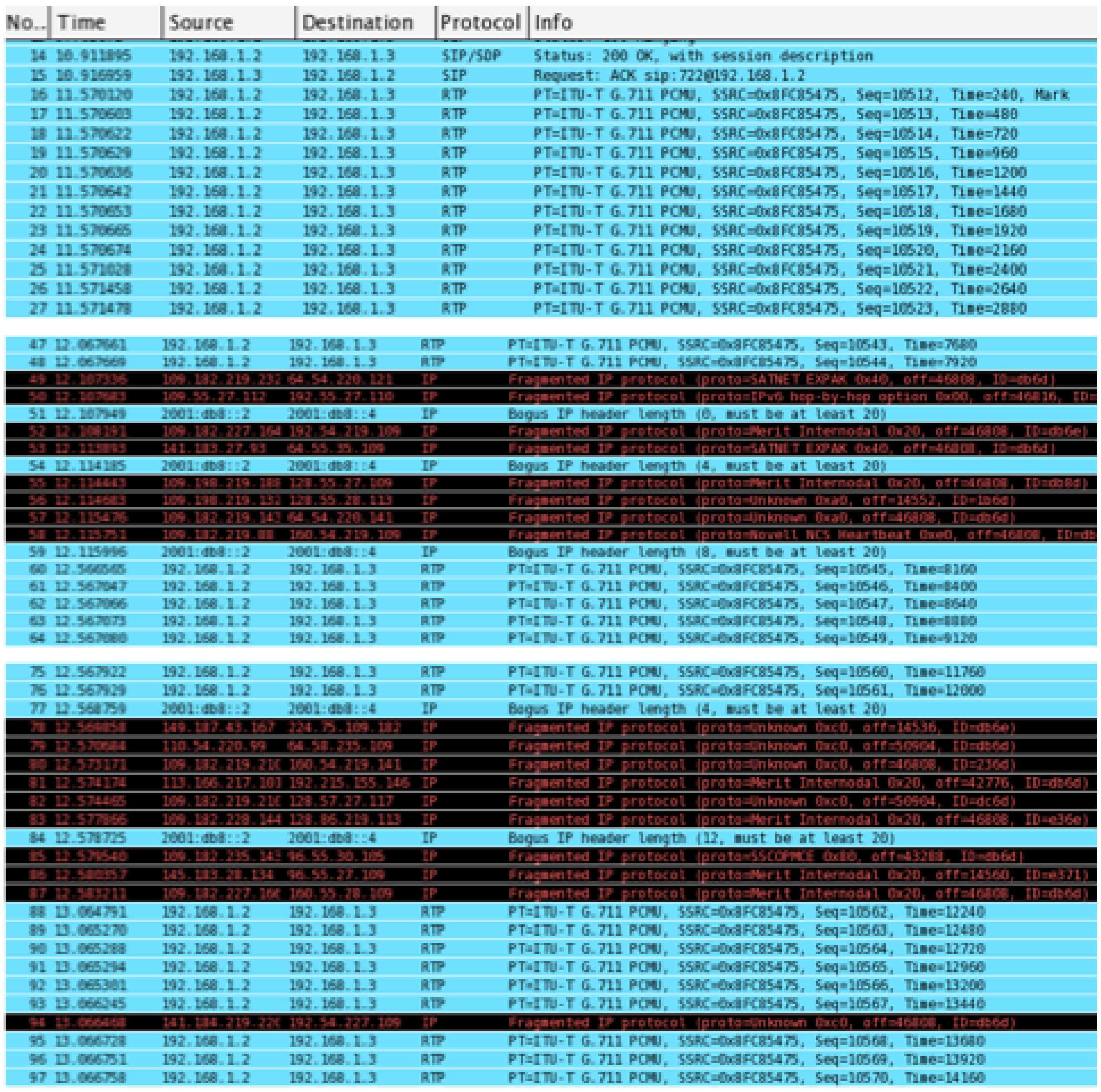
On the other hand, Figure 8.19 and Figure 8.20 shows the communication scheme between both SIP client and Asterisk, in IPv4 and IPv6 respectively.

8.5.2 Receive a Call

Besides the ability to make calls, the SIP VoIP Adapter must be able to receive calls. Therefore, when the SIP VoIP Adapter is registered in Asterisk and the user is not already in a call, the other SIP user (SIP Communicator) also registered in Asterisk is able to call us.

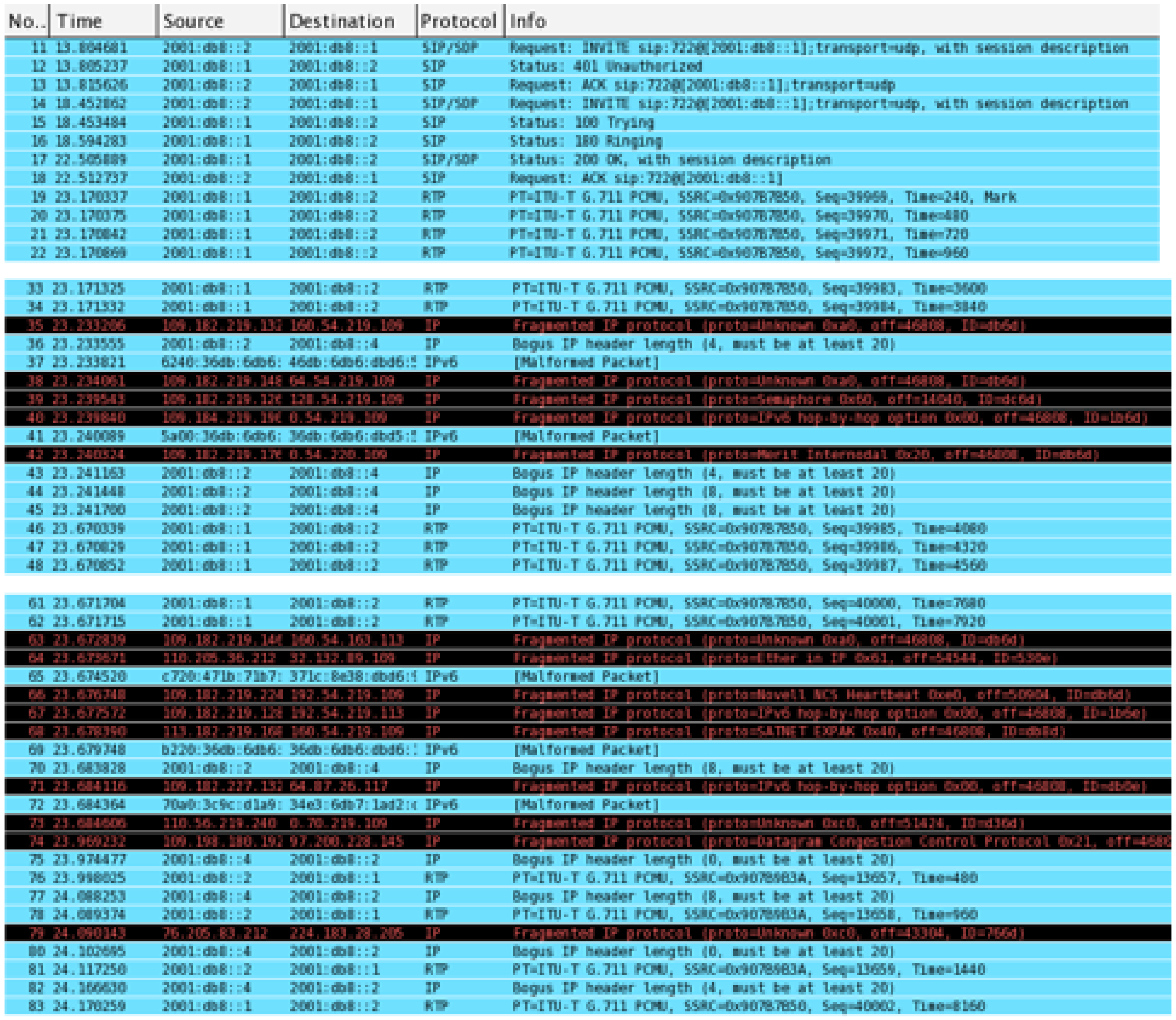
When the SIP Communicator call the SIP VoIP Adapter ("client1") it performs the same actions we have already described in section 8.5.1. Thus, the SIP VoIP Adapter notice the user that someone is trying to call him when an INVITE request message from Asterisk is received. Then, the SIP VoIP Adapter sends a TRYING response message to warn that the user is not in a call, and it opens an information window asking the user if wants or not answer the call. If the chosen option is "Yes", the SIP VoIP Adapter sends an OK response message with an SDP body content in it. Once an ACK response from Asterisk is received, the SIP VoIP Adapter can start the RTP/RTCP communication with the parameters established by the SIP signalization, as well as the UDP communication.

Once the user of SIP VoIP Adapter answer the incoming call, the EndDevice Test application can be started acquiring audio from the microphone and playing the audio received.



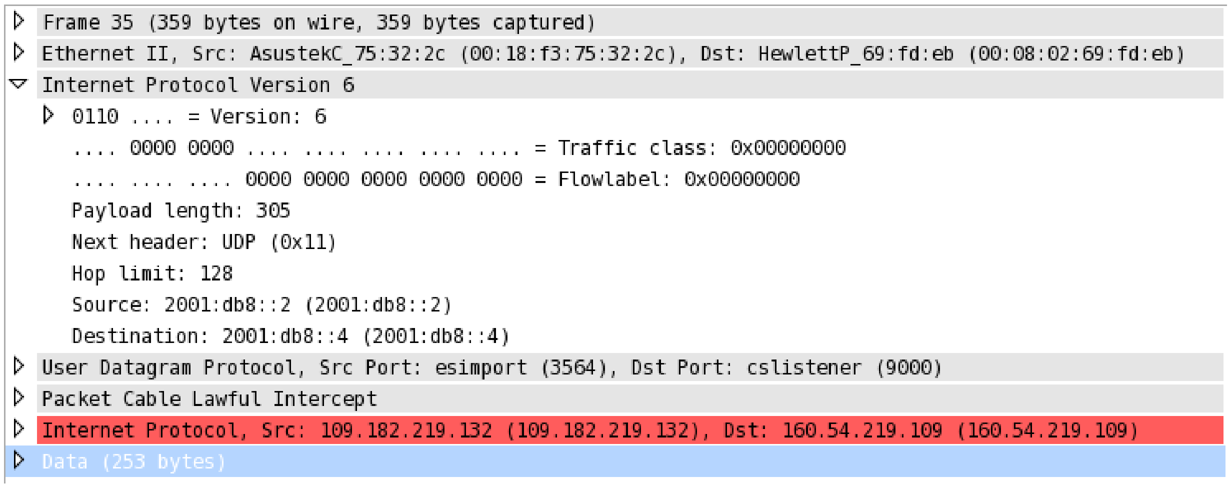
```
No.. Time       Source           Destination      Protocol  Info
  14 10.911895  192.168.1.2      192.168.1.3      SIP/SDP   Status: 200 OK, with session description
  15 10.916959  192.168.1.3      192.168.1.2      SIP       Request: ACK sip:722@192.168.1.2
  16 11.570120  192.168.1.2      192.168.1.3      RTP       PT=ITU-T G.711 PCMU, SSRC=0x8FC85475, Seq=10512, Time=240, Mark
  17 11.570603  192.168.1.2      192.168.1.3      RTP       PT=ITU-T G.711 PCMU, SSRC=0x8FC85475, Seq=10513, Time=480
  18 11.570622  192.168.1.2      192.168.1.3      RTP       PT=ITU-T G.711 PCMU, SSRC=0x8FC85475, Seq=10514, Time=720
  19 11.570629  192.168.1.2      192.168.1.3      RTP       PT=ITU-T G.711 PCMU, SSRC=0x8FC85475, Seq=10515, Time=960
  20 11.570636  192.168.1.2      192.168.1.3      RTP       PT=ITU-T G.711 PCMU, SSRC=0x8FC85475, Seq=10516, Time=1200
  21 11.570642  192.168.1.2      192.168.1.3      RTP       PT=ITU-T G.711 PCMU, SSRC=0x8FC85475, Seq=10517, Time=1440
  22 11.570653  192.168.1.2      192.168.1.3      RTP       PT=ITU-T G.711 PCMU, SSRC=0x8FC85475, Seq=10518, Time=1680
  23 11.570665  192.168.1.2      192.168.1.3      RTP       PT=ITU-T G.711 PCMU, SSRC=0x8FC85475, Seq=10519, Time=1920
  24 11.570674  192.168.1.2      192.168.1.3      RTP       PT=ITU-T G.711 PCMU, SSRC=0x8FC85475, Seq=10520, Time=2160
  25 11.571028  192.168.1.2      192.168.1.3      RTP       PT=ITU-T G.711 PCMU, SSRC=0x8FC85475, Seq=10521, Time=2400
  26 11.571458  192.168.1.2      192.168.1.3      RTP       PT=ITU-T G.711 PCMU, SSRC=0x8FC85475, Seq=10522, Time=2640
  27 11.571478  192.168.1.2      192.168.1.3      RTP       PT=ITU-T G.711 PCMU, SSRC=0x8FC85475, Seq=10523, Time=2880

  47 12.067661  192.168.1.2      192.168.1.3      RTP       PT=ITU-T G.711 PCMU, SSRC=0x8FC85475, Seq=10543, Time=7680
  48 12.067669  192.168.1.2      192.168.1.3      RTP       PT=ITU-T G.711 PCMU, SSRC=0x8FC85475, Seq=10544, Time=7920
  49 12.107336  109.182.219.232 64.54.220.121    IP        Fragmented IP protocol (proto=SATNET EXPAK 0x40, off=46808, ID=db6d)
  50 12.107683  109.55.27.112    192.55.27.110    IP        Fragmented IP protocol (proto=IPv6 hop-by-hop option 0x00, off=46816, ID=
  51 12.107949  2001:db8::2      2001:db8::4      IP        Bogus IP header length (0, must be at least 20)
  52 12.108191  109.182.227.164 192.54.219.109    IP        Fragmented IP protocol (proto=Merit Internodal 0x20, off=46808, ID=db6e)
  53 12.113893  141.183.27.93    64.55.35.109     IP        Fragmented IP protocol (proto=SATNET EXPAK 0x40, off=46808, ID=db6d)
  54 12.114185  2001:db8::2      2001:db8::4      IP        Bogus IP header length (4, must be at least 20)
  55 12.114443  109.198.219.188 128.55.27.109     IP        Fragmented IP protocol (proto=Merit Internodal 0x20, off=46808, ID=db8d)
  56 12.114683  109.198.219.132 128.55.28.113     IP        Fragmented IP protocol (proto=Unknown 0xa0, off=14552, ID=1b6d)
  57 12.115476  109.182.219.143 64.54.220.141     IP        Fragmented IP protocol (proto=Unknown 0xa0, off=46808, ID=db6d)
  58 12.115751  109.182.219.88  160.54.219.109    IP        Fragmented IP protocol (proto=Novell NCS Heartbeat 0xe0, off=46808, ID=db
  59 12.115996  2001:db8::2      2001:db8::4      IP        Bogus IP header length (8, must be at least 20)
  60 12.566565  192.168.1.2      192.168.1.3      RTP       PT=ITU-T G.711 PCMU, SSRC=0x8FC85475, Seq=10545, Time=8160
  61 12.567047  192.168.1.2      192.168.1.3      RTP       PT=ITU-T G.711 PCMU, SSRC=0x8FC85475, Seq=10546, Time=8400
  62 12.567066  192.168.1.2      192.168.1.3      RTP       PT=ITU-T G.711 PCMU, SSRC=0x8FC85475, Seq=10547, Time=8640
  63 12.567073  192.168.1.2      192.168.1.3      RTP       PT=ITU-T G.711 PCMU, SSRC=0x8FC85475, Seq=10548, Time=8880
  64 12.567080  192.168.1.2      192.168.1.3      RTP       PT=ITU-T G.711 PCMU, SSRC=0x8FC85475, Seq=10549, Time=9120

  75 12.567922  192.168.1.2      192.168.1.3      RTP       PT=ITU-T G.711 PCMU, SSRC=0x8FC85475, Seq=10560, Time=11760
  76 12.567929  192.168.1.2      192.168.1.3      RTP       PT=ITU-T G.711 PCMU, SSRC=0x8FC85475, Seq=10561, Time=12000
  77 12.568759  2001:db8::2      2001:db8::4      IP        Bogus IP header length (4, must be at least 20)
  78 12.569858  149.187.43.167   224.75.109.182   IP        Fragmented IP protocol (proto=Unknown 0xc0, off=14536, ID=db6e)
  79 12.570684  110.54.220.99    64.58.235.109    IP        Fragmented IP protocol (proto=Unknown 0xc0, off=50904, ID=db6d)
  80 12.573171  109.182.219.21( 160.54.219.141    IP        Fragmented IP protocol (proto=Unknown 0xc0, off=46808, ID=236d)
  81 12.574174  113.166.217.10] 192.215.155.146   IP        Fragmented IP protocol (proto=Merit Internodal 0x20, off=42776, ID=db6d)
  82 12.574465  109.182.219.21( 128.57.27.117     IP        Fragmented IP protocol (proto=Unknown 0xc0, off=50904, ID=dc6d)
  83 12.577866  109.182.228.144 128.86.219.113    IP        Fragmented IP protocol (proto=Merit Internodal 0x20, off=46808, ID=e36e)
  84 12.578725  2001:db8::2      2001:db8::4      IP        Bogus IP header length (12, must be at least 20)
  85 12.579540  109.182.235.14: 96.55.30.105      IP        Fragmented IP protocol (proto=SSCOPMCE 0x80, off=43288, ID=db6d)
  86 12.580357  145.183.28.134   96.55.27.109     IP        Fragmented IP protocol (proto=Merit Internodal 0x20, off=14560, ID=e371)
  87 12.583211  109.182.227.16( 160.55.28.109     IP        Fragmented IP protocol (proto=Merit Internodal 0x20, off=46808, ID=db6d)
  88 13.064791  192.168.1.2      192.168.1.3      RTP       PT=ITU-T G.711 PCMU, SSRC=0x8FC85475, Seq=10562, Time=12240
  89 13.065270  192.168.1.2      192.168.1.3      RTP       PT=ITU-T G.711 PCMU, SSRC=0x8FC85475, Seq=10563, Time=12480
  90 13.065288  192.168.1.2      192.168.1.3      RTP       PT=ITU-T G.711 PCMU, SSRC=0x8FC85475, Seq=10564, Time=12720
  91 13.065294  192.168.1.2      192.168.1.3      RTP       PT=ITU-T G.711 PCMU, SSRC=0x8FC85475, Seq=10565, Time=12960
  92 13.065301  192.168.1.2      192.168.1.3      RTP       PT=ITU-T G.711 PCMU, SSRC=0x8FC85475, Seq=10566, Time=13200
  93 13.066245  192.168.1.2      192.168.1.3      RTP       PT=ITU-T G.711 PCMU, SSRC=0x8FC85475, Seq=10567, Time=13440
  94 13.066468  141.184.219.22( 192.54.227.109    IP        Fragmented IP protocol (proto=Unknown 0xc0, off=46808, ID=db6d)
  95 13.066728  192.168.1.2      192.168.1.3      RTP       PT=ITU-T G.711 PCMU, SSRC=0x8FC85475, Seq=10568, Time=13680
  96 13.066751  192.168.1.2      192.168.1.3      RTP       PT=ITU-T G.711 PCMU, SSRC=0x8FC85475, Seq=10569, Time=13920
  97 13.066758  192.168.1.2      192.168.1.3      RTP       PT=ITU-T G.711 PCMU, SSRC=0x8FC85475, Seq=10570, Time=14160
```

Figure 8.15: SIP VoIP Adapter packets exchanged in a VoIP call, IPv4



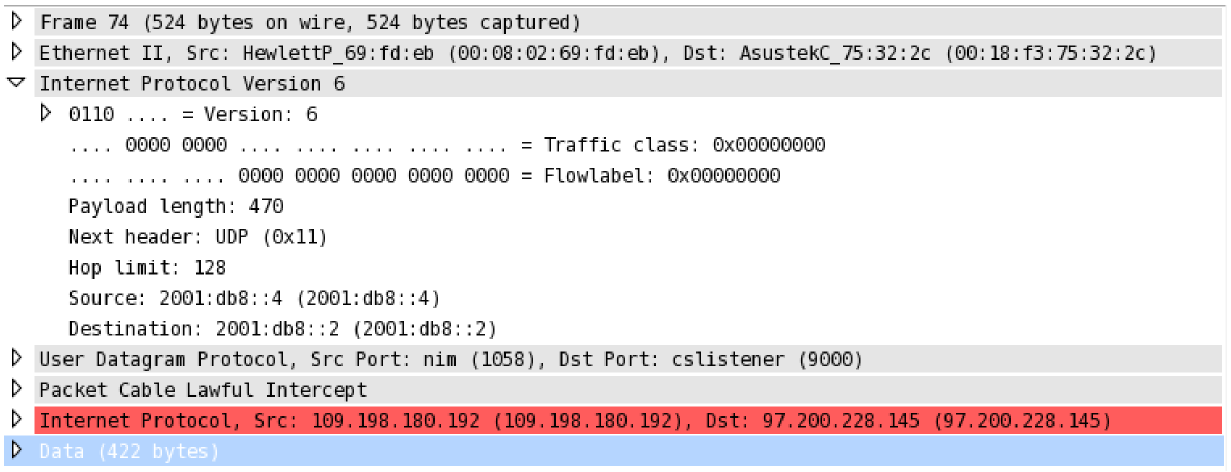
No..	Time	Source	Destination	Protocol	Info
11	13.804681	2001:db8::2	2001:db8::1	SIP/SDP	Request: INVITE sip:722@[2001:db8::1];transport=udp, with session description
12	13.805237	2001:db8::1	2001:db8::2	SIP	Status: 401 Unauthorized
13	13.815626	2001:db8::2	2001:db8::1	SIP	Request: ACK sip:722@[2001:db8::1];transport=udp
14	18.452862	2001:db8::2	2001:db8::1	SIP/SDP	Request: INVITE sip:722@[2001:db8::1];transport=udp, with session description
15	18.453484	2001:db8::1	2001:db8::2	SIP	Status: 100 Trying
16	18.594283	2001:db8::1	2001:db8::2	SIP	Status: 180 Ringing
17	22.505889	2001:db8::1	2001:db8::2	SIP/SDP	Status: 200 OK, with session description
18	22.512737	2001:db8::2	2001:db8::1	SIP	Request: ACK sip:722@[2001:db8::1]
19	23.170337	2001:db8::1	2001:db8::2	RTP	PT=ITU-T G.711 PCMU, SSRC=0x907B7850, Seq=39969, Time=240, Mark
20	23.170375	2001:db8::1	2001:db8::2	RTP	PT=ITU-T G.711 PCMU, SSRC=0x907B7850, Seq=39970, Time=480
21	23.170842	2001:db8::1	2001:db8::2	RTP	PT=ITU-T G.711 PCMU, SSRC=0x907B7850, Seq=39971, Time=720
22	23.170869	2001:db8::1	2001:db8::2	RTP	PT=ITU-T G.711 PCMU, SSRC=0x907B7850, Seq=39972, Time=960
33	23.171325	2001:db8::1	2001:db8::2	RTP	PT=ITU-T G.711 PCMU, SSRC=0x907B7850, Seq=39983, Time=3600
34	23.171332	2001:db8::1	2001:db8::2	RTP	PT=ITU-T G.711 PCMU, SSRC=0x907B7850, Seq=39984, Time=3840
35	23.233206	109.182.219.13	160.54.219.109	IP	Fragmented IP protocol (proto=Unknown 0xa0, off=46808, ID=db6d)
36	23.233555	2001:db8::2	2001:db8::4	IP	Bogus IP header length (4, must be at least 20)
37	23.233821	6240:36db:6db6:	46db:6db6:dbd6:	IPv6	[Malformed Packet]
38	23.234061	109.182.219.148	64.54.219.109	IP	Fragmented IP protocol (proto=Unknown 0xa0, off=46808, ID=db6d)
39	23.239543	109.182.219.12	128.54.219.109	IP	Fragmented IP protocol (proto=Semaphore 0x60, off=14040, ID=dc6d)
40	23.239840	109.184.219.19	0.54.219.109	IP	Fragmented IP protocol (proto=IPv6 hop-by-hop option 0x00, off=46808, ID=1b6d)
41	23.240089	5a00:36db:6db6:	36db:6db6:dbd5:	IPv6	[Malformed Packet]
42	23.240324	109.182.219.17	0.54.220.109	IP	Fragmented IP protocol (proto=Merit Internodal 0x20, off=46808, ID=db6d)
43	23.241163	2001:db8::2	2001:db8::4	IP	Bogus IP header length (4, must be at least 20)
44	23.241448	2001:db8::2	2001:db8::4	IP	Bogus IP header length (8, must be at least 20)
45	23.241700	2001:db8::2	2001:db8::4	IP	Bogus IP header length (8, must be at least 20)
46	23.670339	2001:db8::1	2001:db8::2	RTP	PT=ITU-T G.711 PCMU, SSRC=0x907B7850, Seq=39985, Time=4080
47	23.670829	2001:db8::1	2001:db8::2	RTP	PT=ITU-T G.711 PCMU, SSRC=0x907B7850, Seq=39986, Time=4320
48	23.670852	2001:db8::1	2001:db8::2	RTP	PT=ITU-T G.711 PCMU, SSRC=0x907B7850, Seq=39987, Time=4560
61	23.671704	2001:db8::1	2001:db8::2	RTP	PT=ITU-T G.711 PCMU, SSRC=0x907B7850, Seq=40000, Time=7680
62	23.671715	2001:db8::1	2001:db8::2	RTP	PT=ITU-T G.711 PCMU, SSRC=0x907B7850, Seq=40001, Time=7920
63	23.672839	109.182.219.14	160.54.163.113	IP	Fragmented IP protocol (proto=Unknown 0xa0, off=46808, ID=db6d)
64	23.673671	110.205.36.212	32.132.89.109	IP	Fragmented IP protocol (proto=Ether in IP 0x61, off=54544, ID=530e)
65	23.674520	c720:471b:71b7:	371c:8e38:dbd6:	IPv6	[Malformed Packet]
66	23.676748	109.182.219.224	192.54.219.109	IP	Fragmented IP protocol (proto=Novell NCS Heartbeat 0xe0, off=50904, ID=db6d)
67	23.677572	109.182.219.128	192.54.219.113	IP	Fragmented IP protocol (proto=IPv6 hop-by-hop option 0x00, off=46808, ID=1b6e)
68	23.678390	113.182.219.168	160.54.219.109	IP	Fragmented IP protocol (proto=SATNET EXPAK 0x40, off=46808, ID=db8d)
69	23.679748	b220:36db:6db6:	36db:6db6:dbd6:	IPv6	[Malformed Packet]
70	23.683828	2001:db8::2	2001:db8::4	IP	Bogus IP header length (8, must be at least 20)
71	23.684116	109.182.227.13	64.87.26.117	IP	Fragmented IP protocol (proto=IPv6 hop-by-hop option 0x00, off=46808, ID=db6e)
72	23.684364	70a0:3c9c:d1a9:	34e3:6db7:1ad2:	IPv6	[Malformed Packet]
73	23.684606	110.56.219.240	0.70.219.109	IP	Fragmented IP protocol (proto=Unknown 0xc0, off=51424, ID=d36d)
74	23.969232	109.198.180.19	97.200.228.145	IP	Fragmented IP protocol (proto=Datagram Congestion Control Protocol 0x21, off=4680
75	23.974477	2001:db8::4	2001:db8::2	IP	Bogus IP header length (0, must be at least 20)
76	23.998025	2001:db8::2	2001:db8::1	RTP	PT=ITU-T G.711 PCMU, SSRC=0x9078983A, Seq=13657, Time=480
77	24.088253	2001:db8::4	2001:db8::2	IP	Bogus IP header length (8, must be at least 20)
78	24.089374	2001:db8::2	2001:db8::1	RTP	PT=ITU-T G.711 PCMU, SSRC=0x9078983A, Seq=13658, Time=960
79	24.090143	76.205.83.212	224.183.28.205	IP	Fragmented IP protocol (proto=Unknown 0xc0, off=43304, ID=766d)
80	24.102685	2001:db8::4	2001:db8::2	IP	Bogus IP header length (0, must be at least 20)
81	24.117250	2001:db8::2	2001:db8::1	RTP	PT=ITU-T G.711 PCMU, SSRC=0x9078983A, Seq=13659, Time=1440
82	24.166630	2001:db8::4	2001:db8::2	IP	Bogus IP header length (4, must be at least 20)
83	24.170259	2001:db8::1	2001:db8::2	RTP	PT=ITU-T G.711 PCMU, SSRC=0x907B7850, Seq=40002, Time=8160

Figure 8.16: SIP VoIP Adapter packets exchanged in a VoIP call, IPv6



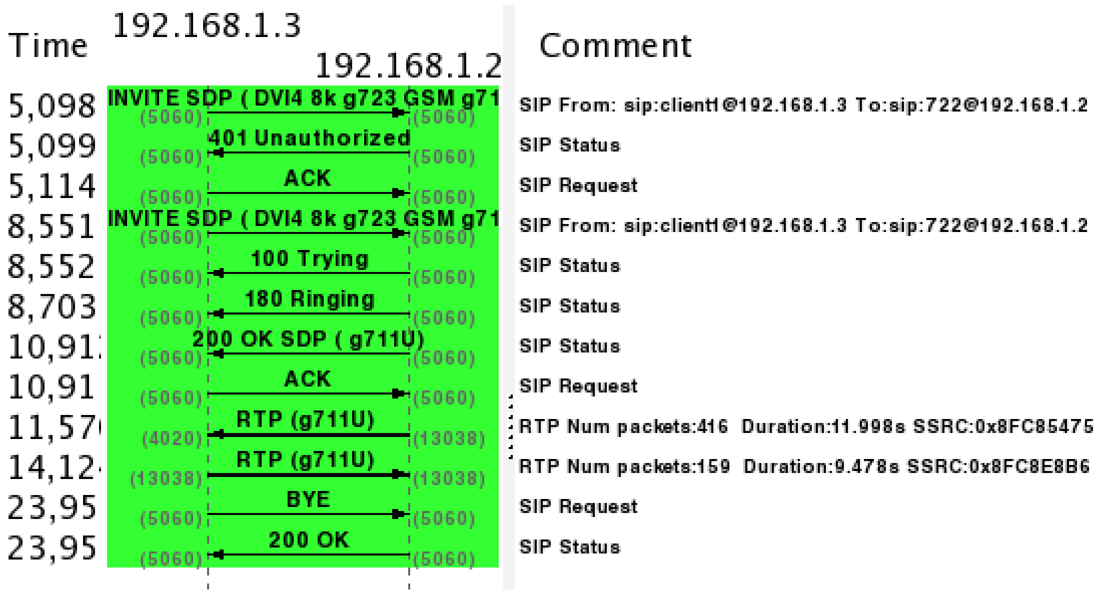
```
▷ Frame 35 (359 bytes on wire, 359 bytes captured)
▷ Ethernet II, Src: AsustekC_75:32:2c (00:18:f3:75:32:2c), Dst: HewlettP_69:fd:eb (00:08:02:69:fd:eb)
▽ Internet Protocol Version 6
  ▷ 0110 .... = Version: 6
    .... 0000 0000 .... .... .... .... .... = Traffic class: 0x00000000
    .... .... .... 0000 0000 0000 0000 0000 = Flowlabel: 0x00000000
    Payload length: 305
    Next header: UDP (0x11)
    Hop limit: 128
    Source: 2001:db8::2 (2001:db8::2)
    Destination: 2001:db8::4 (2001:db8::4)
▷ User Datagram Protocol, Src Port: esimport (3564), Dst Port: cslistener (9000)
▷ Packet Cable Lawful Intercept
▷ Internet Protocol, Src: 109.182.219.132 (109.182.219.132), Dst: 160.54.219.109 (160.54.219.109)
▷ Data (253 bytes)
```

Figure 8.17: Datagram IPv6 packet transmit from SIP VoIP Adapter to EndDevice Test



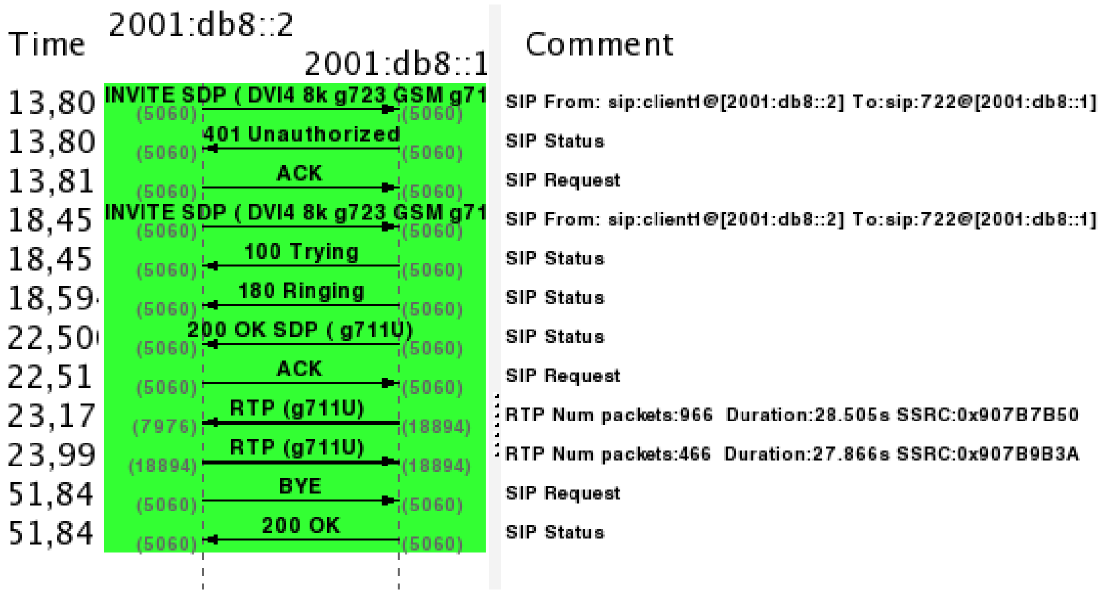
```
▷ Frame 74 (524 bytes on wire, 524 bytes captured)
▷ Ethernet II, Src: HewlettP_69:fd:eb (00:08:02:69:fd:eb), Dst: AsustekC_75:32:2c (00:18:f3:75:32:2c)
▽ Internet Protocol Version 6
  ▷ 0110 .... = Version: 6
    .... 0000 0000 .... .... .... .... .... = Traffic class: 0x00000000
    .... .... .... 0000 0000 0000 0000 0000 = Flowlabel: 0x00000000
    Payload length: 470
    Next header: UDP (0x11)
    Hop limit: 128
    Source: 2001:db8::4 (2001:db8::4)
    Destination: 2001:db8::2 (2001:db8::2)
▷ User Datagram Protocol, Src Port: nim (1058), Dst Port: cslistener (9000)
▷ Packet Cable Lawful Intercept
▷ Internet Protocol, Src: 109.198.180.192 (109.198.180.192), Dst: 97.200.228.145 (97.200.228.145)
▷ Data (422 bytes)
```

Figure 8.18: Datagram IPv6 packet transmit from EndDevice Test to SIP VoIP Adapter



```
Time     192.168.1.3   192.168.1.2                     Comment
5,098    INVITE SDP ( DVI4 8k g723 GSM g71  (5060) -> (5060)   SIP From: sip:client1@192.168.1.3 To:sip:722@192.168.1.2
5,099    401 Unauthorized                  (5060) <- (5060)   SIP Status
5,114    ACK                               (5060) -> (5060)   SIP Request
8,551    INVITE SDP ( DVI4 8k g723 GSM g71  (5060) -> (5060)   SIP From: sip:client1@192.168.1.3 To:sip:722@192.168.1.2
8,552    100 Trying                        (5060) <- (5060)   SIP Status
8,703    180 Ringing                       (5060) <- (5060)   SIP Status
10,91    200 OK SDP ( g711U)               (5060) <- (5060)   SIP Status
10,91    ACK                               (5060) -> (5060)   SIP Request
11,57    RTP (g711U)                       (4020) <- (13038)  RTP Num packets:416  Duration:11.998s SSRC:0x8FC85475
14,12    RTP (g711U)                       (13038) -> (13038) RTP Num packets:159  Duration:9.478s SSRC:0x8FC8E8B6
23,95    BYE                               (5060) -> (5060)   SIP Request
23,95    200 OK                            (5060) <- (5060)   SIP Status
```

Figure 8.19: Communication scheme between SIP VoIP Adapter and Asterisk in IPv4



```
Time     2001:db8::2   2001:db8::1                     Comment
13,80    INVITE SDP ( DVI4 8k g723 GSM g71  (5060) -> (5060)   SIP From: sip:client1@[2001:db8::2] To:sip:722@[2001:db8::1]
13,80    401 Unauthorized                  (5060) <- (5060)   SIP Status
13,81    ACK                               (5060) -> (5060)   SIP Request
18,45    INVITE SDP ( DVI4 8k g723 GSM g71  (5060) -> (5060)   SIP From: sip:client1@[2001:db8::2] To:sip:722@[2001:db8::1]
18,45    100 Trying                        (5060) <- (5060)   SIP Status
18,59    180 Ringing                       (5060) <- (5060)   SIP Status
22,50    200 OK SDP ( g711U)               (5060) <- (5060)   SIP Status
22,51    ACK                               (5060) -> (5060)   SIP Request
23,17    RTP (g711U)                       (7976) <- (18894)  RTP Num packets:966  Duration:28.505s SSRC:0x907B7B50
23,99    RTP (g711U)                       (18894) -> (18894) RTP Num packets:466  Duration:27.866s SSRC:0x907B9B3A
51,84    BYE                               (5060) -> (5060)   SIP Request
51,84    200 OK                            (5060) <- (5060)   SIP Status
```

Figure 8.20: Communication scheme between SIP VoIP Adapter and Asterisk in IPv6

8.6 EndDevice Statistics

The EndDevice Test application has been developed with additional options to observe the audio data that it is being received from SIP VoIP Adapter and transmitted to SIP VoIP Adapter. The user can get access to the statistics in the statics menu bar in main window.

The statistics caught are:

- **UDP Packets Transmitted:** Number of datagram packets transmitted to SIP VoIP Adapter.
- **Total Data Transmitted:** Total number of bytes transmitted to SIP VoIP Adapter, and so, the total number of bytes acquired from the microphone.
- **UDP Packets Received:** Number of datagram packets received from SIP VoIP Adapter.
- **Total Data Received:** Total number of bytes received from SIP VoIP Adapter, and so, the total number of bytes played.
- **Audio Format:** Audio format used for the UDP communication.

It is worth mentioning that these statistics serve only to see the amount of audio sent and received, because in the LAN configured there is no packet loss.

8.7 Summary

In this chapter, the reader is introduced to the configuration of all devices involved in the operation test of the SIP VoIP Adapter. As a main goal of this chapter, we discusses the operations that SIP VoIP application is able to perform in a local network managed by an Asterisk PBX platform, as a registration or call request. We also discussed both communications, the SIP signaling and the audio transmission between the SIP VoIP Adapter and the other end user, through Asterisk. In addition, we discusses the audio exchanged between the SIP VoIP and the EndDevice Test application to test its correct operation.

We have used the network protocol analyzer Wireshark to inspect the packets exchanged over the network.

On the other hand, it is important to emphasize that the communication codecs used for the tests results are: PCMU (μ-law) for the RTP communication and GSM for the UDP communication.

Chapter 9

Conclusions and Recomendations

9.1 Research Conclusions

Throughout the research conducted in this thesis, we learned in depth the main protocols involved in a VoIP communication such as SIP and RTP/RTCP. In addition, we learned to install, set up and manage an Asterisk PBX platform for establishing and controlling a VoIP communication using a SIP signalization over both IPv4 and IPv6.

We succeeded in designing and implementing an application for using a 6lowPAN device as an end device in a voice communication through Asterisk PBX using the SIP signaling protocol. The solution is designed to be very flexible and generic. It can easily be adapted to another end device connected on the computer, in addition to UDP sockets, as for example TCP sockets or through the computer serial port, etc. Furthermore, several protocols can be plugged easily for both RTP and UDP communications thanks to JMF's API flexibility.

Although the SIP VoIP Adapter is designed to establish a voice communication, it has been developed taking into account a future implementation of SIP-based suit of standards for instant messaging and presence information called SIMPLE (Session Initial Protocol for Instant Messaging and Presence Leveraging Extensions).

9.2 Recomendations for future work

There are many options for future development of this project. The most obvious is the configuration of the 6lowPAN USB Router and the 6lowPAN Board Device provided by the Atmel AVR RZ Raven 2.4 GHz Wireless Evaluation Kit. To use the 6lowPAN end device as the sound device of the SIP VoIP Adapter.

Another interesting topic, already mentioned above, is developing the SIP-based suit of standards for instant messaging and presence information. It will let the SIP VoIP Adapter notifies its presence information as well the transmission of text messages.

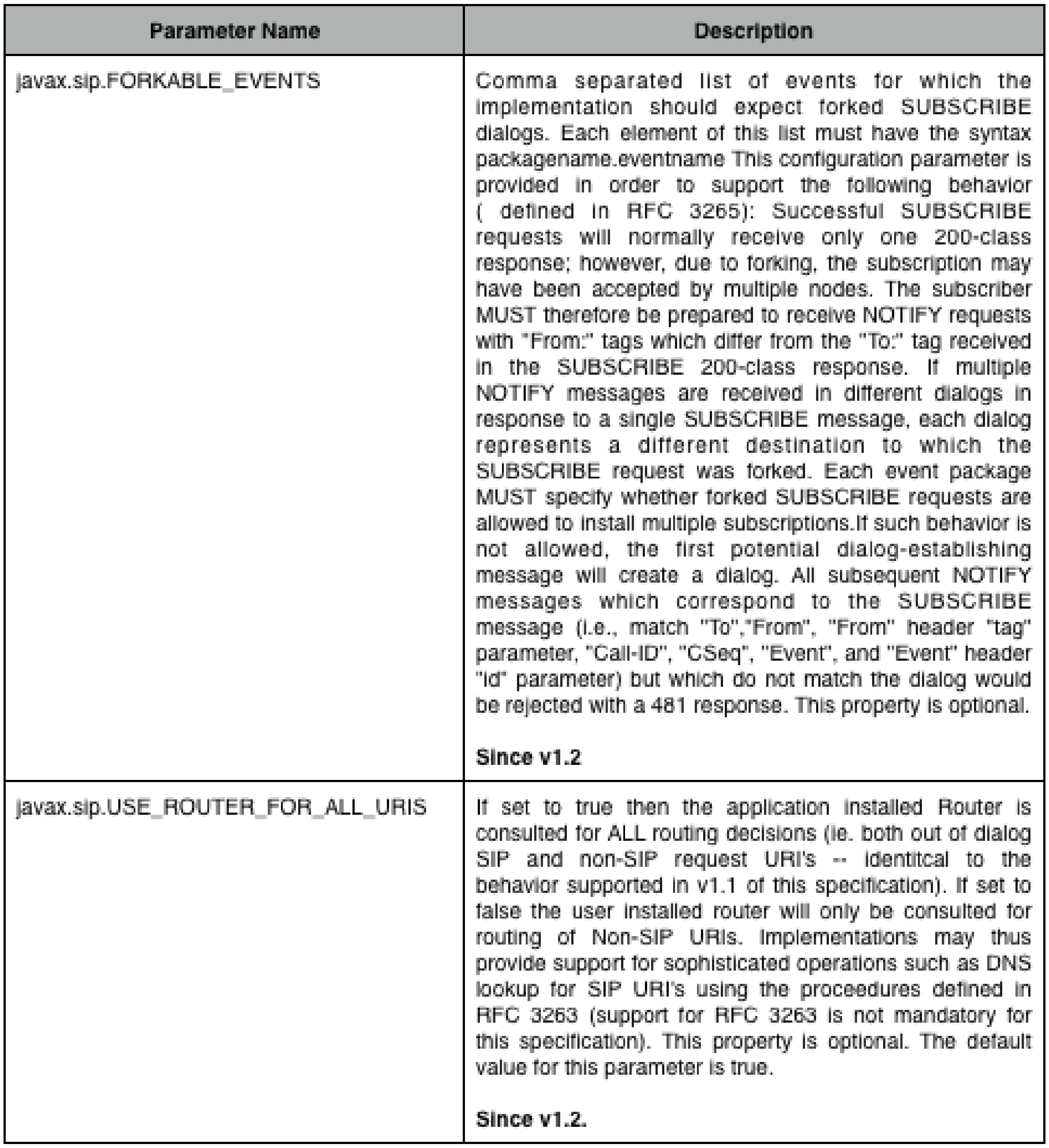
Other interesting option is to implement a communication channel between SIP VoIP Adapter and the 6lowPAN end device with the aim to notify the end device when there is an incoming call, to allow answering the call from the end device.

Appendix A

SIP Stack parameters

Parameter Name	Description
javax.sip.IP_ADDRESS	**Deprecated v1.2.** It is recommended in this specification that the IP Address should be set using the enhanced ListeningPoint architecture, therefore this property is no longer mandatory. When this parameter is specified as null, a singleton stack instance will be created and returned by the SipFactory and the IP Address attributes can be managed via the createListeningPoint(String, int, String) method. For backwards compatability if this flag is set the SipFactory will return a new SipStack instance each time a SipStack is created with a new IP Address. This configuration parameter will become the default IP address of the SipStack. The SIP Factory will return any any existing instance of SipStack that already exist for this IP Address.
javax.sip.STACK_NAME	Sets a user friendly name to identify the underlying stack implementation to the property value i.e. NISTv1.2. The stack name property should contain no spaces. This property is mandatory.
javax.sip.OUTBOUND_PROXY	Sets the outbound proxy of the SIP Stack. The fromat for this string is "ipaddress:port/transport" i.e. 129.1.22.333:5060/UDP. This property is optional.
javax.sip.ROUTER_PATH	Sets the fully qualified classpath to the application supplied Router object that determines how to route messages when the stack cannot make a routing decision (ie. non-sip URIs). In version 1.2 of this specification, out of Dialog SIP URIs are routed by the Routing algorithm defined in RFC 3261 which is implemented internally by the stack provided that javax.sip.USE_ROUTER_FOR_ALL_URIS is set to false. In this case, the installed Router object is consulted for routing decisions pertaining to non-SIP URIs. An application defined Router object must implement the javax.sip.Router interface. This property is optional.
javax.sip.EXTENSION_METHODS	This configuration value informs the underlying implementation of supported extension methods that create new dialog's. This list must not include methods that are natively supported by this specification such as INVITE, SUBSCRIBE and REFER. This configuration flag should only be used for dialog creating extension methods, other extension methods that don't create dialogs can be used using the method parameter on Request assuming the implementation understands the method. If more than one method is supported in this property each extension should be seprated with a colon for example "FOO:BAR". This property is optional.

Parameter Name	Description
javax.sip.RETRANSMISSON_FILTER	**Deprecated v1.2.** Applications can request retransmission alerts from the ServerTransaction.enableRetransmissionAlerts(). The default retransmission behaviour of this specification is dependent on the application core and is defined as follows: - User Agent Client: Retransmissions of ACK Requests are the responsibility of the application. All other retansmissions are handled by the SipProvider. - User Agent Server: Retransmissions of 1xx, 2xx Responses are the responsibility of the application. All other retansmissions are handled by the SipProvider. - Stateful Proxy: As stateful proxies have no Invite transactions all retransmissions are handled by the SipProvider. - Stateless Proxy: As stateless proxies are not transactional all retransmissions are the responsibility of the application and will not be handled the SipProvider. This filter can be viewed as a helper function for User Agents that can be set by an application to prevent the application from handling retransmission of ACK Requests, 1xx and 2xx Responses for INVITE transactions, i.e. the SipProvider will handle the retransmissions. This utility is useful for hiding protocol retransmission semantics from higher level programming environments.
javax.sip.AUTOMATIC_DIALOG_SUPPORT	This property specifies the defined values 'ON' and 'OFF'. The default value is 'ON'. The default behavior represents a common mode of stack operation and allows the construction of simple user agents. This property is optional. This is summarized as: - A dialog gets created on a dialog creating transaction. - The first respose having both a From and a To tag creates the transaction. - The first 2xx response to the transaction will drive the dialog to the CONFIRMED state. The ability to turn of dialog support is motivated by dialog free servers (such as proxy servers) that do not want to pay the overhead of the dialog layer and user agents that may want to create multiple dialogs for a single INVITE (as a result of forking by proxy servers). The following behavior is defined when the configuration parameter is set to 'OFF'. - The application is responsible to create the Dialog if desired. - The application may create a Dialog and associate it with a response (provisional or final) of a dialog creating request.



Parameter Name	Description
javax.sip.FORKABLE_EVENTS	Comma separated list of events for which the implementation should expect forked SUBSCRIBE dialogs. Each element of this list must have the syntax packagename.eventname This configuration parameter is provided in order to support the following behavior (defined in RFC 3265): Successful SUBSCRIBE requests will normally receive only one 200-class response; however, due to forking, the subscription may have been accepted by multiple nodes. The subscriber MUST therefore be prepared to receive NOTIFY requests with "From:" tags which differ from the "To:" tag received in the SUBSCRIBE 200-class response. If multiple NOTIFY messages are received in different dialogs in response to a single SUBSCRIBE message, each dialog represents a different destination to which the SUBSCRIBE request was forked. Each event package MUST specify whether forked SUBSCRIBE requests are allowed to install multiple subscriptions.If such behavior is not allowed, the first potential dialog-establishing message will create a dialog. All subsequent NOTIFY messages which correspond to the SUBSCRIBE message (i.e., match "To","From", "From" header "tag" parameter, "Call-ID", "CSeq", "Event", and "Event" header "id" parameter) but which do not match the dialog would be rejected with a 481 response. This property is optional. **Since v1.2**
javax.sip.USE_ROUTER_FOR_ALL_URIS	If set to true then the application installed Router is consulted for ALL routing decisions (ie. both out of dialog SIP and non-SIP request URI's -- identitcal to the behavior supported in v1.1 of this specification). If set to false the user installed router will only be consulted for routing of Non-SIP URIs. Implementations may thus provide support for sophisticated operations such as DNS lookup for SIP URI's using the proceedures defined in RFC 3263 (support for RFC 3263 is not mandatory for this specification). This property is optional. The default value for this parameter is true. **Since v1.2.**

Appendix B

SIP messages exchange

B.1 SIP Registration and Call messages exchange in IPv4

```
REGISTER sip:192.168.1.2:5060 SIP/2.0
Call-ID: 5b5191767defbb3a2f5827f86a5982f9@192.168.1.3
From: <sip:client1@192.168.1.3>;tag=1496
To: <sip:client1@192.168.1.3>
Via: SIP/2.0/UDP 192.168.1.3:5060
Max-Forwards: 2
Contact: <sip:client1@192.168.1.3:5060;transport=udp>;expires=600
CSeq: 1 REGISTER
Content-Length: 0

SIP/2.0 401 Unauthorized
Via: SIP/2.0/UDP 192.168.1.3:5060;branch=z9hG4bKcdeae64257907955639ae692d9f58cc3;
received=192.168.1.3
From: <sip:client1@192.168.1.3>;tag=1496
To: <sip:client1@192.168.1.3>;tag=as2edcce66
Call-ID: 5b5191767defbb3a2f5827f86a5982f9@192.168.1.3
CSeq: 1 REGISTER
User-Agent: Asterisk PBX asteriskv6-20080107
Allow: INVITE,ACK,CANCEL,OPTIONS,BYE,REFER,SUBSCRIBE,NOTIFY
Supported: replaces
WWW-Authenticate: Digest nonce="614ca32c",realm="asterisk",algorithm=MD5
Content-Length: 0

REGISTER sip:192.168.1.2:5060 SIP/2.0
Call-ID: 5b5191767defbb3a2f5827f86a5982f9@192.168.1.3
From: <sip:client1@192.168.1.3>;tag=1496
To: <sip:client1@192.168.1.3>
```

```
Max-Forwards: 2
Contact: <sip:client1@192.168.1.3:5060;transport=udp>;expires=600
CSeq: 2 REGISTER
Authorization: Digest response="bc6b3b80c2821eef370f35ab805c1d68",username="client1",
nonce="614ca32c",realm="asterisk",opaque="",uri="sip:192.168.1.2:5060",algorithm=MD5
Via: SIP/2.0/UDP 192.168.1.3:5060;branch=z9hG4bK227eb2e33e771df2ad1db72f55ec0641
Content-Length: 0

REGISTER sip:192.168.1.2:5060 SIP/2.0
Call-ID: 5b5191767defbb3a2f5827f86a5982f9@192.168.1.3
From: <sip:client1@192.168.1.3>;tag=1496
To: <sip:client1@192.168.1.3>
Max-Forwards: 2
Contact: <sip:client1@192.168.1.3:5060;transport=udp>;expires=600
CSeq: 2 REGISTER
Authorization: Digest response="bc6b3b80c2821eef370f35ab805c1d68",username="client1",
nonce="614ca32c",realm="asterisk",opaque="",uri="sip:192.168.1.2:5060",algorithm=MD5
Via: SIP/2.0/UDP 192.168.1.3:5060;branch=z9hG4bK227eb2e33e771df2ad1db72f55ec0641
Content-Length: 0

SIP/2.0 200 OK
Via: SIP/2.0/UDP 192.168.1.3:5060;branch=z9hG4bK227eb2e33e771df2ad1db72f55ec0641;
received=192.168.1.3
From: <sip:client1@192.168.1.3>;tag=1496
To: <sip:client1@192.168.1.3>;tag=as2edcce66
Call-ID: 5b5191767defbb3a2f5827f86a5982f9@192.168.1.3
CSeq: 2 REGISTER
User-Agent: Asterisk PBX asteriskv6-20080107
Allow: INVITE,ACK,CANCEL,OPTIONS,BYE,REFER,SUBSCRIBE,NOTIFY
Supported: replaces
Expires: 600
Contact: <sip:client1@192.168.1.3:5060;transport=udp>;expires=600
Date: Tue, 15 Dec 2009 01:03:59 GMT
Content-Length: 0

INVITE sip:722@192.168.1.2;transport=udp SIP/2.0
Call-ID: 29aa52b3684051072ff2f47e94a3181f@192.168.1.3
CSeq: 1 INVITE
From: <sip:client1@192.168.1.3>;tag=4689
To: <sip:722@192.168.1.2>
Via: SIP/2.0/UDP 192.168.1.3:5060;branch=z9hG4bK6c75806ac7346be70d9052787cbe6b88
Max-Forwards: 2
Contact: <sip:client1@192.168.1.3:5060;transport=udp>
Content-Type: application/sdp
Content-Length: 113

v=0
o=client1 161199 162568 IN IP4 192.168.1.3
s=-
c=IN IP4 192.168.1.3
t=0 0
m=audio 1526 RTP/AVP 5 4 3 0
```

```
SIP/2.0 401 Unauthorized
Via: SIP/2.0/UDP 192.168.1.3:5060;branch=z9hG4bK6c75806ac7346be70d9052787cbe6b88;
received=192.168.1.3
From: <sip:client1@192.168.1.3>;tag=4689
To: <sip:722@192.168.1.2>;tag=as2a7d0cdd
Call-ID: 29aa52b3684051072ff2f47e94a3181f@192.168.1.3
CSeq: 1 INVITE
User-Agent: Asterisk PBX asteriskv6-20080107
Allow: INVITE,ACK,CANCEL,OPTIONS,BYE,REFER,SUBSCRIBE,NOTIFY
Supported: replaces
WWW-Authenticate: Digest nonce="63ca95f8",realm="asterisk",algorithm=MD5
Content-Length: 0

INVITE sip:722@192.168.1.2;transport=udp SIP/2.0
Call-ID: 29aa52b3684051072ff2f47e94a3181f@192.168.1.3
CSeq: 2 INVITE
From: <sip:client1@192.168.1.3>;tag=4689
To: <sip:722@192.168.1.2>
Max-Forwards: 2
Contact: <sip:client1@192.168.1.3:5060;transport=udp>
Content-Type: application/sdp
Authorization: Digest response="d5a6070d81ff683d21e086dd7046ecb5",username="client1",
nonce="63ca95f8",realm="asterisk",opaque="",uri="sip:192.168.1.2:5060",algorithm=MD5
Via: SIP/2.0/UDP 192.168.1.3:5060;branch=z9hG4bKaeac6411967901d8217099f146839669
Content-Length: 113

v=0
o=client1 161199 162568 IN IP4 192.168.1.3
s=-
c=IN IP4 192.168.1.3
t=0 0
m=audio 1526 RTP/AVP 5 4 3 0

SIP/2.0 100 Trying
Via: SIP/2.0/UDP 192.168.1.3:5060;branch=z9hG4bKaeac6411967901d8217099f146839669;
received=192.168.1.3
From: <sip:client1@192.168.1.3>;tag=4689
To: <sip:722@192.168.1.2>
Call-ID: 29aa52b3684051072ff2f47e94a3181f@192.168.1.3
CSeq: 2 INVITE
User-Agent: Asterisk PBX asteriskv6-20080107
Allow: INVITE,ACK,CANCEL,OPTIONS,BYE,REFER,SUBSCRIBE,NOTIFY
Supported: replaces
Contact: <sip:722@192.168.1.2>
Content-Length: 0

SIP/2.0 180 Ringing
Via: SIP/2.0/UDP 192.168.1.3:5060;branch=z9hG4bKaeac6411967901d8217099f146839669;
received=192.168.1.3
From: <sip:client1@192.168.1.3>;tag=4689
To: <sip:722@192.168.1.2>;tag=as0a1f9946
```

```
Call-ID: 29aa52b3684051072ff2f47e94a3181f@192.168.1.3
CSeq: 2 INVITE
User-Agent: Asterisk PBX asteriskv6-20080107
Allow: INVITE,ACK,CANCEL,OPTIONS,BYE,REFER,SUBSCRIBE,NOTIFY
Supported: replaces
Contact: <sip:722@192.168.1.2>
Content-Length: 0

SIP/2.0 200 OK
Via: SIP/2.0/UDP 192.168.1.3:5060;branch=z9hG4bKaeac6411967901d8217099f146839669;
received=192.168.1.3
From: <sip:client1@192.168.1.3>;tag=4689
To: <sip:722@192.168.1.2>;tag=as0a1f9946
Call-ID: 29aa52b3684051072ff2f47e94a3181f@192.168.1.3
CSeq: 2 INVITE
User-Agent: Asterisk PBX asteriskv6-20080107
Allow: INVITE,ACK,CANCEL,OPTIONS,BYE,REFER,SUBSCRIBE,NOTIFY
Supported: replaces
Contact: <sip:722@192.168.1.2>
Content-Type: application/sdp
Content-Length: 217

v=0
o=root 1223686393 1223686393 IN IP4 192.168.1.2
s=Asterisk PBX asteriskv6-20080107
c=IN IP4 192.168.1.2
t=0 0
m=audio 10652 RTP/AVP 0
a=rtpmap:0 PCMU/8000
a=silenceSupp:off - - - -
a=ptime:20
a=sendrecv

Sending ACK :
ACK sip:722@192.168.1.2 SIP/2.0
Via: SIP/2.0/UDP 192.168.1.3:5060;branch=z9hG4bKdb4bd9167244ad31992a051ff9a7b1b0
CSeq: 2 ACK
From: <sip:client1@192.168.1.3>;tag=4689
To: <sip:722@192.168.1.2>;tag=as0a1f9946
Call-ID: 29aa52b3684051072ff2f47e94a3181f@192.168.1.3
User-Agent: Asterisk PBX asteriskv6-20080107
Allow: INVITE,ACK,CANCEL,OPTIONS,BYE,REFER,SUBSCRIBE,NOTIFY
Supported: replaces
Max-Forwards: 70
Content-Length: 0

Start Data Transmission...

Track 0 is set to transmit as:
  ULAW/rtp, 8000.0 Hz, 8-bit, Mono, FrameSize=8 bits
Created RTP session: 192.168.1.2 dest 10652
```

```
  - reading audio data from socket. Data = 462
  - reading audio data from socket. Data = 462
  - reading audio data from socket. Data = 462
  - reading audio data from socket. Data = 462
  - reading audio data from socket. Data = 462

  - Recevied new RTP stream: ULAW/rtp, 8000.0 Hz, 8-bit, Mono
format list : 0
      The sender of this stream had yet to be identified.

  - reading audio data from socket. Data = 462
  - reading audio data from socket. Data = 462
  - reading audio data from socket. Data = 462
Track 0 is set to transmit as:
  gsm, 44100.0 Hz, 16-bit, Mono
In DataSourceHandler: instanceof PushBufferDataSource
  - reading audio data from socket. Data = 462
gsm, 44100.0 Hz, 16-bit, Mono, FrameSize=264 bits
Buffer object writed into byte array. Buffer length = 297
Buffer item: 1
gsm, 44100.0 Hz, 16-bit, Mono, FrameSize=264 bits
Buffer object writed into byte array. Buffer length = 297
Buffer item: 2
...
gsm, 44100.0 Hz, 16-bit, Mono, FrameSize=264 bits
Buffer object writed into byte array. Buffer length = 297
Buffer item: 103
  - reading audio data from socket. Data = 462
  - reading audio data from socket. Data = 462
  - reading audio data from socket. Data = 462
...

Finish Data Transmission...

BYE sip:722@192.168.1.2 SIP/2.0
Via: SIP/2.0/UDP 192.168.1.3:5060;branch=z9hG4bKaa1cc1eb21897d97c8be725e1fc572e0
CSeq: 2 BYE
From: <sip:client1@192.168.1.3>;tag=4689
To: <sip:722@192.168.1.2>
Call-ID: 29aa52b3684051072ff2f47e94a3181f@192.168.1.3
User-Agent: Asterisk PBX asteriskv6-20080107
Allow: INVITE,ACK,CANCEL,OPTIONS,BYE,REFER,SUBSCRIBE,NOTIFY
Supported: replaces
WWW-Authenticate: Digest nonce="63ca95f8",realm="asterisk",algorithm=MD5
Max-Forwards: 70
Content-Length: 0

SIP/2.0 200 OK
Via: SIP/2.0/UDP 192.168.1.3:5060;branch=z9hG4bKaa1cc1eb21897d97c8be725e1fc572e0;
received=192.168.1.3
From: <sip:client1@192.168.1.3>;tag=4689
To: <sip:722@192.168.1.2>;tag=as0a1f9946
```

```
Call-ID: 29aa52b3684051072ff2f47e94a3181f@192.168.1.3
CSeq: 2 BYE
User-Agent: Asterisk PBX asteriskv6-20080107
Allow: INVITE,ACK,CANCEL,OPTIONS,BYE,REFER,SUBSCRIBE,NOTIFY
Supported: replaces
Contact: <sip:722@192.168.1.2>
Content-Length: 0
```

B.2 SIP Registration and Call messages exchange in IPv6

```
REGISTER sip:[2001:db8::1]:5060 SIP/2.0
Call-ID: 3fb2e7b18f0397efd44583e70e8cbfaa@2001:db8:0:0:0:0:0:2
From: <sip:client1@[2001:db8::2]>;tag=2306
To: <sip:client1@[2001:db8::2]>
Via: SIP/2.0/UDP [2001:db8:0:0:0:0:0:2]:5060
Max-Forwards: 2
Contact: <sip:client1@[2001:db8:0:0:0:0:0:2]:5060;transport=udp>;expires=600
CSeq: 1 REGISTER
Content-Length: 0

SIP/2.0 401 Unauthorized
Via: SIP/2.0/UDP [2001:db8:0:0:0:0:0:2]:5060;branch=z9hG4bK4e98f1deaccf2598f6649f605c58097d;
received=2001:db8::2
From: <sip:client1@[2001:db8::2]>;tag=2306
To: <sip:client1@[2001:db8::2]>;tag=as4fa0014e
Call-ID: 3fb2e7b18f0397efd44583e70e8cbfaa@2001:db8:0:0:0:0:0:2
CSeq: 1 REGISTER
User-Agent: Asterisk PBX asteriskv6-20080107
Allow: INVITE,ACK,CANCEL,OPTIONS,BYE,REFER,SUBSCRIBE,NOTIFY
Supported: replaces
WWW-Authenticate: Digest nonce="15cefd15",realm="asterisk",algorithm=MD5
Content-Length: 0

REGISTER sip:[2001:db8::1]:5060 SIP/2.0
Call-ID: 3fb2e7b18f0397efd44583e70e8cbfaa@2001:db8:0:0:0:0:0:2
From: <sip:client1@[2001:db8::2]>;tag=2306
To: <sip:client1@[2001:db8::2]>
Max-Forwards: 2
Contact: <sip:client1@[2001:db8:0:0:0:0:0:2]:5060;transport=udp>;expires=600
CSeq: 2 REGISTER
Authorization: Digest response="821dae77f8674c833e8808084fa26e63",username="client1",
nonce="15cefd15",realm="asterisk",opaque="",uri="sip:[2001:db8::1]:5060",algorithm=MD5
Via: SIP/2.0/UDP [2001:db8:0:0:0:0:0:2]:5060;branch=z9hG4bKf9fc56a9d9b24ad3cddf2c1a922cadf9
Content-Length: 0

REGISTER sip:[2001:db8::1]:5060 SIP/2.0
```

```
Call-ID: 3fb2e7b18f0397efd44583e70e8cbfaa@2001:db8:0:0:0:0:0:2
From: <sip:client1@[2001:db8::2]>;tag=2306
To: <sip:client1@[2001:db8::2]>
Max-Forwards: 2
Contact: <sip:client1@[2001:db8:0:0:0:0:0:2]:5060;transport=udp>;expires=600
CSeq: 2 REGISTER
Authorization: Digest response="821dae77f8674c833e8808084fa26e63",username="client1",
nonce="15cefd15",realm="asterisk",opaque="",uri="sip:[2001:db8::1]:5060",algorithm=MD5
Via: SIP/2.0/UDP [2001:db8:0:0:0:0:0:2]:5060;branch=z9hG4bKf9fc56a9d9b24ad3cddf2c1a922cadf9
Content-Length: 0

SIP/2.0 200 OK
Via: SIP/2.0/UDP [2001:db8:0:0:0:0:0:2]:5060;branch=z9hG4bKf9fc56a9d9b24ad3cddf2c1a922cadf9;
received=2001:db8::2
From: <sip:client1@[2001:db8::2]>;tag=2306
To: <sip:client1@[2001:db8::2]>;tag=as4fa0014e
Call-ID: 3fb2e7b18f0397efd44583e70e8cbfaa@2001:db8:0:0:0:0:0:2
CSeq: 2 REGISTER
User-Agent: Asterisk PBX asteriskv6-20080107
Allow: INVITE,ACK,CANCEL,OPTIONS,BYE,REFER,SUBSCRIBE,NOTIFY
Supported: replaces
Expires: 600
Contact: <sip:client1@[2001:db8:0:0:0:0:0:2]:5060;transport=udp>;expires=600
Date: Tue, 15 Dec 2009 03:56:50 GMT
Content-Length: 0

INVITE sip:722@[2001:db8::1];transport=udp SIP/2.0
Call-ID: f4862f350d3ddea0223db97812bc748d@2001:db8:0:0:0:0:0:2
CSeq: 1 INVITE
From: <sip:client1@[2001:db8::2]>;tag=7034
To: <sip:722@[2001:db8::1]>
Via: SIP/2.0/UDP [2001:db8:0:0:0:0:0:2]:5060;branch=z9hG4bKce8d589ad5992cc7a1baf6ebaacc5482
Max-Forwards: 2
Contact: <sip:client1@[2001:db8:0:0:0:0:0:2]:5060;transport=udp>
Content-Type: application/sdp
Content-Length: 133

v=0
o=client1 564457 565826 IN IP6 2001:db8:0:0:0:0:0:2
s=-
c=IN IP6 2001:db8:0:0:0:0:0:2
t=0 0
m=audio 7976 RTP/AVP 5 4 3 0

SIP/2.0 401 Unauthorized
Via: SIP/2.0/UDP [2001:db8:0:0:0:0:0:2]:5060;branch=z9hG4bKce8d589ad5992cc7a1baf6ebaacc5482;
received=2001:db8::2
From: <sip:client1@[2001:db8::2]>;tag=7034
To: <sip:722@[2001:db8::1]>;tag=as383f5ce7
Call-ID: f4862f350d3ddea0223db97812bc748d@2001:db8:0:0:0:0:0:2
CSeq: 1 INVITE
User-Agent: Asterisk PBX asteriskv6-20080107
```

```
Allow: INVITE,ACK,CANCEL,OPTIONS,BYE,REFER,SUBSCRIBE,NOTIFY
Supported: replaces
WWW-Authenticate: Digest nonce="3760c1c5",realm="asterisk",algorithm=MD5
Content-Length: 0

INVITE sip:722@[2001:db8::1];transport=udp SIP/2.0
Call-ID: f4862f350d3ddea0223db97812bc748d@2001:db8:0:0:0:0:0:2
CSeq: 2 INVITE
From: <sip:client1@[2001:db8::2]>;tag=7034
To: <sip:722@[2001:db8::1]>
Max-Forwards: 2
Contact: <sip:client1@[2001:db8:0:0:0:0:0:2]:5060;transport=udp>
Content-Type: application/sdp
Authorization: Digest response="6b65fe2e5b3e99e3e09b7b9b25056977",username="client1",
nonce="3760c1c5",realm="asterisk",opaque="",uri="sip:[2001:db8::1]:5060",algorithm=MD5
Via: SIP/2.0/UDP [2001:db8:0:0:0:0:0:2]:5060;branch=z9hG4bKfa347561a9f5dea79334319e64f69bb9
Content-Length: 133

v=0
o=client1 564457 565826 IN IP6 2001:db8:0:0:0:0:0:2
s=-
c=IN IP6 2001:db8:0:0:0:0:0:2
t=0 0
m=audio 7976 RTP/AVP 5 4 3 0

INVITE sip:722@[2001:db8::1];transport=udp SIP/2.0
Call-ID: f4862f350d3ddea0223db97812bc748d@2001:db8:0:0:0:0:0:2
CSeq: 2 INVITE
From: <sip:client1@[2001:db8::2]>;tag=7034
To: <sip:722@[2001:db8::1]>
Max-Forwards: 2
Contact: <sip:client1@[2001:db8:0:0:0:0:0:2]:5060;transport=udp>
Content-Type: application/sdp
Authorization: Digest response="6b65fe2e5b3e99e3e09b7b9b25056977",username="client1",
nonce="3760c1c5",realm="asterisk",opaque="",uri="sip:[2001:db8::1]:5060",algorithm=MD5
Via: SIP/2.0/UDP [2001:db8:0:0:0:0:0:2]:5060;branch=z9hG4bKfa347561a9f5dea79334319e64f69bb9
Content-Length: 133

v=0
o=client1 564457 565826 IN IP6 2001:db8:0:0:0:0:0:2
s=-
c=IN IP6 2001:db8:0:0:0:0:0:2
t=0 0
m=audio 7976 RTP/AVP 5 4 3 0

SIP/2.0 100 Trying
Via: SIP/2.0/UDP [2001:db8:0:0:0:0:0:2]:5060;branch=z9hG4bKfa347561a9f5dea79334319e64f69bb9;
received=2001:db8::2
From: <sip:client1@[2001:db8::2]>;tag=7034
To: <sip:722@[2001:db8::1]>
Call-ID: f4862f350d3ddea0223db97812bc748d@2001:db8:0:0:0:0:0:2
CSeq: 2 INVITE
```

```
User-Agent: Asterisk PBX asteriskv6-20080107
Allow: INVITE,ACK,CANCEL,OPTIONS,BYE,REFER,SUBSCRIBE,NOTIFY
Supported: replaces
Contact: <sip:722@[2001:db8::1]>
Content-Length: 0

SIP/2.0 180 Ringing
Via: SIP/2.0/UDP [2001:db8:0:0:0:0:0:2]:5060;branch=z9hG4bKfa347561a9f5dea79334319e64f69bb9;
received=2001:db8::2
From: <sip:client1@[2001:db8::2]>;tag=7034
To: <sip:722@[2001:db8::1]>;tag=as269e7101
Call-ID: f4862f350d3ddea0223db97812bc748d@2001:db8:0:0:0:0:0:2
CSeq: 2 INVITE
User-Agent: Asterisk PBX asteriskv6-20080107
Allow: INVITE,ACK,CANCEL,OPTIONS,BYE,REFER,SUBSCRIBE,NOTIFY
Supported: replaces
Contact: <sip:722@[2001:db8::1]>
Content-Length: 0

SIP/2.0 200 OK
Via: SIP/2.0/UDP [2001:db8:0:0:0:0:0:2]:5060;branch=z9hG4bKfa347561a9f5dea79334319e64f69bb9;
received=2001:db8::2
From: <sip:client1@[2001:db8::2]>;tag=7034
To: <sip:722@[2001:db8::1]>;tag=as269e7101
Call-ID: f4862f350d3ddea0223db97812bc748d@2001:db8:0:0:0:0:0:2
CSeq: 2 INVITE
User-Agent: Asterisk PBX asteriskv6-20080107
Allow: INVITE,ACK,CANCEL,OPTIONS,BYE,REFER,SUBSCRIBE,NOTIFY
Supported: replaces
Contact: <sip:722@[2001:db8::1]>
Content-Type: application/sdp
Content-Length: 217

v=0
o=root 1235654005 1235654005 IN IP6 2001:db8::1
s=Asterisk PBX asteriskv6-20080107
c=IN IP6 2001:db8::1
t=0 0
m=audio 18894 RTP/AVP 0
a=rtpmap:0 PCMU/8000
a=silenceSupp:off - - - -
a=ptime:20
a=sendrecv

ACK sip:722@[2001:db8::1] SIP/2.0
Via: SIP/2.0/UDP [2001:db8:0:0:0:0:0:2]:5060;branch=z9hG4bKaaa97a6964345da7055f3419583c0425
CSeq: 2 ACK
From: <sip:client1@[2001:db8::2]>;tag=7034
To: <sip:722@[2001:db8::1]>;tag=as269e7101
Call-ID: f4862f350d3ddea0223db97812bc748d@2001:db8:0:0:0:0:0:2
User-Agent: Asterisk PBX asteriskv6-20080107
Allow: INVITE,ACK,CANCEL,OPTIONS,BYE,REFER,SUBSCRIBE,NOTIFY
```

```
Supported: replaces
Max-Forwards: 70
Content-Length: 0

Start Data Transmission...

Track 0 is set to transmit as:
  ULAW/rtp, 8000.0 Hz, 8-bit, Mono, FrameSize=8 bits
  - Recevied new RTP stream: ULAW/rtp, 8000.0 Hz, 8-bit, Mono
      The sender of this stream had yet to be identified.
Track 0 is set to transmit as:
  gsm, 44100.0 Hz, 16-bit, Mono

gsm, 44100.0 Hz, 16-bit, Mono, FrameSize=264 bits
Buffer object writed into byte array. Buffer length = 297
Buffer item: 1
gsm, 44100.0 Hz, 16-bit, Mono, FrameSize=264 bits
Buffer object writed into byte array. Buffer length = 297
Buffer item: 2
gsm, 44100.0 Hz, 16-bit, Mono, FrameSize=264 bits
Buffer object writed into byte array. Buffer length = 297
Buffer item: 3
...
  - reading audio data from socket. Data = 462
  - reading audio data from socket. Data = 462
  - reading audio data from socket. Data = 462
...
gsm, 44100.0 Hz, 16-bit, Mono, FrameSize=264 bits
Buffer object writed into byte array. Buffer length = 297
Buffer item: 23

Fisish Data Transmission...

BYE sip:722@[2001:db8::1] SIP/2.0
Via: SIP/2.0/UDP [2001:db8:0:0:0:0:0:2]:5060;branch=z9hG4bK54bedf1ac89a5c18fb7328013d3b5d16
CSeq: 2 BYE
From: <sip:client1@[2001:db8::2]>;tag=1213
To: <sip:722@[2001:db8::1]>
Call-ID: c07be06bedf086b6d6aa53ac69c01789@2001:db8:0:0:0:0:0:2
User-Agent: Asterisk PBX asteriskv6-20080107
Allow: INVITE,ACK,CANCEL,OPTIONS,BYE,REFER,SUBSCRIBE,NOTIFY
Supported: replaces
WWW-Authenticate: Digest nonce="7f1aa68d",realm="asterisk",algorithm=MD5
Max-Forwards: 70
Content-Length: 0

SIP/2.0 200 OK
Via: SIP/2.0/UDP [2001:db8:0:0:0:0:0:2]:5060;branch=z9hG4bK54bedf1ac89a5c18fb7328013d3b5d16;
received=2001:db8::2
From: <sip:client1@[2001:db8::2]>;tag=1213
To: <sip:722@[2001:db8::1]>;tag=as1b9369e5
Call-ID: c07be06bedf086b6d6aa53ac69c01789@2001:db8:0:0:0:0:0:2
```

```
CSeq: 2 BYE
User-Agent: Asterisk PBX asteriskv6-20080107
Allow: INVITE,ACK,CANCEL,OPTIONS,BYE,REFER,SUBSCRIBE,NOTIFY
Supported: replaces
Contact: <sip:722@[2001:db8::1]>
Content-Length: 0
```

Appendix C

SIP VoIP Adapter: How to use

C.1 Introduction

This chapter describes how to use the SIP VoIP Adapter in a system handled by an Asterisk PBX. The building process and the configuration are explained as well as the running of the aplication. Finally, an overview is given on how to extend SIP VoIP Adapter with new features.

C.2 Platform requirements

The following software setup is necessary in order to build and run the project:

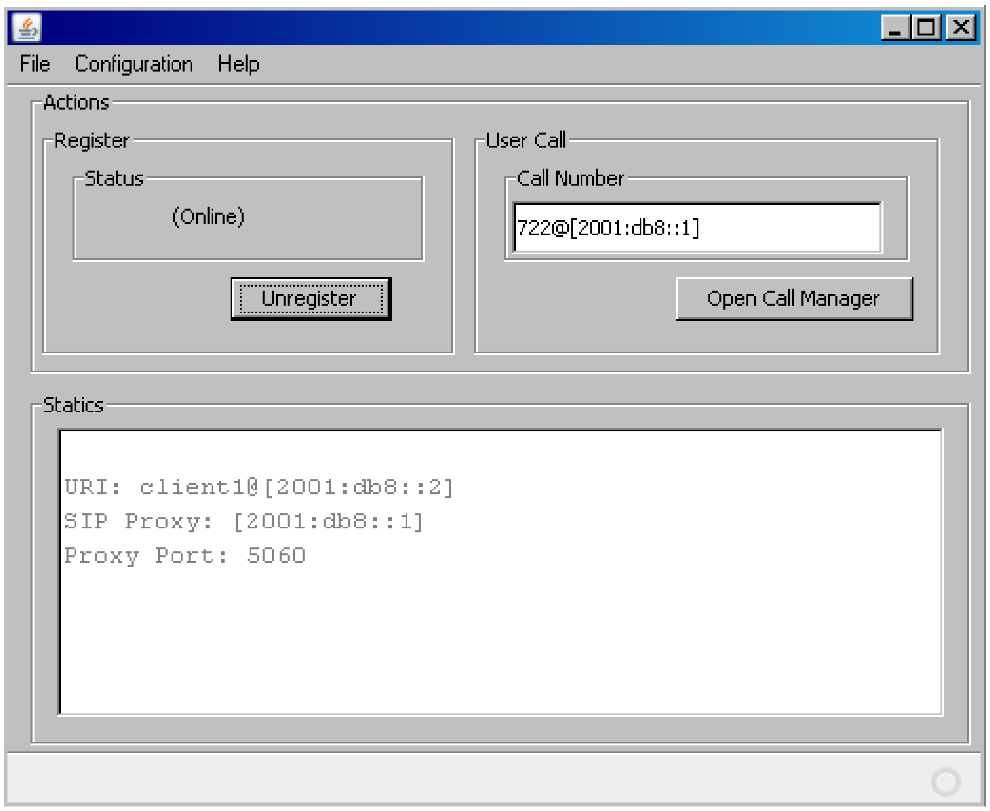
- Microsoft Windows XP Service Pack 2 is prefered. However, Linux, Solaris, Mac OS X Operative System are also able to run the SIP VoIP Adapter. See `http://www.sun.com/bigadmin/jsp/descFile.jsp?url=descAll/ipv6_rtsp_rtp_strea` for running the application in another Operating System instead MS Windows XP.
- IPv6 installed.
- Java Development Kit (JDK) 1.5 or higher.
- Java Media Frameworks 2.1.1.

C.3 Editing SIP VoIP Adapter

The SIP VoIP Adapter application has been developed using the Netbeans IDE platform version 6.7. Thus, the folder provided with the Master Thesis CD called SIPVoIPAdapterIPv6 represents the Project Folder in a Netbeans IDE. Therefore, to open the application in order to modify it, just open an exisiting project and select the SIPVoIPAdapterfolder as a project.

C.4 Running SIP VoIP Adapter

To buid and run the application in Windows XP SP2 go Start, Run and type ”cmd” and a command prompt will be opened. Go to /SIPVoIPAdapterIPV6/dist/ and check if file SIPVoIPAdapterIPV6.jar exists, then type ”java -jar SIPVoIPAdapterIPv6.jar”, and the application main window will be opened, see Figure C.1. Some details of the execution and its subsequent operation will be shown in the console window already opened.



722@[2001:db8::1]

```
URI: client1@[2001:db8::2]
SIP Proxy: [2001:db8::1]
Proxy Port: 5060
```

Figure C.1: SIP VoIP Adapter main window example

C.5 Configurating SIP VoIP Adapter

Once the application has been executed. Go to menu bar and select ”Configuration” click in ”User Configuration”. A new window will be opened with the following required fields:

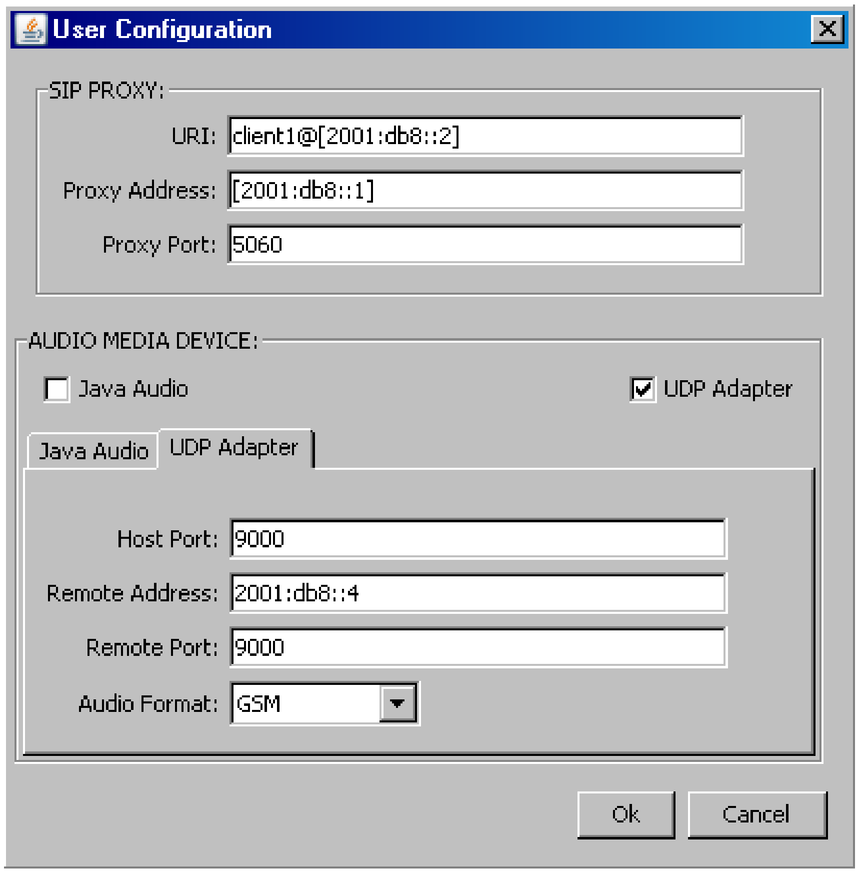
- **Sip Proxy:**
  - **URI:** Local SIP identity as:

    sip:[user name]@[local host IP address]
  - **Proxy Address:** The outbound via proxy address (IP address where Asterisk PBX is installed)
  - **Proxy Port:** The port number where SIP messages are transmitted (aka Asterisk listening port).
- **Audio Mendia Device**
  - **Java Audio:** Computer sound standard device
  - **Adapter:**
    - **Host Port:** Listening port number for incoming UDP audio data.
    - **Remote Address:** Remote IP address for sending to and acquiring from the audio data over UDP.
    - **Remote Port:** Remote port number for transmiting the audio data over a socket UDP.
    - **Audio Format:** Codec used on the UDP communication.

Figure C.2 shows a SIP VoIP Adapter configuration parameters running in a computer with IPv6 address 2001:db8::2 and Asterisk installed in a computer with address IPv6 2001:db8::1 and Asterisk listening port 5060 (see 7.4 for Asterisk configuration information). As well as transmitting the audio data received from the SIP call over a UDP socket to remote IPv6 address 2001:db8::4 and remote port 9000. And receiving the audio data to be transmitted to the SIP call from a socket UDP with listening port 9000. The audio codec is the audio format used in the UDP data transmission.

C.6 Registering to Asterisk and Making a Call

For registering the application to Asterisk PBX to be able to make a call to another SIP end client, click button ”register”. A new window is opened asking the user for the user



SIP PROXY:

URI:	client1@[2001:db8::2]
Proxy Address:	[2001:db8::1]
Proxy Port:	5060

AUDIO MEDIA DEVICE:

[ ] Java Audio [✓ UDP Adapter]

Host Port:	9000
Remote Address:	2001:db8::4
Remote Port:	9000
Audio Format:	GSM

Figure C.2: SIP VoIP Adapter configuration example in IPv6

name and password. See Figure C.3. Once the parameters has been filled, and the button ”Ok” has been pushed, if Asterisk accepts the registration, SIP VoIP Adapter will change the ”Register Status” from ”Trying...” to ”(Online)”.

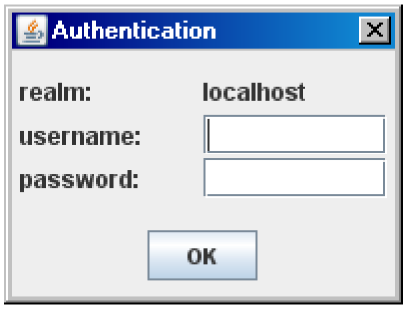


Figure C.3: Authentication window

For calling another SIP end client, type in ”Call Number” the SIP address of the user to call as follows:

[extension]@[outboud proxy IP address]

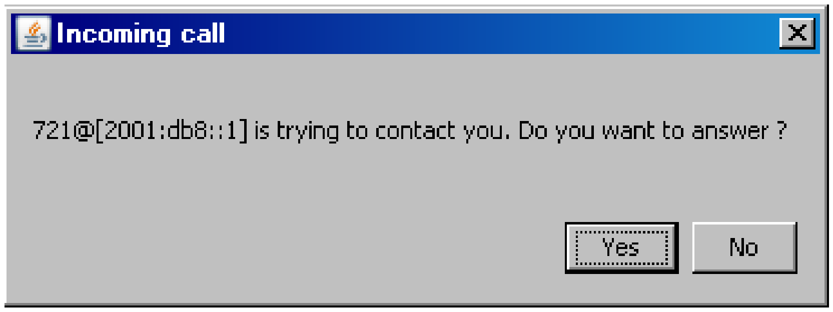
Then click on Open Call Manager and a new window will be opened. Click button ”Make Call” to call the user already specified in main window.In addition, the call window shows some information about the current call, in the text area located on the bottom of the window.

C.7 Receiving a Call

If SIP VoIP Adapter is already configured and registered to Asterisk PBX, and another SIP end client is calling us, a new information window is opened asking the user if wants or not answer the call. If the chosen option is ÓYesÓ, the SIP VoIP Adapter starts the RTP communication and so, the UDP as well. Figure C.4 shows the incoming message window.

C.8 Developing new features

JMF provides the JMFRegistry which is a stand alone Java application that a user can use to register new PlugIns for adapting custom codecs for the audio codifications. So, that it can be used with the JMF 2.1.1 intsallation.


Incoming call

721@[2001:db8::1] is trying to contact you. Do you want to answer ?

Figure C.4: Incoming call message

To provide the SIP VoIP Adapter with a new End Sound Device instead the already implemented, which uses a UDP socket communication. Next classes should be changed:

- TransferHandler.java for transmitting over a custom protocol such as TCP socket, or over a Computer Serial Port, etc.
- UDPDataSource.java and UDPInputStream for receive media data from a custom protocol such as Sockets TCP or from a Computer Serial Port, etc.

For adding new features to SIP VoIP Adapter as the SIP-based suit of standards for instant messaging and presence information, called SIMPLE. The user might follow the application scheme for the handling requests and responses, the following classes should be rewritten:

- SipManager.java for adding SIMPLE user actions.
- MessageListener.java to catch the events generated when some SIMPLE message is received.
- PhoneController.java for the controlling new SIMPLE status.
- Some other classes will need to be changed depending on the SIMPLE implementation.

Bibliography

[1] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler.
”SIP: Session Initiation Protocol”, [RFC 3261] , IETF
http://tools.ietf.org/html/rfc3261

[2] M. Handley, S. Casner, R. Frederick, V. Jacobson.
”SDP: Session Description Protocol”, [RFC 4566] , IETF
http://tools.ietf.org/html/rfc4566

[3] H. Schulzrinne, V. Jacobson, C. Perkins.
”RTP: A Transport Protocol for Real-Time Applications”, [RFC 3550] , IETF
http://tools.ietf.org/html/rfc3550

[4] A. Johnston, S. Donovan, R. Sparks, C. Cunningham, K. Summers.
”Session Initiation Protocol (SIP) Basic Call Flow Examples”, [RFC 3665], IETF
http://tools.ietf.org/html/rfc3665

[5] J. Franks, P. Hallam-Baker, S. Lawrence, P. Leach, A. Luoten, L. Stewart
”Session Initiation Protocol (SIP) Basic Call Flow Examples”, [RFC 2617], IETF
http://tools.ietf.org/html/rfc2617

[6] N. Kushalnagar, G. Montenegro, C. Schumacher.)
”6LoWPAN: Overview, Assumptions, Problem Statement and Goals”, Internet-Draft, IETF
http://tools.ietf.org/html/draft-ietf-6lowpan-problem-08

[7] AVR RZ RAVEN 2.4 GHz Evaluation and Starter Kit
http://www.atmel.com/dyn/Products/tools_card.asp?tool_id=4291

[8] JSR 32: JAIN^{TM}SIP API Specification
http://jcp.org/en/jsr/detail?id=32

[9] Java Media Framework (JMF)
http://java.sun.com/products/java-media/jmf/

[10] JMF API specifications
http://java.sun.com/javase/technologies/desktop/media/jmf/2.1.1/apidocs/

[11] JMF 2.1.1 Solutions: Code Samples and Apps
`http://java.sun.com/javase/technologies/desktop/media/jmf/2.1.1/solutions/index.html`

[12] JMFRegistry User's Guide: Documentation
`http://java.sun.com/javase/technologies/desktop/media/jmf/2.1.1/jmfregistry/jmfregistry.html`

[13] Java Sun
`http://java.sun.com/`

[14] SIP communicator official web site
`https://sip-communicator.dev.java.net/`

[15] NetBeans IDE
`http://www.netbeans.org/`

[16] WireShark
`http://www.wireshark.org/`

[17] Fedora Linux
`http://fedoraproject.org/`