Peers documentation

Yohann Martineau

Peers documentation

Yohann Martineau

Abstract

Peers is a java SIP softphone. Peers documentation is divided into several chapters.

The first chapter gives a reasonable background on voice over IP, especially on SIP. This chapter is accurate for beginners, if you feel comfortable with codecs, rtp, sip and sdp, you can skip this chapter. This chapter applies to any sip client.

The second chapter gives an overview of peers project and its components.

The third chapter shows how peers library can be used in a simple java project. This chapter will definitely be useful for developers coming to peers and wondering how to use the library in their own project.

The fourth chapter gives more details about build system (maven) and modules packaging. This chapter is useful for integrators. You will also probably need to read this chapter if you want to use peers library or any of its demo projects.

Finally, the fifth chapter provides a detailed analysis of each peers module. This part is for experienced sip and java developers.

Table of Contents

List of Figures

List of Examples

Chapter 1. Technical background

Let's start with simple things. First, it may seem obvious, but we have to answer the question: what is Peers? Peers is a lightweight java SIP softphone, i.e. a software that enables its users to place calls on internet.

1.1. Why standards?

When you place calls over internet, your computer is doing many things: capture microphone, play remote contact's voice, send and receive voice over the network, etc. When we look at existing solutions, we see that there are many ways to do this. I will not list all applications that can be used to place calls over internet, but let's see main solutions: skype, msn, gtalk, yahoo. Those solutions are all proprietary solutions, i.e. big companies use their own way to communicate between their client applications (that's what we call a protocol). The benefit of this solution is that they control evolutions of their protocol.

Concerning microsoft, google or yahoo, we must not forget that those companies take the opportunity of their visibility and their marketing power to distribute their applications. They are not necessarily telecommunications experts. This does not mean that they are providing poor quality software, but what would you think if your car manufacturer was to sell you an oven or washing machines? You would probably (at least) hesitate.

The problem of proprietary protocol is that a yahoo client cannot make calls to msn or skype to gtalk, etc. Not directly. There are gateways to place calls from one network to another, but they are error prone and imply complicated translation mechanism. They do not necessarily give exactly the same features, etc. This is the reason why standards have been created. With a common specification, developers can write various applications that communicate between each other. Internet is based on standards: HTTP, HTML, etc. Those standards are the reason of internet success. Everyone can reach everyone, because everyone is using the same language. For web pages, HTTP and HTML are the standard. For media sessions control (voice, video, games, etc.), SIP (Session Initiation Protocol) is the standard. For media sending and reception, RTP (Realtime Transport Protocol) is the standard. And for media sessions description, SDP (Session Description Protocol) is the standard. Those standards (apart HTML) are specified by an organization called IETF [http://www.ietf.org/] (Internet Engineering Task Force). This organization writes its specifications - standards - as plain text files called RFC (Request For Comments). They have a number and a title, but engineers often refer to their number... Thus SIP is specified in RFC3261 [http://tools.ietf.org/html/rfc3261].

Let's take a deep breath and dive in technical details...

SIP is responsible for media sessions establishement, update and teardown. If someone wants to talk with a friend, he or she will tell the software : "I want to invite Bob for an audio session". Let's call Alice the person who is calling Bob. When the conversation is terminated, Alice or Bob will tell their client application : "I'm done with he or she, I want to terminate the session". That's it. This is SIP. SIP means Session Initiation Protocol, but SIP is not only responsible for sessions initiation, but also updates and ending. Its name is not perfect, but let's deal with it.

Neither Alice nor Bob wants to make more complicated things for a conversation. Thus, we can see SIP as the highest level protocol for internet calls, the protocol that "interacts" with users.

Let's make a pause, that was the first level of our technical diving.

Now that we see the aim of SIP, let's understand next steps.

1.2. Media management

Let's start with media handling.

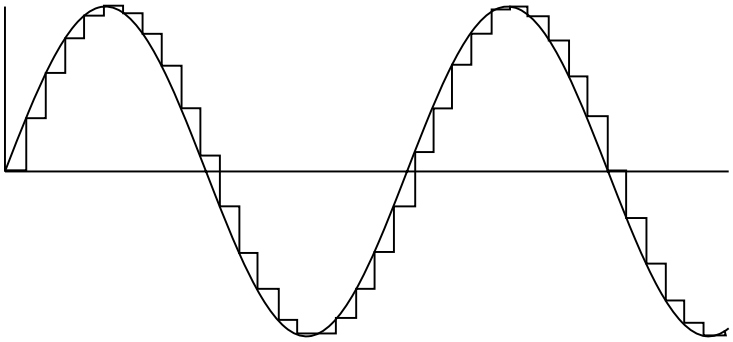
1.2.1. First, voice capture.

Just a glimpse of theory.

1.2.1.1. Sample frequency

Voice is captured using a microphone. Voice stream is an analog signal (a wave). To convert this stream to a computer-usable stream, it has to be converted to a digital signal (stairs). This digital signal is then converted to bits (0, 1) that computer understands. How is it done? A sample is taken in this analog wave at a regular interval and we will consider that this sample is valid up to the next sample. This is not really the case, this is an approximation. And if we take enough samples, we can draw a curve that is very similar to the original wave. The interval at which we take samples is called the sample frequency of the sound.

Figure 1.1. Analog to digital conversion



1.2.1.2. Sample size

Another parameter will play an important role in sampling: sample size. To convert our sample to a computer-usable data, we have to define bounds for the integer that will be considered as the sample value. This is the sample size. The more bits we use to store one sample, the best the sound quality.

1.2.1.3. Number of channels

As we are diving into digital sound analysis, we should consider the number of sources of sound that can be mixed up and heard by human ears. This is the number of channels. For telephony applications, the number of channels is generally one: one microphone makes voice capture, and delivers one source of sound. Even if voice applications generally consider only one source of sound, it is important to understand this notion as many tools use it to manage sound.

1.2.1.4. Endianness

When we define a data format for integers on a computer, we have to define entirely the way bits are converted to integer and vice versa. Unfortunately, everybody did not use the same conventions to store integers. When we convert a decimal number to binary, we generally write binary numbers as hexadecimal numbers:

```
1000 (decimal) = 1111101000 (binary) = 3E8 (hexadecimal)
```

Binary data is generally sliced in groups of 8 bits, i.e. one byte, eight times the value one or zero. To store one byte on a computer, everybody agrees: let's take the first bit (highest power of two within this byte) and put it first, then second, etc. But when our number cannot fit into eight bits, how do we do? We split our number in groups of eight bits:

```
1000 (decimal) = 03 E8 (hexadecimal)
```

Here, I added a 0 in front of 3 to fit 3 on eight bits, that's what we call zero-padding. Some people decided that 03 (the most significant byte) would come first, followed by E8, the way we write. And some people decided that they would store E8 first (the least significant byte), followed by 03. The first solution is called Big-Endian, and the second one is Little-Endian.

1.2.1.5. Signed/unsigned

The last parameter that will play an important role in audio data format is integer boundaries. Actually, some people decided they needed an integer value that would vary between zero and a positive value. And some people decided they needed a value that would be positive or negative, with a sign in front of the previous absolute number. The problem is that we still have a multiple of eight bits to store our integer. The common solution is to drop the most significant bit (not byte) and use its empty space to store the sign of our integer.

1.2.2. Then, audio data encoding.

The samples we have for the moment are sometimes called raw data of linear PCM (Pulse Code Modulation). To find the bandwidth that's necessary to transfer audio data, the following formula can be applied:

```
bandwidth = frequency * sample size * number of channels
```

(signed/unsigned or endianness do not infer on bandwidth as they keep the same storage space) If we consider that 16 bits samples are taken at 8 kHz with only one channel, 128000 bits of data must be sent during one second to keep our voice quality. Even if internet providers bandwidths are growing, such an upload bandwidth is huge and generally not available on internet.

To face this issue, smart optimisations are applied on audio data so that they can fit on reasonable bandwidth, available on most networks on internet. Those optimisations are called codecs (coder decoder). They rely on voice and ear physical properties to avoid naive raw data transfert. I will not give too much details on codecs in this documentation. There are many audio codecs, but the first one that is generally implemented by SIP clients is G711 mu-law. Please refer to wikipedia article [http://en.wikipedia.org/wiki/G.711] for more information. Once optimisations are done, data is compressed and needs less space to transport the same voice stream.

1.2.3. Packetization

After data encoding, audio stream is packetized. It means that slices of audio stream are extracted from the encoder output. But on a SIP network, media data cannot be sent raw, as is. It must be sent using RTP (Realtime Transport Protocol). RTP strives to solve realtime media transport issues that can occur on IP networks. Thus, it provides a header to include a timestamp. This timestamp gives a clue about when the packet must be played by the receiver. It also includes a sequence number that enables packets re-ordering. The fact is that, on SIP networks, RTP is often transported over UDP, because we can afford losing a few media packets and small disorder in packets reception. UDP is appreciated for its speed over its reliability, which is a big advantage in realtime environment. RTP is a binary protocol which transports binary data.

On the receiver side, RTP packets are parsed. RTP headers are dropped and media data is extracted.

1.2.4. Audio playback

Once media data is extracted, still compressed, it is transmitted to decoder that will generate raw uncompressed data samples. Those playable samples are then transmitted to a player which will send instructions to the sound card to play voice samples correctly.

1.3. Session control

Let's now understand how SIP makes people reachable on a network and talking the same codec.

1.3.1. Registration

Let's come back to Alice and Bob. Alice and Bob both use an IP network to reach each other. When Alice wants to call Bob, she knows his SIP uri (sip:bob@biloxi.com), but her computer does not know where Bob is, on which computer, on which IP address. Thus, Alice and Bob's client application registers when their computer starts or when they want to be reachable by SIP network to tell a central server: hello, I'm here, my IP address is 1.2.3.4 and the port I'm using is 5060. Smart readers that you are know that there are NATs (Network Address Translation) on internet but for the moment, I will consider that a public IP address is used for Alice and Bob's client application.

As Bob's computer is registered on a central server (called registrar), Alice's client application (User-Agent) sends its request to Bob's registrar, who will then forward the request to Bob's IP address and port. SIP considers that there may be several domains/realm with a registrar for each domain (several providers). Another important element on SIP networks is proxy. A proxy is an element that receives requests from User-Agents (or other SIP nodes), may modify those requests, ask authentication, computes routes, and then, forwards those requests to other proxies, registrars or User-Agents. It's a sort of relay. It may filter malformed requests, etc.

1.3.2. Codec negotiation

Now that our User-Agents (client software) are talking the same control protocol: SIP, they must establish a media session so that Alice can hear Bob and Bob can hear Alice. SIP is a flexible protocol. Thus, it states that User-Agents can support several codecs to send or receive media packets. It specifies that G711 must be supported at least. This is the reason why most SIP User-Agents implement G711 first, and then add more complicated codecs.

As several codecs can be supported, User-Agents use a common language to describe their codecs in their SIP messages. This is SDP (Session Description Protocol). When a User-Agent sends a request to create a media session, it includes a description of its supported codecs. And when a User-Agent answers a request that is willing to create a media session, it also includes the set of supported codecs, even if there is only one. This is codec negotiation. Each User-Agent takes the remote party's codec list and takes the first one in this list that matches a codec in its own codecs list. Thus both User-Agents use the same way to encode and decode media data for Alice and Bob's voice. Generally, User-Agents put their "worst" codec in last position in their list so that best quality codecs are preferred.

Actually, codec negotiation relies on offer/answer model for SDP. This model is specified in RFC3264 [http://tools.ietf.org/html/rfc3264]. Thus, the request that wants to create a new media session may be empty, without any offer. In this case, the User-Agent is telling: make me an offer, and I'll give you my supported codecs appropriately. In this case, the SDP offer is in SIP response (200 OK), and the SDP answer is in ACK.

Chapter 2. Overview

The following paragraphs apply to peers since version 0.5.

Peers is distributed several ways: either as a binary "desktop" zip file which can be downloaded on sourceforge in peers project page [http://sourceforge.net/projects/peers], or as a git project hosted on github [https://github.com/ymartineau/peers] for souce code. The binary zip file can be extracted on windows, linux or mac and the .bat or .sh startup script can be used to start peers. To download peers source code, a git client must be available on your computer, then you have to run:

```
git clone https://github.com/ymartineau/peers.git
cd peers
git checkout 0.5
```

Checkout command will put you in 0.5 tag. Thus you use the validated version and not latest source versions from master branch.

If you take a look at peers source code, you will see that peers is actually a maven [http://maven.apache.org/] project containing several modules:

- peers-demo
- peers-doc
- peers-gui
- peers-javaxsound
- peers-js
- peers-jws
- peers-lib

Maven is a java build tool which is pretty useful to manage dependencies amongst modules or on external libraries. You must download it, extract it and setup your environment variables if you want to build peers from source code. Maven uses files called pom.xml to define projects and modules.

Peers-demo is a command line demo project to show peers library usage with a simple example. See next chapter.

Peers-doc is a module which contains this documentation as docbook. It generates an html file and a pdf file.

Only three modules are used to generate peers desktop binary zip file: peers-gui, peers-lib and peers-javaxsound. Peers-gui is the graphical user interface based on swing [http://docs.oracle.com/javase/tutorial/uiswing/]. Peers-javaxsound contains the classes using javax.sound [http://docs.oracle.com/javase/tutorial/sound/] java extension package for microphone capture and sound playback. Peers-lib implements everything that is not related to graphical user interface and sound capture/playback (network stacks, audio encoding, etc.). It's actually the most important module in peers: it contains SIP, SDP and RTP stacks.

Actually, peers binary zip file is built by peers-gui. Peers-gui has a dependency on peers-lib and peers-javaxsound. Thus, peers-gui retrieves a binary file from peers-lib and peers-javaxsound, adds its own directories and files and packages the whole as a zip file.

In addition to peers-demo example, peers-jws and peers-js give two demo applications to demonstrate how peers can be launched from browser or used directly in browser. You should not use those modules in a deployed application, they give examples to create new applications based on peers-lib. You can consider those demo applications as proofs of concept.

Peers-jws is a Java Web Start [http://docs.oracle.com/javase/tutorial/deployment/webstart/] application. A Java Web Start application is just a java application that can be launched from a web browser with a simple click. Actually, you generally need to grant system resources security access rights (microphone, write to the disk, etc.) but it's running outside your browser.

Peers-js is another demo application. Peers-js is a java applet running in browser. When you run this demo application, the java applet is not visible in browser, user interface is developed using html and javascript. In this case, peers interface is javascript, hence its name: peers-js.

Take care though with this module: since java 1.7.0-21, applets not signed by a java certificate provided by a certification authority cannot be run. There is a workaround for developers. In java config panel security panel, a trusted source website can be defined. In addition to this restriction added in java 1.7.0-21, firefox is now blocking java applets by default since firefox 26. You can force java applets execution in firefox but they are blocked by default.

In addition to maven modules, peers root directory contains several utility directories:

- conf
- logs
- media

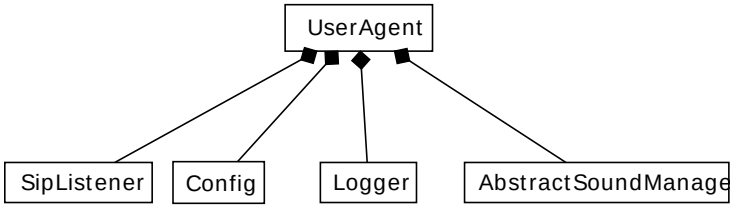
Conf directory contains peers xml configuration file and its grammar. This is the file where you provide your user account info: username (userpart), password and server (domain). Logs contains log files once you started peers in root directory and media directory contains binary files with raw audio data generated when media debug option is turned on.

Chapter 3. Peers library usage

3.1. Overview

Peers provides one class to use sip and media stacks (UserAgent) and one interface to receive notifications from sip stack (SipListener). The UserAgent constructor takes a few parameters: SipListener, Config, Logger and AbstractSoundManager. SipListener parameter in UserAgent constructor must be a class implementing the corresponding interface. It can be the class that creates UserAgent so that there is only one class to communicate with peers API. Config is an interface which must be implemented by your application. This simple interface will provide configuration parameters to sip stack such as username, password and domain name. If you want to use an xml configuration file to store those parameters, an XmlConfig class is available in peers-lib. Logger is also a simple interface to log messages. You can use null in User-Agent constructor so that a default logger is used. Default logger will output messages on standard output. The last parameter is AbstractSoundManager. Here you can provide your own implementation of AbstractSoundManager or you can use JavaxSoundManager from peers-javaxsound module. This AbstractSoundManager is a class that defines the way to access low level API to write sound to speakers and read sound from microphone.

Figure 3.1. Peers API



3.2. Example

The simplest way to develop an application based on peers is using maven.

As peers is a maven project, you just have to add the right dependencies to your application and you can easily integrate a sip phone in your own application.

If you want or need to use peers from a non-maven project, you can just retrieve the appropriate jar files in your maven repository once you have built peers (using "mvn clean install" in peers root directory) or you can download peers standard desktop version and retrieve peers.jar once you extracted the archive. This peers.jar contains peers-lib, peers-javaxsound and peers-gui classes.

The following example application will be a command line interface (cli) phone based on peers. Let's call this application peers-demo. This application will run from command line, notify user about events (remote hangup, etc.) and take commands from console. Those commands will be as simple as possible: call <number>, hangup. This application will support only one simultaneous call. SIP registration parameters will be provided using Config java interface. This demo application will rely on peers-lib to manage network stacks and on peers-javaxsound to manage microphone and speakers.

The final application code is available in peers-demo module in peers source code [https://github.com/ymartineau/peers/tree/master/peers-demo].

Here is an example to create an empty maven application:

```
mvn archetype:generate
      -DarchetypeGroupId=org.apache.maven.archetypes
      -DgroupId=net.sourceforge.peers
      -DartifactId=peers-demo
```

It creates a nice empty maven project. Let's add peers-lib dependency to this project. In pom.xml, add the following dependency:

```
    <dependency>
        <groupId>net.sourceforge.peers</groupId>
        <artifactId>peers-lib</artifactId>
        <version>0.5</version>
        <scope>compile</scope>
      </dependency>
      <dependency>
        <groupId>net.sourceforge.peers</groupId>
        <artifactId>peers-javaxsound</artifactId>
        <version>0.5</version>
        <scope>compile</scope>
      </dependency>
```

Let's check that this project still builds from newly created project directory:

```
mvn clean install
...
[INFO] BUILD SUCCESS
...
```

Great, now let's start developing our real application. My IDE (Integrated Development Environment) is eclipse [https://www.eclipse.org/]. To generate eclipse project files easily, I use the following command line either in peers root directory or in any maven module (let's do it in peers-demo new project):

```
mvn eclipse:eclipse
```

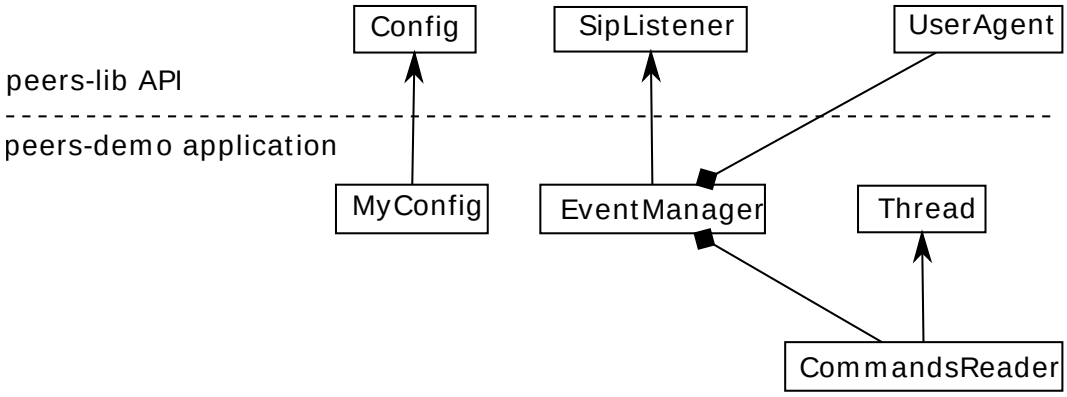
Thanks to this command, you can import existing project in eclipse: File > Import > Existing Projects into Workspace > Select the root directory from which you want to scan projects and browse to the place where you created your new maven project > Finish and that's it. You can now open your project in eclipse. Maven already created a main class for you called App. You can delete this class and the corresponding unit test in src/test/java.

When you modify project library dependencies in pom.xml, don't forget to run mvn eclipse:eclipse again and refresh project in eclipse. It updates dependencies in eclipse project.

We create a new package to host classes for this application: net.sourceforge.peers.demo. We will create a Thread to read commands from console (CommandsReader) and a main object will receive events from console and sip stack. Let's call this main object EventManager. It will handle all events coming from sip stack and from CommandsReader and will make decisions appropriately (display message on console, call peers API, etc.). Actually, in a real application this central class is a finite state machine.

This application will just behave as a client, it will not support incoming calls. This is just to provide a very simple example.

Figure 3.2. peers-demo application UML class diagram



Let's start with the class that will read commands from console:

Example 3.1. CommandsReader

```
package net.sourceforge.peers.demo;
[...]
public class CommandsReader extends Thread {

    [...]

    @Override
    public void run() {
        InputStreamReader inputStreamReader = new InputStreamReader(System.in);
        BufferedReader bufferedReader = new BufferedReader(inputStreamReader);
        setRunning(true);
        while (isRunning()) {
            String command;
            try {
                command = bufferedReader.readLine();
            } catch (IOException e) {
                e.printStackTrace();
                break;
            }
            command = command.trim();
            if (command.startsWith(CALL)) {
                String callee = command.substring(
                        command.lastIndexOf(' ') + 1);
                eventManager.call(callee);
            } else if (command.startsWith(HANGUP)) {
                eventManager.hangup();
            } else {
                System.out.println("unknown command " + command);
            }
        }
    }
```

This class has a reference to the central class EventManager, a getter and a setter on an isRunning boolean. This variable will be used to stop the thread. As you can see here, string commands are just read from System.in using a BufferedReader to get one line at a time. Useless spaces are trimmed and one argument is read for call command.

Here is the corresponding EventManager class, in its first version.

Example 3.2. EventManager interfaces

```
public class EventManager implements SipListener {

    // commands methods
    public void call(String callee) { }
    public void hangup() { }


    // SipListener methods

    @Override
    public void registering(SipRequest sipRequest) { }

    @Override
    public void registerSuccessful(SipResponse sipResponse) { }

    @Override
    public void registerFailed(SipResponse sipResponse) { }

    @Override
    public void incomingCall(SipRequest sipRequest, SipResponse provResponse) { }

    @Override
    public void remoteHangup(SipRequest sipRequest) { }

    @Override
    public void ringing(SipResponse sipResponse) { }

    @Override
    public void calleePickup(SipResponse sipResponse) { }

    @Override
    public void error(SipResponse sipResponse) { }

}
```

This EventManager class implements methods called from CommandsReader and the SipListener interface from peers API to receive notifications from peers-lib SIP stack (to notify our application that callee picked up the call for example).

Now, let's add our utility classes to initialize UserAgent class in our EventManager constructor:

Example 3.3. Create a UserAgent in EventManager

```
private UserAgent userAgent;
private CommandsReader commandsReader;

public EventManager() throws SocketException {
    Config config = new CustomConfig();
    Logger logger = new FileLogger(null);
    JavaxSoundManager javaxSoundManager = new JavaxSoundManager(false,
        logger, null);
    userAgent = new UserAgent(this, config, logger, javaxSoundManager);
    new Thread() {
        public void run() {
            try {
                userAgent.register();
            } catch (SipUriSyntaxException e) {
                e.printStackTrace();
            }
        }
    }.start();
    commandsReader = new CommandsReader(this);
    commandsReader.start();
}
```

Here, we use a custom implementation of Config interface (CustomConfig) to provide hardcoded account parameters. The FileLogger(null) instance will actually log messages to console on standard output. Its name is pretty dumb, but don't mind. If you create a logs dir in your project root directory (peers-demo), it would log messages to two text files: logs/peers.log and logs/transport.log. transport.log would give full traces of sip messages (outgoing and incoming) and peers.log all traces about peers internal sip stack (finite state machine transitions, important steps, etc.). JavaxSoundManager provides the real access to microphone and speakers. This class relies upon oracle java javax.sound extension. Once those utility objects are available, peers UserAgent class can be instantiated.

userAgent.register() will start event registration, we call this method in a new thread to register user account in background.

Here is an example CustomConfig class:

Example 3.4. CustomConfig

```
package net.sourceforge.peers.demo;

[...]
public class CustomConfig implements Config {

    private InetAddress publicIpAddress;

    @Override
    public InetAddress getLocalInetAddress() {
        InetAddress inetAddress;
        try {
            // if you have only one active network interface, getLocalHost()
            // should be enough
            //inetAddress = InetAddress.getLocalHost();
            // if you have several network interfaces like I do,
            // select the right one after running ipconfig or ifconfig
            inetAddress = InetAddress.getByName("192.168.1.11");
        } catch (UnknownHostException e) {
            e.printStackTrace();
            return null;
        }
        return inetAddress;
    }

    @Override
    public InetAddress getPublicInetAddress() { return publicIpAddress; }
    @Override public String getUserPart() { return "alice"; }
    @Override public String getDomain() { return "atlanta.com"; }
    @Override public String getPassword() { return "secret1234"; }
    @Override  // use microphone and speakers to capture and playback sound
    public MediaMode getMediaMode() { return MediaMode.captureAndPlayback; }

    @Override
    public void setPublicInetAddress(InetAddress inetAddress) {
        publicIpAddress = inetAddress;
    }
```

Example 3.5. CustomConfig (cont.)

```
    @Override public SipURI getOutboundProxy() { return null; }
    @Override public int getSipPort() { return 0; } // use default sip port 5060
    @Override public boolean isMediaDebug() { return false; }
    @Override public String getMediaFile() { return null; }
    @Override public int getRtpPort() { return 0; } // use random rtp port

    // in this simple example, we don't need those modifiers, but they are
    // required by the interface
    @Override public void setLocalInetAddress(InetAddress inetAddress) { }
    @Override public void setUserPart(String userPart) { }
    @Override public void setDomain(String domain) { }
    @Override public void setPassword(String password) { }
    @Override public void setOutboundProxy(SipURI outboundProxy) { }
    @Override public void setSipPort(int sipPort) { }
    @Override public void setMediaMode(MediaMode mediaMode) { }
    @Override public void setMediaDebug(boolean mediaDebug) { }
    @Override public void setMediaFile(String mediaFile) { }
    @Override public void setRtpPort(int rtpPort) { }
    @Override public void save() { }

}
```

Now, we need to invoke userAgent api when a command is invoked from command line. In EventManager, we add a sipRequest to keep a reference to the current call so that we can hangup when a call is established.

Example 3.6. Invoke UserAgent peers API

```
public class EventManager implements SipListener {
    [...]
    private SipRequest sipRequest;
    [...]
    // commands methods
    public void call(final String callee) {
        new Thread() {
            @Override
            public void run() {
                try {
                    sipRequest = userAgent.invite(callee, null);
                } catch (SipUriSyntaxException e) {
                    e.printStackTrace();
                }
            }
        }.start();
    }

    public void hangup() {
        new Thread() {
            @Override
            public void run() {
                userAgent.terminate(sipRequest);
            }
        }.start();
    }
    [...]
```

Here we use a null second parameter to invite so that callId is generated automatically by sip stack. A call id is just a unique call identifier used in sip message. We also call userAgent methods in new threads so that console does not stick on commands input.

Our demo project is almost done, we still need a main to instantiate EventManager:

Example 3.7. Demo project main method

```
    public static void main(String[] args) {
        try {
            new EventManager();
        } catch (SocketException e) {
            e.printStackTrace();
        }
    }
```

Now run this demo project: in eclipse right-click EventManager and Run As > Java Application. You should see something like this (credentials in traces don't match github sources code because those traces corresponds to a real account):

```
logging to stdout
2014-03-22 00:20:07,893 INFO  [main] starting user agent
    [myAddress: 192.168.1.11, sipPort: 0, userpart: ymartineau,
    domain: ippi.fr]
2014-03-22 00:20:07,977 DEBUG [main] new DatagramSocket(0, /192.168.1.11
2014-03-22 00:20:07,985 INFO  [main] added datagram socket
    192.168.1.11:60588/UDP
2014-03-22 00:20:07,986 INFO  [main] added 192.168.1.11:60588/UDP:
    net.sourceforge.peers.sip.transport.UdpMessageReceiver@7436ef21 to message
    receivers
```

Then, you should see the sip REGISTER message sent and received:

```
2014-03-22 00:20:08,103 SENT to 213.215.45.230/5060 [Thread-2]

2014-03-22 00:20:08,234 SENT to 213.215.45.230/5060 [TransportManager 0]

REGISTER sip:ippi.fr SIP/2.0
Via: SIP/2.0/UDP 192.168.1.11:60588;rport;branch=z9hG4bKu62lFMeRS
Max-Forwards: 70
To: <sip:ymartineau@ippi.fr>
From: <sip:ymartineau@ippi.fr>;tag=tPpKyeqA
Call-ID: F4vlP0Y4-1395444008026@192.168.1.11
CSeq: 2 REGISTER
Contact: <sip:ymartineau@192.168.1.11:60588;transport=UDP>
Authorization: Digest username="ymartineau", realm="ippi.fr",
    nonce="532cca45176760c845efda663182a5be2609f85c", uri="sip:ippi.fr",
    response="d44b8bd2a59f38b003628dc276c69ad7"


2014-03-22 00:20:08,270 RECEIVED from 213.215.45.230/5060 [TransportManager 0]

SIP/2.0 200 OK
Via: SIP/2.0/UDP 192.168.1.11:60588;received=192.168.1.11;rport=60588;
    branch=z9hG4bKu62lFMeRS
To: <sip:ymartineau@ippi.fr>;tag=a910c8153188470b2841623c513a131f.db9b
From: <sip:ymartineau@ippi.fr>;tag=tPpKyeqA
Call-ID: F4vlP0Y4-1395444008026@192.168.1.11
CSeq: 2 REGISTER
Contact: <sip:ymartineau@192.168.1.11:60588;transport=UDP>;
    expires=3600;received="sip:90.32.185.72:60588",
    <sip:ymartineau@192.168.1.11:60587;transport=UDP>;
    expires=3559;received="sip:90.32.185.72:60587",
    <sip:ymartineau@192.168.1.11:61005;transport=UDP>;
    expires=3388;received="sip:90.32.185.72:61005"
Server: OpenSIPS (1.8.2-tls (i386/linux))
Content-Length: 0
```

You could also run this application from the command line in your OS console using the right dependencies in your java classpath.

It may be tricky to write commands on console because of the numerous messages displayed. If you have difficulties, write the command in your preferred text editor first and then copy paste it in the console (eclipse console or OS console) and hit return. Here is an example:

```
call sip:*1234@ippi.fr
```

You should see something like:

```
2014-03-22 00:20:34,965 DEBUG [Thread-6]
    SM z9hG4bKHnwoNi3EZ|INVITE [InviteClientTransactionStateInit ->
        InviteClientTransactionStateCalling] setState
2014-03-22 00:20:34,965 DEBUG [Thread-6] UdpMessageSender.sendMessage
2014-03-22 00:20:34,965 DEBUG [Thread-6] UdpMessageSender.sendBytes
2014-03-22 00:20:34,965 DEBUG [Thread-6] UdpMessageSender.sendBytes 577
    ippi.fr/213.215.45.230:5060
2014-03-22 00:20:34,965 DEBUG [Thread-6] /192.168.1.11
2014-03-22 00:20:34,965 DEBUG [Thread-6] UdpMessageSender.sendBytes packet sent
2014-03-22 00:20:34,965 SENT to 213.215.45.230/5060 [Thread-6]

INVITE sip:*1234@ippi.fr SIP/2.0
Via: SIP/2.0/UDP 192.168.1.11:60588;rport;branch=z9hG4bKHnwoNi3EZ
Max-Forwards: 70
To: <sip:*1234@ippi.fr>
From: <sip:ymartineau@ippi.fr>;tag=nImz7hym
Call-ID: kTw3F5Wk-1395444034935@192.168.1.11
CSeq: 3 INVITE
Content-Length: 212
Content-Type: application/sdp
Contact: <sip:ymartineau@192.168.1.11:60588;transport=UDP>

v=0
o=user1 432074470 1969904641 IN IP4 192.168.1.11
s=-
c=IN IP4 192.168.1.11
t=0 0
m=audio 60590 RTP/AVP 0 8 101
a=rtpmap:0 PCMU/8000
a=rtpmap:8 PCMA/8000
a=rtpmap:101 telephone-event/8000
a=sendrecv

2014-03-22 00:20:34,965 DEBUG [Thread-6] InviteClientTransaction.start
2014-03-22 00:20:34,996 RECEIVED from 213.215.45.230/5060 [TransportManager 0]
```

If the callee pick-ups you should now be able to talk with the callee using your computer microphone and you can use the following command to hangup:

```
hangup
```

If hangup is successful, you should see something like:

```
2014-03-22 00:20:42,286 SENT to 213.215.45.230/5060 [Thread-8]

BYE sip:411@213.215.45.231 SIP/2.0
Via: SIP/2.0/UDP 192.168.1.11:60588;rport;branch=z9hG4bKZechHp2Ua
```

```
To: <sip:*1234@ippi.fr>;tag=as4f8c117e
From: <sip:ymartineau@ippi.fr>;tag=nImz7hym
Call-ID: kTw3F5Wk-1395444034935@192.168.1.11
CSeq: 6 BYE
Route: <sip:213.215.45.230;lr>
Max-Forwards: 70
Proxy-Authorization: Digest username="ymartineau", realm="ippi.fr",
    nonce="532cca60e3cd40a74c9e37655a0addccc6596e41", uri="sip:*1234@ippi.fr",
    response="71f51c20a68e8aa86e3ff00c3a81b71a"


2014-03-22 00:20:42,304 DEBUG [Thread-8] closeLines
2014-03-22 00:20:42,339 RECEIVED from 213.215.45.230/5060 [TransportManager 0]

SIP/2.0 200 OK
Via: SIP/2.0/UDP 192.168.1.11:60588;received=192.168.1.11;rport=60588;
    branch=z9hG4bKZechHp2Ua
From: <sip:ymartineau@ippi.fr>;tag=nImz7hym
To: <sip:*1234@ippi.fr>;tag=as4f8c117e
Call-ID: kTw3F5Wk-1395444034935@192.168.1.11
CSeq: 6 BYE
User-Agent: Asterisk PBX
Allow: INVITE, ACK, CANCEL, OPTIONS, BYE, REFER, SUBSCRIBE, NOTIFY
Contact: <sip:411@213.215.45.231>
Content-Length: 0
```

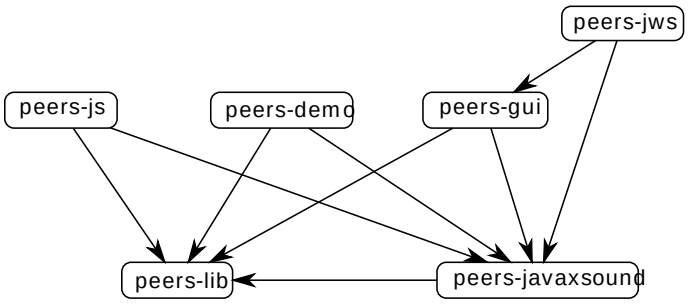
As you can see, we built our first example in half an hour using maven, eclipse and three java classes. You can now build more complex examples using dtmf UserAgent method, or notify user when an incoming call is reaching your application.

Chapter 4. Build system

Peers relies on maven to build all its modules, jar files and zip files.

Here are dependencies amongst modules:

Figure 4.1. Peers modules



My advice is to use "mvn clean install" to build peers project or any of its modules. Generated files for all modules are in their target subdirectory. If you run "mvn clean install" in peers source code root directory, the same command will be run automatically in all modules declared in peers root project pom.xml in <modules>. Actually, it's true for any maven command that you invoke from root directory. If you take a look at peers root pom.xml, you will see that most modules are "attached" to peers parent project.

Peers-lib generates a simple jar file containing sip, rtp and sdp stacks.

Peers-javaxsound generates a simple jar file containing only JavaxSoundManager class.

Peers-gui first generates a jar file containing all its dependencies (peers-jws-with-dependencies.jar). Thus, this file includes classes from peers-lib and peers-javaxsound jars combined in one jar. Then it creates a zip file that can be extracted on standard pc to run peers. In this archive, the previous peers-jws-with-dependencies.jar file is called peers.jar. This archive also includes conf, media and logs directories and two scripts files to start peers on linux or windows easily.

Peers-jws includes a demo web page. Peers-jws generates a zip file including this web page and all dependencies. This zip file can be extracted in web server directory. Then, you just have to browse the corresponding url to test peers-jws.

Similarly, peers-js includes a demo web page and generates a zip file that can be extracted on a web server. This module relies on a jar provided only with sun or oracle java versions (provided in its Java Runtime Environment install directory): ${java.home}/lib/plugin.jar. This jar file contains a netscape.javascript package containing objects used to invoke javascript functions from java classes. To build this module, you must define a JAVA_HOME environment variable that points to your JRE.

Peers-demo generates a jar file containing only its classes, it is not reused by another module.

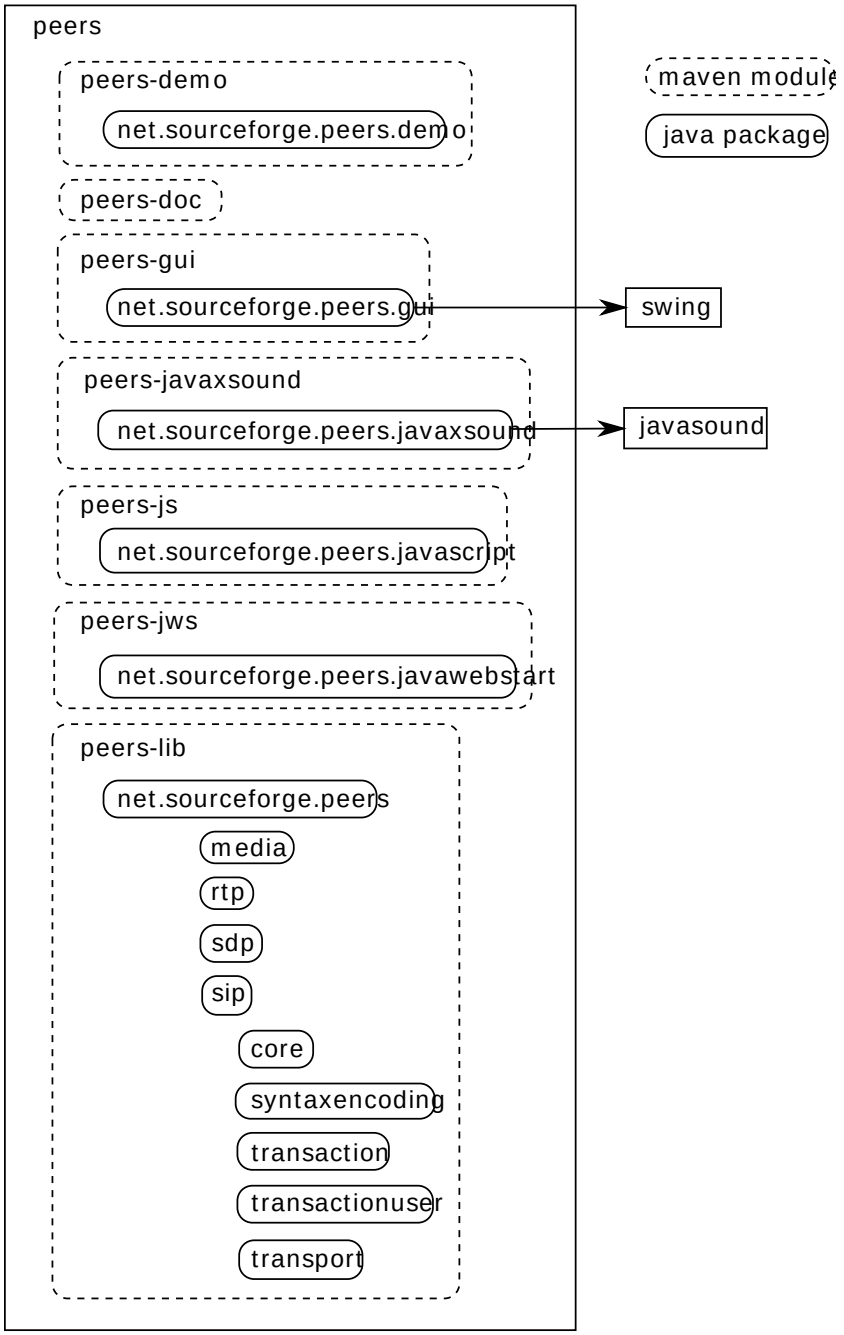
All zip files built by peers modules rely on maven assembly [https://maven.apache.org/plugins/maven-assembly-plugin/] to define the content of the target zip file. This maven plugin [https://maven.apache.org/plugins/] is explicitly invoked in the corresponding modules pom.xml. The archive is defined in an xml file (often called bin.xml) in src/assembly subdirectory.

Chapter 5. Peers source

5.1. Overview

Here is an overview of peers maven modules and java packages:

Figure 5.1. Peers modules and java packages



Peers-demo source code has already been covered in peers usage chapter.

Peers-doc contains this documentation as a docbook [http://docbook.org/tdg/en/html/docbook.html] file: peers.xml. Source svg images are created using Inkscape [http://www.inkscape.org/].

Peers-javaxsound has already been covered, it only opens and closes javax.sound target and source data lines. javax.sound is also known as javasound.

Peers-js contains an implementation of Applet class to define the java applet.

Peers-jws just invokes the standard peers desktop version but from java web start environment (less rights at the beginning). This is the reason why it depends on peers-gui and not peers-lib and peers-javaxsound directly.

Peers-lib and peers-gui are more complex. Peers-gui relies on swing. Swing is the graphical user interface library provided by oracle to build graphical applications.

5.2. Architecture

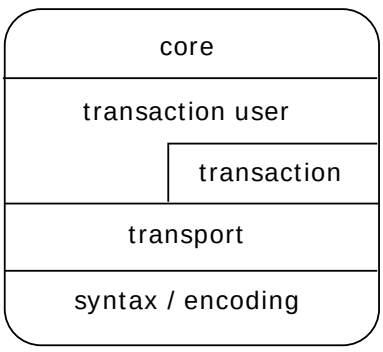
Peers has been developed in java, an object-oriented programming language. Peers relies on standard java specification API. If you are using a standard java platform (openjdk or sun jdk), everything is already included in your environment, you need no other library. Peers has no external dependency. If you don't want to download sources and import them in your preferred IDE, you can browse the source code online, on github [https://github.com/ymartineau/peers].

For your information, nat package in peers-lib is not used for the moment, it was an experiment about Port Restricited Cone NAT traversal. media package is responsible for sound encoding. SDP and SIP do not rely on any external library. Of course, sdp and sip packages contains SDP related sources and SIP stack implementation. rtp package contains peers rtp implementation, which is used by media package. The only complicated (but interesting) package is sip. Let's see what sip is made of:

- net.sourceforge.peers.sip.core
- net.sourceforge.peers.sip.transactionuser
- net.sourceforge.peers.sip.transaction
- net.sourceforge.peers.sip.transport
- net.sourceforge.peers.sip.syntaxencoding

As you probably remarked, it corresponds to RFC3261 [http://tools.ietf.org/html/rfc3261] layers:

Figure 5.2. SIP stack

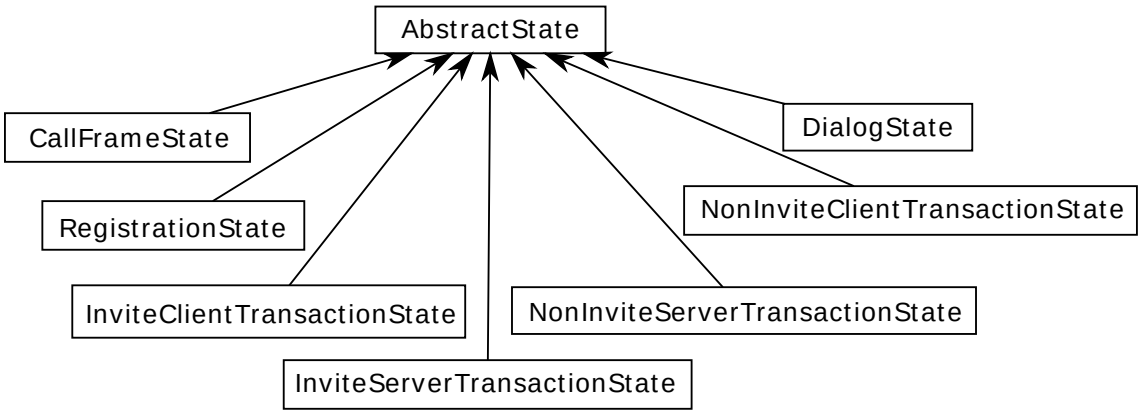


The reader will probably need to keep RFC3261 open to understand thoroughly the following paragraphs. I did not reinvent the wheel. The sip package has been implemented with simplicity and extensibility in mind. This implementation should not be obscure to a java developer that already knows SIP. The following paragraphs will contain UML [http://en.wikipedia.org/wiki/Unified_Modeling_Language] diagrams. But before we explain the meaning of each package, let's see some common techniques that have been used in several packages.

5.3. State machines

SIP defines several state machines, a design pattern [http://sourcemaking.com/design_patterns/state] has been used for state machines. It consists of one class for the object that handles its state, one parent class for all states of this state machine and one class for each state. As there were several state machines, and as it was useful to log state transitions, a general abstract state class has been defined, it just prints the old state, the new state and the transition. Then each parent state class in state machines extends this abstract state class:

Figure 5.3. Abstract state for all state machines



We won't give more details about the role of each state machine now, just keep in mind the way they are managed and implemented, not what they are done for.

5.4. Managers

The second design pattern that has been used in Peers is "Factory". It has been used in several packages. Actually, in Peers, factories are called managers. Those classes are called xxxManager. Managers are more than factories, because they are employed to create object instances, but they are also used to store all references to those objects. For example, TransactionManager implements a method to create a new client transaction: createClientTransaction, and a method to retrieve a client transaction: getClientTransaction. Thus, when an external object needs to access one of the objects created by a manager, it uses its get method. In some cases, one manager can create several types of objects. In the previous example, TransactionManager manages client transactions and server transactions. In such cases, the appropriated get method must be employed. All managers have been implemented the same way. They contain hashtables for the object tables they manage. For those reasons, and as they are used to delete references to those objects, the word manager has been preferred to factory.

5.5. Package users

Interaction between packages is sometimes made using interfaces and javabeans event pattern. Those interfaces defines objects users. Thus when an object outside a package needs to get information from

one object, it implements its corresponding User interface. And then it gets notified about events. Users interfaces are quite similar to Listeners. But they are not called Listeners because they do not necessarily apply to pure beans or POJOs (plain old java objects).

In SIP theory, only one layer relies on two other layers: transaction user on transaction and transport layers. In peers implementation, you can find dependencies on several packages: syntaxencoding for message access, media for sound management, etc. This does not break this model. Most of those accesses are done to retrieve data from other objects, not to perform an action on an event.

5.6. Package details

5.6.1. SIP

Let's start with sip-related packages. The best way to get in touch with SIP is probably using wireshark network analyzer and its sip filter, and trying to place calls. We have already seen that Peers source code is made of several packages that correspond to SIP layers. We will start with the lowest layer (syntax/encoding) with simple message examples. Then we will climb up the layer stack. The next step is transport management, i.e. the way messages travel over the network. Next, we will see how those messages are grouped to make transactions. Then, we will explain how those transactions are grouped to manage dialogs. And last but not least, we will understand how dialogs are managed by core layer. But before we explain how those high-level layers are implemented in java, let's discover SIP by its messages.

5.6.1.1. Message anatomy

SIP uses two types of messages: requests and responses. Requests contain a method (*INVITE* in the following example) that will give request aim and a request-uri (*sip:bob@biloxi.example.com*) for the person/server we want to reach. Responses contain a status code (an integer, *200* in next example) that gives response status: success, failure, etc. Each SIP message is made of several headers and one body. A header has a name, and generally one value but it may contain several values. A SIP header can contain one or several parameters with the following syntax:

```
header_name: header_value;param=param_value
```

Here is an example message quoting RFC3665 [http://tools.ietf.org/html/rfc3665], which gives simple call-flow examples.

```
INVITE sip:bob@biloxi.example.com SIP/2.0
Via: SIP/2.0/TCP client.atlanta.example.com:5060;branch=z9hG4bK74bf9
Max-Forwards: 70
From: Alice <sip:alice@atlanta.example.com>;tag=9fxced76sl
To: Bob <sip:bob@biloxi.example.com>
Call-ID: 3848276298220188511@atlanta.example.com
CSeq: 1 INVITE
Contact: <sip:alice@client.atlanta.example.com;transport=tcp>
Content-Type: application/sdp
Content-Length: 151

v=0
o=alice 2890844526 2890844526 IN IP4 client.atlanta.example.com
s=-
c=IN IP4 192.0.2.101
t=0 0
m=audio 49172 RTP/AVP 0
```

```
a=rtpmap:0 PCMU/8000
```

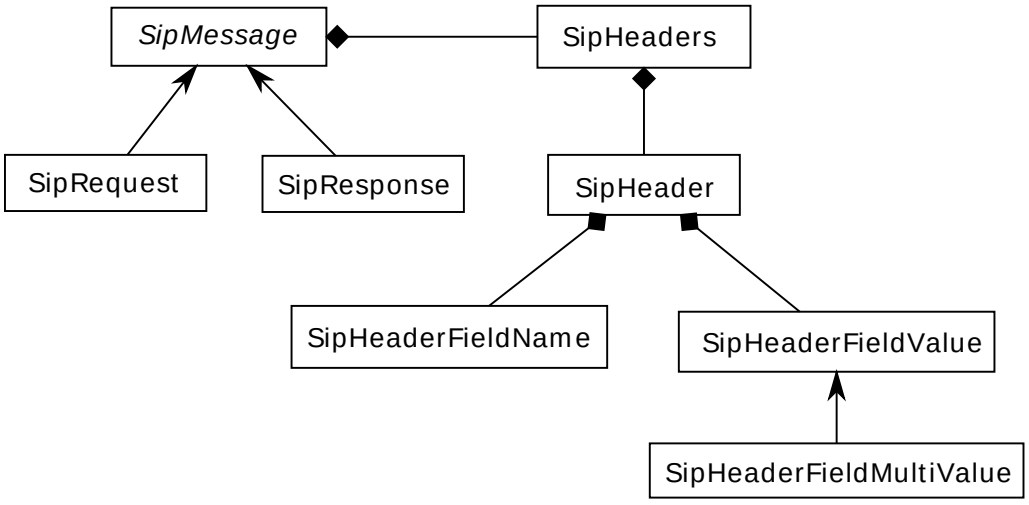
This message is a request. Here, is an example response. Here message bodies are plain text. Generally, SIP message body is either empty, either text. But RFC3261 states that body can contain any type of data, even binary data.

```
SIP/2.0 200 OK
Via: SIP/2.0/TCP client.atlanta.example.com:5060;branch=z9hG4bK74bf9
;received=192.0.2.101
From: Alice <sip:alice@atlanta.example.com>;tag=9fxced76sl
To: Bob <sip:bob@biloxi.example.com>;tag=8321234356
Call-ID: 3848276298220188511@atlanta.example.com
CSeq: 1 INVITE
Contact: <sip:bob@client.biloxi.example.com;transport=tcp>
Content-Type: application/sdp
Content-Length: 147

v=0
o=bob 2890844527 2890844527 IN IP4 client.biloxi.example.com
s=-
c=IN IP4 192.0.2.201
t=0 0
m=audio 3456 RTP/AVP 0
a=rtpmap:0 PCMU/8000
```

Thus, here is how those messages have been designed to enable message content access in Peers.

Figure 5.4. SIP message and its components



For the moment, don't mind body content, this is SDP (starting with v=0...). We will explain this syntax later. Just remember that this is not SIP but SDP, and thus, it's specified in another RFC.

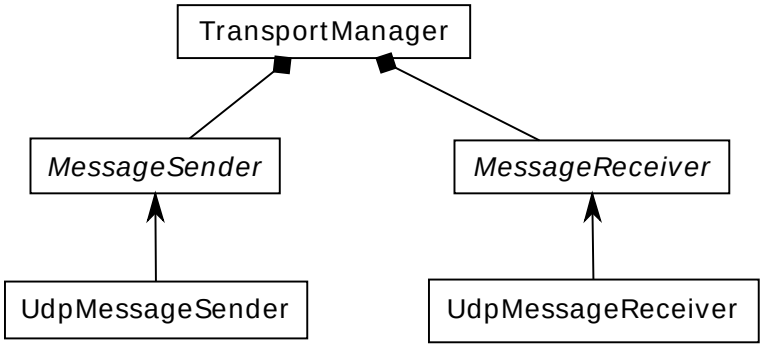
5.6.1.2. Transport

Now that we have seen SIP message bones, let's see how those messages are transported over the network.

Transport package is quite simple: TransportManager creates client transports and server transports. Those client transports and server transports are called message senders and message receivers. Actually, behind

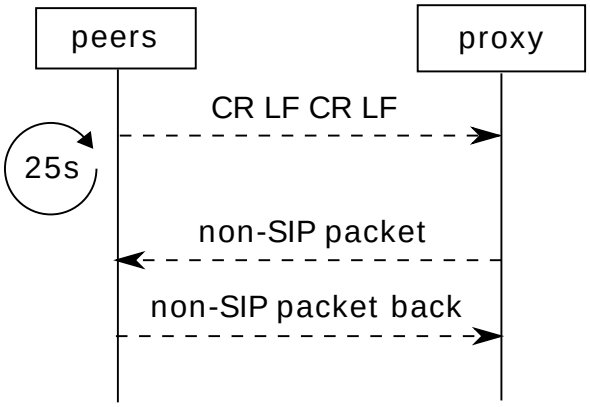
the stage, DatagramSockets are doing the real job. It must be noted that TCP transport is not supported in Peers. Most SIP stacks first support UDP, and then TCP. Peers does not break the rules. That's why TCP does not appear on the following class diagram. The transport layer is almost always responsible for message retransmissions (when first message sent has not been acknowledged). As SIP works over UDP, those message retransmissions are very important to avoid lost packets.

Figure 5.5. SIP transport management



To enable simple NATs traversal, each message sender sends "empty" SIP packets at scheduled interval (ping). Actually, those packets contain four bytes: 0x0d 0x0a 0x0d 0x0a. Those bytes correspond to a couple of carriage return / line feed ("\r\n" in java). Furthermore, on packet reception, if the first line of this packet does not contain "SIP/2.0", this packet is considered as a keep-alive / ping packet. Thus, a pong is sent, i.e. the same packet is sent back to the source of the previous packet. Sent and received at fixed rate, those packets prevent NATs from closing binding sessions between local and remote sip stacks. Nevertheless, it doesn't work with symmetric NATs.

Figure 5.6. Sending and receiving keep-alive packets



For datagram sockets indexing (in transport manager hashtable), it must be noted that keys are not strings nor integers, but SipTransportConnection objects. Those objects contain local ip address, remote ip address, local port, remote port and transport protocol used to convey packets. Thus when it's necessary to communicate with the same machine on the same port and using the same transport protocol, the same object is used.

Thread management is not the same for message sending and message reception. MessageReceiver implements Runnable, thus it must be started in its own Thread. It has been considered that was necessary to perform message reception in one Thread, as it can occur at any time. But message sending is not done in its own Thread. It's done in caller's Thread. This is the reason why applications relying on peers API must call UserAgent methods in a background thread.

Transport management has been done in a very naive way. In theory, UDP packets may contain several SIP messages, but this feature is not implemented in Peers. Actually on client side, this would probably be very odd to receive several SIP messages in the same UDP packet. In day-to-day life, it never occurs. Several multi-SIP messages UDP packets generally only occur between high-loaded servers, not on User-Agents. In theory, SIP messages bigger than MTU or 1300 bytes if MTU [http://en.wikipedia.org/wiki/Maximum_transmission_unit] is unknown are supposed to be sent on a reliable transport protocol such as TCP. In peers, this feature is not implemented, all packets are sent over UDP.

We won't explain all SIP routing theory, but remember that requests are routed using Route header if it's present, and request-uri domain name or IP address if Route header is not in message. Responses are routed using Via header. It generally contains an IP address and a port on which the response must be sent.

5.6.1.3. Transaction

Those of you who are familiar with databases probably already know transactions. We could also compare SIP transactions with financial transactions. In each case, transaction aim is the same: do something if each individual step is successful, otherwise do nothing. It's exactly the same with SIP. If any error occurs during transaction management, abort modifications on transaction-related objects (dialog state, etc.) and come back to the original state, before transaction management. Actually, this is quite dumb to start Peers transaction implementation description with transaction-fallback mechanism because no "failover" technique has been implemented in Peers... but at least, you are aware of it.

In SIP, a transaction is made of:

- exactly one request,
- eventually one or several provisional response(s) (status code between 101 and 199),
- exactly one final response (status code between 200 and 699).

We forget forking, it's intended. Fork is not implemented in Peers.

Transaction layer is probably the most complex layer in SIP specification. There are several transaction families. To find transaction family, you have to answer two questions:

- Does this transaction receive or sent the request on the network?
- Does this transaction create a Dialog?

Both questions have two mutually exclusive answers. Transactions that receive requests are called server transactions and transactions which send requests are called client transactions. Transactions that create a dialog are called invite transactions as INVITE is the only method that can create dialogs in RFC3261. Transactions that will not create a dialog are called non-invite transactions. Thus there are four transaction types:

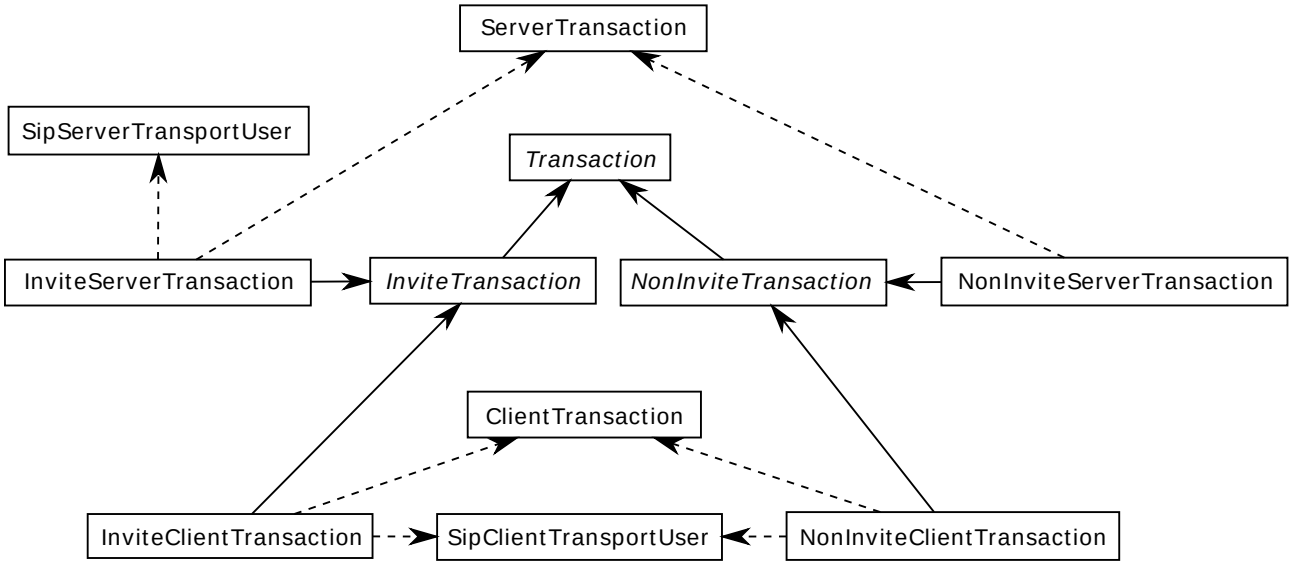
- invite client transaction,
- invite server transaction,
- non-invite client transaction,
- non-invite server transaction.

Transactions are uniquely identified using branch parameter in header Via (*z9hG4bK74bf9* in the following example) and method name (*INVITE*). As requests and responses belong to a transaction, those parameters are present in both request and response.

```
INVITE sip:bob@biloxi.example.com SIP/2.0
Via: SIP/2.0/TCP client.atlanta.example.com:5060;branch=z9hG4bK74bf9
Max-Forwards: 70
[...]
SIP/2.0 200 OK
Via: SIP/2.0/TCP client.atlanta.example.com:5060;branch=z9hG4bK74bf9
;received=192.0.2.101
From: Alice <sip:alice@atlanta.example.com>;tag=9fxced76sl
To: Bob <sip:bob@biloxi.example.com>;tag=8321234356
Call-ID: 3848276298220188511@atlanta.example.com
CSeq: 1 INVITE
Contact: <sip:bob@client.biloxi.example.com;transport=tcp>
[...]
```

Server and client aspect of transactions has been implemented as interfaces in Peers, and invite and non-invite property has been implemented in abstract classes. Thus those four transactions have been implemented in their own class in Peers, extending and implementing the appropriate class and interface, as shown in the following diagram:

Figure 5.7. SIP transactions class diagram



This class diagram also shows which classes are using transport layer using their corresponding SipXxxTransportUser interface.

There are many classes in peers transaction package because each transaction type has its own state machine and each state machine has one parent class and one class for each state. Those state machines are provided in RFC3261 but here is the corresponding class diagram for peers:

Figure 5.8. Invite client transaction state machine

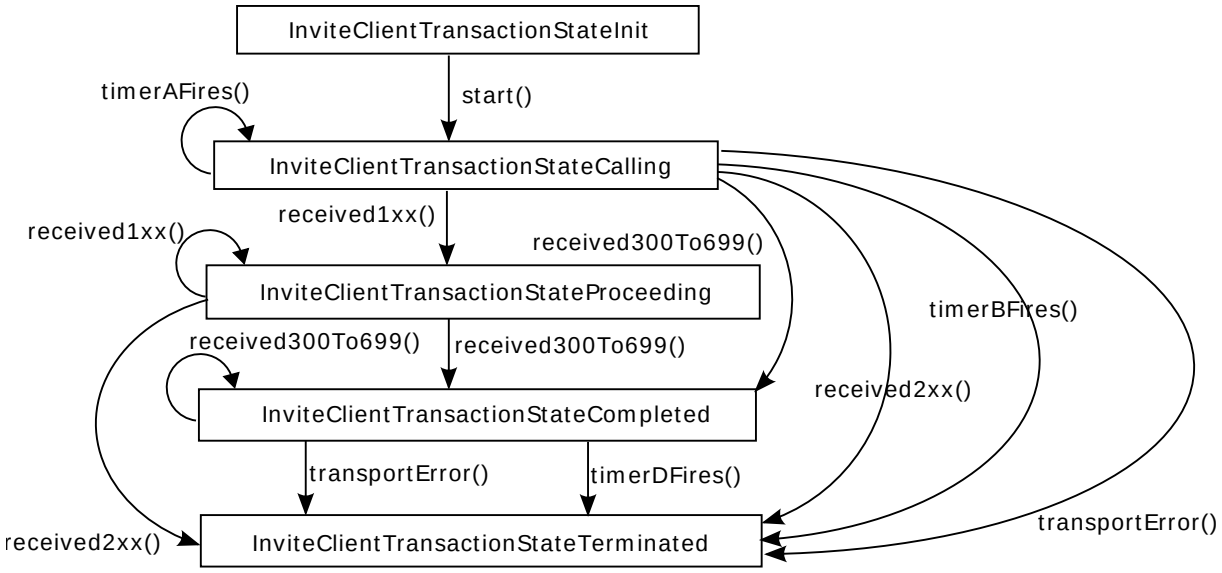


Figure 5.9. Invite server transaction state machine

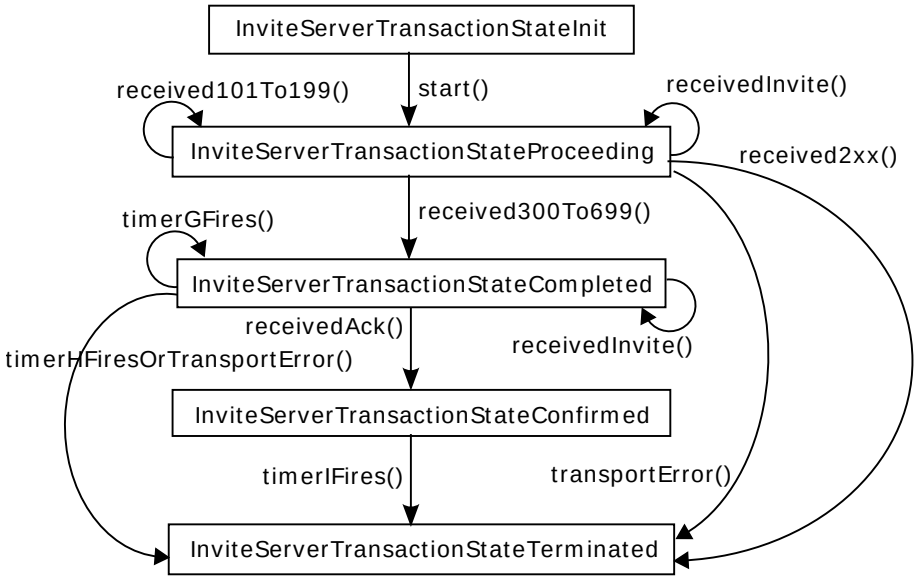


Figure 5.10. Non-invite client transaction state machine

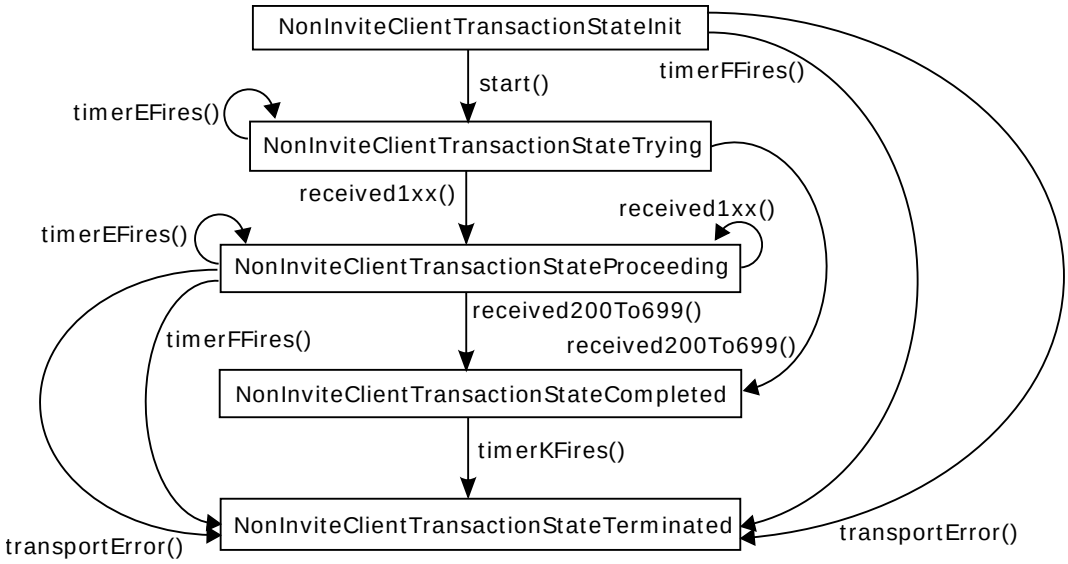
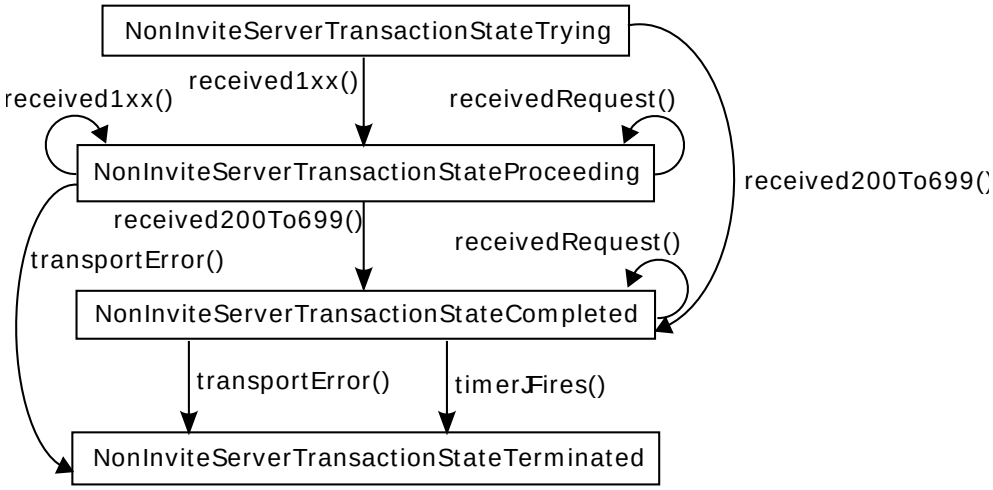
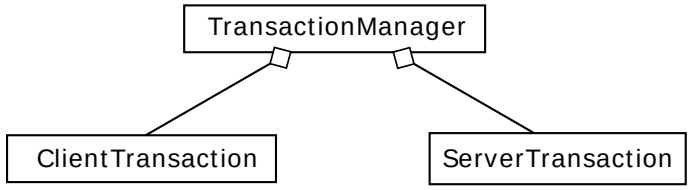


Figure 5.11. Non-invite server transaction state machine



Well, now that we know our transactions behavior, let's see their manager. Transaction manager works with transactions using their client/server property. Thus it uses ClientTransaction and ServerTransaction interfaces to handle them.

Figure 5.12. Transaction manager

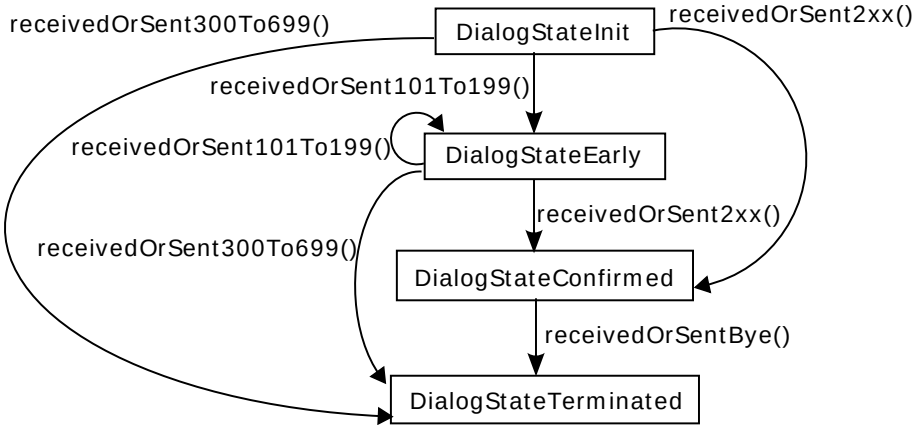


5.6.1.4. Dialog

Actually, in SIP specification there's a sort of confusion between transaction user and dialog layer. Several layers are using transaction layer on the upper side: core and dialog. Core is either User-Agent, Proxy, Registrar or Redirect Server; and dialog is transaction user.

Transaction user is probably the most simple layer in SIP. It contains Dialogs. A dialog is the representation of a media session on the control side. Remember there are two sides in SIP: media and control. Dialog is on control side, and media session is on media side. Media session is often the term employed in SDP and RTP. One state machine is necessary for dialogs. Please refer to RFC3261 for information about what must be inside a dialog. Bird view: local and remote contact addresses, unique id, etc. It's not a surprise, dialogs are managed using DialogManager.

Figure 5.13. Dialog state machine



Actually, Dialog is not really a group of transactions, but a transaction can occur within a dialog or not. The parameter that will determine if a transaction is performed within a dialog is its Call-ID header. To be exhaustive, a dialog is identified by its Call-ID, the tag parameter in header From and the tag parameter in header To. This is what you will see in peers.log. Thus a transaction which is performed within a dialog must use the same Call-ID, the same local-tag and the same remote-tag. Local-tag and remote-tag are To-tag and From-tag if the request is coming from the UAC (User-Agent Client) but they may be switched if the request is coming from the UAS (User-Agent Server), i.e. the one who received the call. Here is an illustration of dialog identifier components in request and response:

```
INVITE sip:bob@biloxi.example.com SIP/2.0
Via: SIP/2.0/TCP client.atlanta.example.com:5060;branch=z9hG4bK74bf9
Max-Forwards: 70
From: Alice <sip:alice@atlanta.example.com>;tag=9fxced76sl
To: Bob <sip:bob@biloxi.example.com>
Call-ID: 3848276298220188511@atlanta.example.com
[...]
SIP/2.0 200 OK
Via: SIP/2.0/TCP client.atlanta.example.com:5060;branch=z9hG4bK74bf9
;received=192.0.2.101
From: Alice <sip:alice@atlanta.example.com>;tag=9fxced76sl
To: Bob <sip:bob@biloxi.example.com>;tag=8321234356
Call-ID: 3848276298220188511@atlanta.example.com
[...]
```

In this example, the dialog id is the concatenation of 3848276298220188511@atlanta.example.com, 9fxced76sl and 8321234356. The request does not contain a tag parameter in header To. Actually, at this time, the dialog does not exist yet.

5.6.1.5. User-Agent

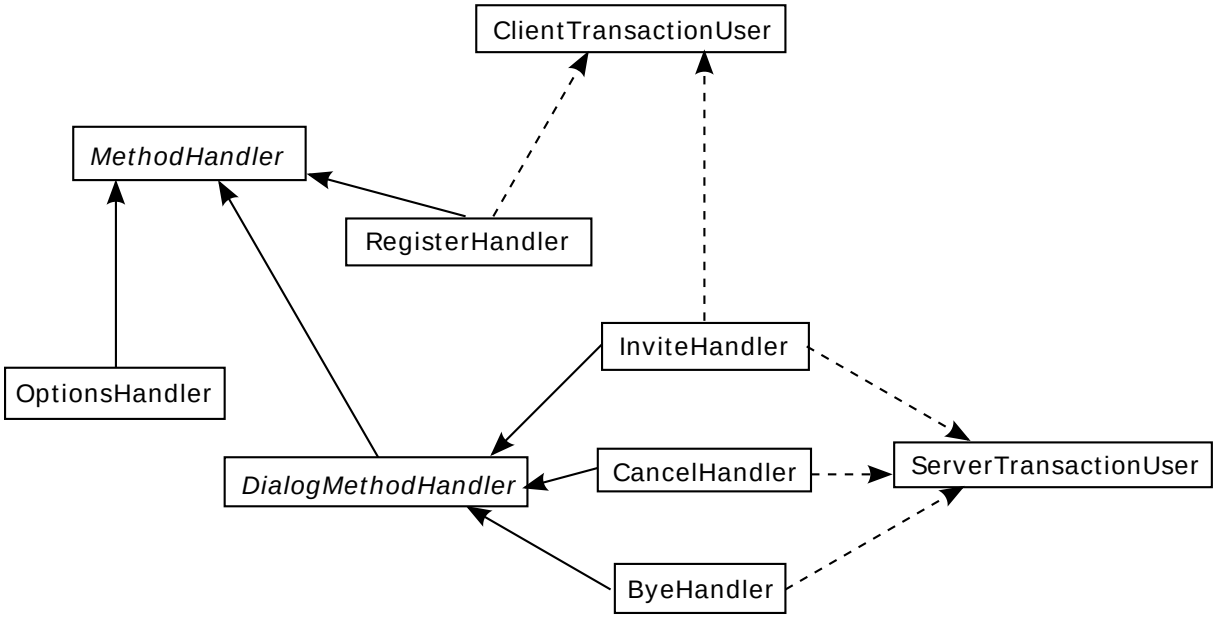
On top of transaction user layer, we find core layer. Core layer defines the SIP element role. On a SIP network, we've already seen that there were serveral nodes (roles):

- proxy,
- registrar,
- redirect server,
- user-agent.

Peers is a user-agent. It's the software employed by users to place or receive calls. Actually, a user-agent is just the SIP part of this software. User-agent can be considered as the image of the software in SIP stack. This is the reason why the corresponding package name is: net.sourceforge.peers.sip.core.useragent. Peers SIP core layer, or core role is User-Agent.

In SIP, the core layer is the brain. Depending on its role, it can be more or less sophisticated, but it's the place where general behavior is defined. Another property of SIP protocol is that complex things are managed in client software applications. Sometimes we hear that complexity is implemented on the edge of the network in SIP protocol. To support this complexity, each single feature has been implemented in a separate class in Peers. There are two types of classes in core layer: request managers and handlers. Handlers implement method-specific code. In basic SIP specification, there are several methods: INVITE, BYE, CANCEL, ACK, OPTIONS and REGISTER. Thus, each method has its own handler. All methods are not dialog-related methods, those methods are implemented in classes that inherit MethodHandler directly. Dialog related methods classes have a common abstract class called DialogMethodHandler. This class is reponsible for dialog construction and updates, calling the appropriate methods in dialog package. The following class diagram shows those classes:

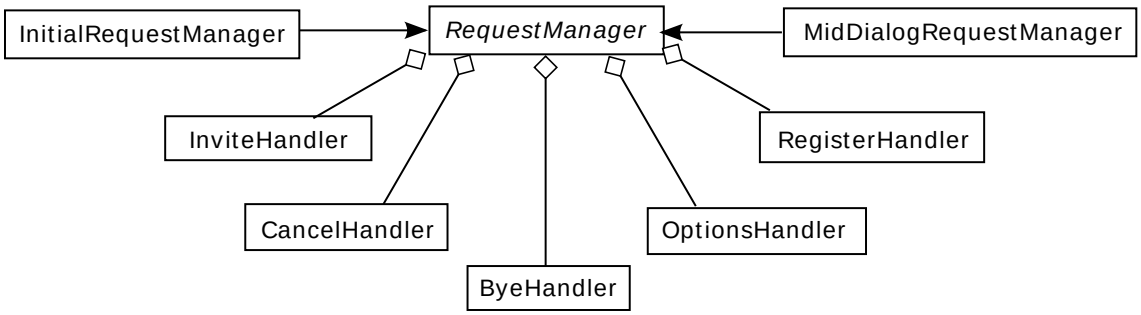
Figure 5.14. Method handlers



Method names are generally quite explicit: INVITE is there to create dialogs, CANCEL cancels dialogs in progress and BYE terminates dialogs. But INVITE can also be used to update codec, the IP address and port on which RTP packets can be sent. In this case, they are called re-INVITEs, but the actual method that is present in requests is INVITE. The trick to find if an INVITE is an initial INVITE or a subsequent INVITE (re-INVITE) is to look at To header tag parameter. If this parameter is defined, a dialog has already been created and thus, the INVITE request is within this dialog. REGISTER is used to register user-agent IP address and port, so that it can receive SIP calls. And OPTIONS is used to get information about what is supported in user-agent, proxy, etc. Well, I did not mention ACK for the moment... ACK is a very particular method. It does not generate any response. It's just employed to acknowledge the creation of a dialog on the client side. Thus the server side is notified of dialog creation.

A user-agent always contains both a user-agent client and a user-agent server. A user-agent client is responsible for requests sending and a user-agent server is responsible for incoming requests processing. In Peers, they are called UAC and UAS. Sorry, I must insist on one thing. The most important aspect of a request is whether this request occurs inside a dialog (subsequent request) or outside any dialog (initial request). This is really important because the processing in user-agent is absolutely different. In one case, you may have to create a dialog, in the other you may have to update this dialog. Some processing is the same for all methods for initial requests and some processing is the same for all subsequent requests. Request creation is the typical example. All methods share some processing in creation process. Thus, in Peers, InitialRequestManager is responsible for initial requests specific handling and MidDialogRequestManager is responsible for the requests handling occurring in a dialog.

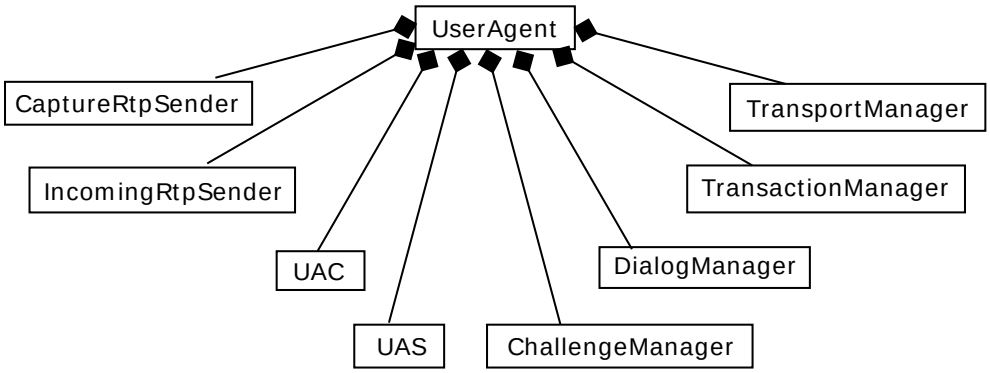
Figure 5.15. Request managers



The RequestManager is the class that keeps a reference to all Handlers. Even if it's an abstract class, it provides references to each method handler in its subclasses: InitialRequestManager and MidDialogRequestManager. Instances of those classes are references in UAC and UAS. May Peers act as user-agent server or user-agent client, it has to support intial requests sending, subsequent requests sending, initial requests reception and subsequent requests reception.

Let's come to our main class: UserAgent. UserAgent keeps references to many important objects: UAC, UAS, media related objects (CaptureRtpSender and IncomingRtpReader), and managers (ChallengeManager, DialogManager, TransactionManager and TransportManager). Each layer manager is referenced here.

Figure 5.16. User-Agent

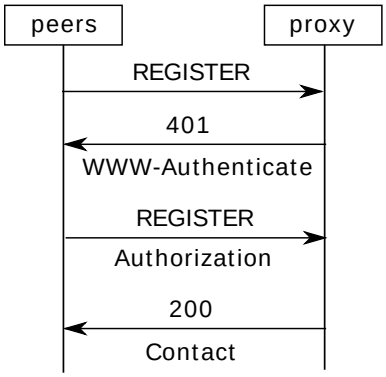


All core Handlers and Managers are instantiated within UserAgent constructor. Thus, when you instantiate a new UserAgent, you implicitly create its underlying layers objects. This is the reason why it's really easy to use Peers in external applications.

5.6.1.6. Challenge management

There's another important manager, it's ChallengeManager. In a SIP network, each element that receives a request can reject that request, asking the sender a common secret, this is a Challenge. ChallengeManager is responsible for Challenge management. Challenges are specified in RFC2617 [http://tools.ietf.org/html/rfc2617]. They provide a sort of authentication in SIP. Only two methods support challenges in peers: INVITE and REGISTER. And the only two response status codes supported by peers are 401 and 407. All combinations of supported methods and response codes share the same behavior for authentication. Thus this class handles them. This RFC defines a framework for HTTP authentication, but the same authentication framework is applied to SIP. Amongst the amount of authentication schemes specified for SIP, MD5 is the only authentication scheme supported in peers. AKA and other authentication schemes are not supported. Inspite those limitations, peers is able to authenticate on most standard SIP servers.

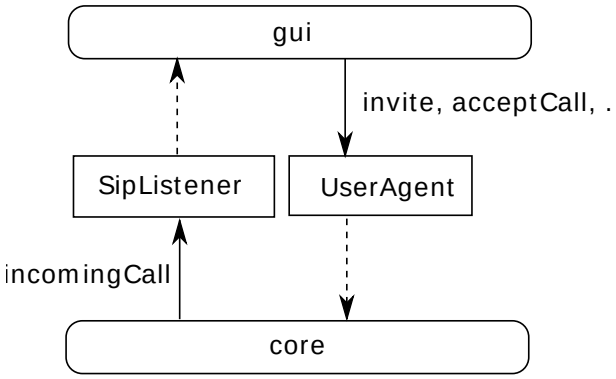
Figure 5.17. Registration example



5.6.1.7. Communication with user interface

Communication between core SIP layer (user-agent) and graphical user interface is done using an interface called SipListener. Thus, separation between SIP layers and user interface is clearly identified. The communication is done with gui package, but this graphical user interface can be replaced with another gui, a web interface, or even a console interface thanks to this generic way of communicating between sip core layer and upper layer.

Figure 5.18. Interaction between core and gui



As already seen, a software which needs to use peers SIP stack with another gui can instantiate a UserAgent object and then use this user agent instance to communicate with peers SIP stack. The SipListener interface offers the user interface a way to be notified of SIP incoming events: incoming call, callee pickup, remote hangup, etc. and user actions are provided to sip core through UserAgent methods: invite(), register(), acceptCall(), rejectCall(), etc. Incoming events correspond to methods in SipListener interface. This interface is implemented by a main class in upper layer (EventManager in gui package), the upper layer processes the incoming events and eventually takes action on SIP core layer depending on previous events and user actions.

5.6.2. SDP

SDP package is responsible for codec negotiation. SDP itself is the way media sessions are described, it's specified in RFC4566 [http://tools.ietf.org/html/rfc4566]. This codec negotiation is specified in RFC3264 [http://tools.ietf.org/html/rfc3264]. The negotiation principle is quite simple. At any time, an entity generates an offer, with all supported codecs. This offer is sent to another entity. Later, the entity that received the offer parses this offer, analyzes it, and generates an answer. There is always one answer for one offer. The answer depends on offer, it's not always the same.

In SIP theory, an offer can be present in either INVITE or 200 body. If the offer is in INVITE, the answer is in 200, and if the offer is in 200, the answer is in ACK body and INVITE body is empty. In practice, this former case is extremely rare. SDP contains critical information about media streams. It provides the IP address and the port on which the softphone wishes to receive RTP packets, but it also describes the payload types that it supports. Remember, SDP gives media description, not media content. The protocol that transports media streams is RTP. This protocol transports encoded media with a specific wrapping format, this is the payload type. Here is an example SDP session description:

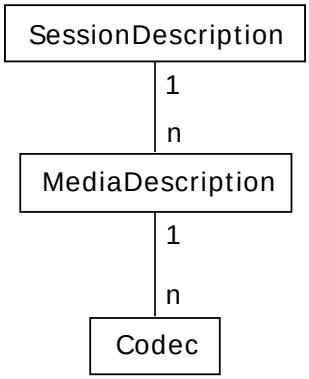
```
v=0
o=alice 2890844526 2890844526 IN IP4 client.atlanta.example.com
s=-
c=IN IP4 192.0.2.101
t=0 0
m=audio 49172 RTP/AVP 0
a=rtpmap:0 PCMU/8000
```

Reading this SDP, we can conclude that this entity is listening on IP address 192.0.2.101 on port 49172 for RTP PCMU packets (payload type 0), sampled at 8000 Hz. Payload types and their corresponding codec are given in RFC3551 [http://tools.ietf.org/html/rfc3551].

If we take a look at Peers source code, SDPManager is the place where everything is done at SDP level. This class generates offers, parses answers to extract useful information (IP address, port, payload

type), and generates answers based on incoming offers. The object employed to describe an SDP body is SessionDescription. A SessionDescription can contain several MediaDescriptions. A MediaDescription typically corresponds to an audio stream or a video stream (video is not supported in peers, but its sdp stack could support it in theory). A MediaDescription can contain several Codecs.

Figure 5.19. SDP objects



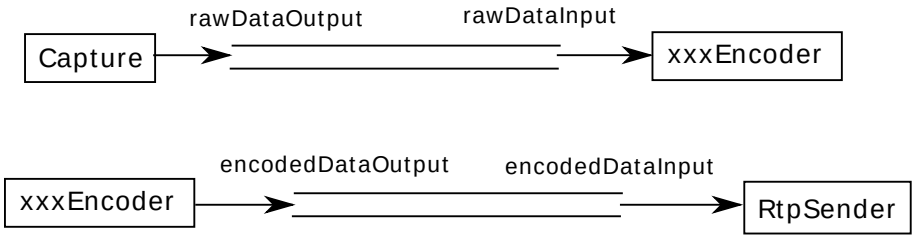
The combination of IP address, port and Codec is grouped as a MediaDestination. This class eases RTP targets description. The content of those classes corresponds to RFC4566 [http://tools.ietf.org/html/rfc4566] parameters description.

5.6.3. Media

The main classes of this package are IncomingRtpReader and CaptureRtpSender. Actually, IncomingRtpReader is responsible for RTP header parsing, media decompression and media playback ; and CaptureRtpReader is responsible for microphone capture, media encoding and RTP header insertion. Thus, CaptureRtpSender has references to Capture, Encoder and RtpSender instances. Each of this class implements Runnable, and is running in a separate Thread. Data is transmitted using pipes (PipedOutputStreams and PipedInputStreams) amongst those media handling objects. Nevertheless, IncomingRtpReader does not use separate threads for parsing and media playback.

The following figure illustrates pipes between Capture, xxxEncoder and RtpSender which allow data transmission from microphone capture to RTP packet sending.

Figure 5.20. Media pipes



5.6.3.1. Javaxsound

This module and its corresponding package relies on oracle javasound. Javasound is standard sun javasound API. Thus you can use their web pages for more information and tutorials [http://download.oracle.com/javase/tutorial/sound/TOC.html]. All interaction with javasound API is done in JavaxSoundManager. Thus, if you want to use another sound API, you can implement your own version of AbstractSoundManager, defined in peers-lib. Peers-javaxsound defines SourceDataLines for media

playback and TargetDataLine for media capture. A global AudioSystem class is there to retrieve all information about sound card, etc. AudioDataFormats give description about codec and audio bitstream format. The last important aspect of javasound is Line. Lines are used for stream control: start/stop, etc. Peers captures audio data at 8 kHz, using 16 bits samples, one channel (mono), and signed little-endian samples.

The use of javasound for media capture and playback is critical. Even if it's not the simplest java media API, it has the advantage of being tested by sun on each supported platform (windows, linux, mac, solaris, etc.). Peers has been tested successfully on linux, windows and mac os 10.6. Javasound has many drawbacks: few guaranteed features, no standard audio data format (frequency, sample size, etc.). But it's already integrated in java standard edition API, and it avoids third-party libraries with native parts, etc.

AbstractSoundManager defines two methods to open and close lines on the host sound card and one method to "write" data to speakers. To "read" from microphone, the SoundSource interface has been defined. This interface is implemented by AbstractSoundManager for standard usage but also by FileReader to read raw media files and send them over the network to the remote party. Check <mediaFile> in peers.xml config file.

In peers-javaxsound, the line used to play sound (SourceDataLine) and the line used to capture sound (TargetDataLine) are opened one after another, not in parallel. Tests proved that it is more secure to open one line after another. Parallel lines opening can create JVM crashes. Once opened, those line take and provide data.

5.6.3.2. Media debug

Peers provides an option in its configuration file called <mediaDebug>. This parameter takes a boolean value. If true, almost all media data will be written to several files in media/ directory corresponding to each step in both directions.

From peers user to remote contact:

- data captured from javasound before encoding => _PCM_SIGNED_8000.0_16_1_le_microphone.output
- encoder input => _g711_encoder.input
- encoder output => _g711_encoder.output
- rtp sender input => _rtp_sender.input
- rtp session output (including rtp header) => _rtp_session.output

From remote contact to peers user:

- rtp session input (incoming rtp packets received, including rtp header) => _rtp_session.input
- data provided to javasound to really play sound => _PCM_SIGNED_8000.0_16_1_le_speaker.input

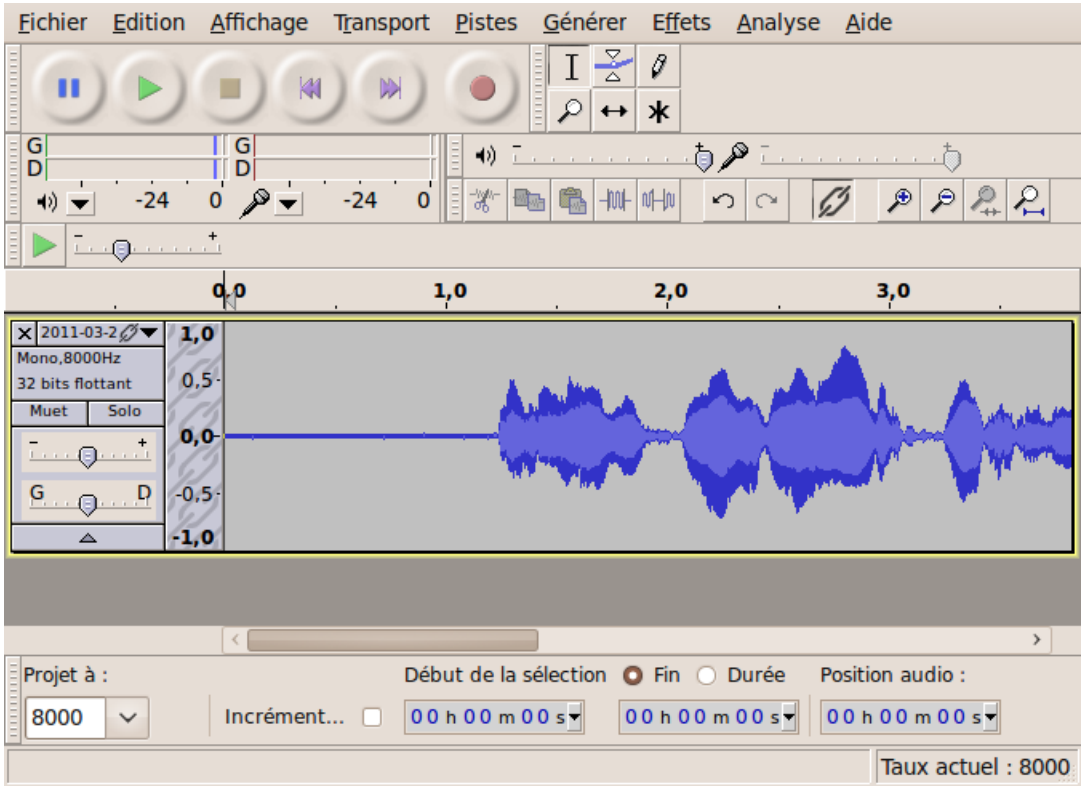
As you can see, decoder input and output are not written to their file, this is because this step didn't required too much debug during implementation. All files start with date using the following format: yyyy-MM-dd_HH-mm-ss. Remember that this parameter is only useful for debug purpose, it slows down media streams very much and it has not been extensively tested. It will probably work in simple scenarios, but probably not in complex ones.

PCM_SIGNED_... give the format of raw data contained in this file. Those files can be imported in Audacity [http://audacity.sourceforge.net/] using File > Import > Raw data and the following parameters:

- Signed 16 bit PCM
- Little-endian
- 1 Canal
- Sample Frequency: 8000 Hz

Thus you can see what is being provided to javasound. This may help media debugging.

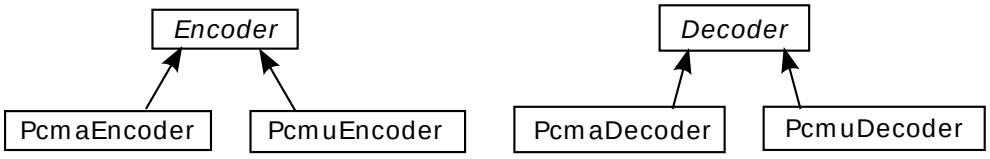
Figure 5.21. Importing raw data in Audacity



5.6.3.3. Codecs

Peers codecs have been implemented the most generic way as possible. Thus two abstract classes Encoder and Decoder have been created. Encoder is a Runnable and Decoder is not. Those abstract classes are overloaded by codecs. As PCMU and PCMA are supported in peers, each one has it own Encoder and Decoder:

Figure 5.22. Codecs implementation

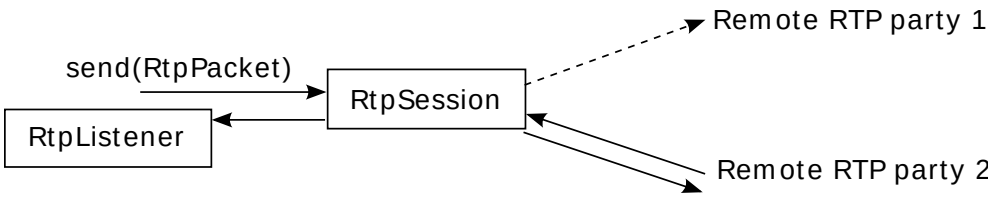


Encoder and Decoder classes only define a process(...) method that will work on input to generate an output depending on codec algorithm. Encoder is started at the beginning of a call and only the codec-specific encoding is done in real codec class.

5.6.4. RTP

Peers RTP stack is very simple. The main class of this stack is RtpSession. This class handles the packet sending and reception. It relies on java 5 ExecutorService to receive packets. Nevertheless it does not create a separate thread to send packets, it simply sends them on demand using send(RtpPacket) method. As other packets, RTP package provides a listener to notify the reception of an RTP packet. When a packet is received, the raw data is parsed using RtpParser class. Thus a new RtpPacket is created and provided to the RtpListeners which subscribed to RTP packet reception on RtpSession.

Figure 5.23. RTP packet flow



An RtpPacket contains standard RTP headers and data (ssrc, sequenceNumber, payloadType, etc.). Peers RTP stack is based on RFC3550 [http://tools.ietf.org/html/rfc3550]. Before it can be used, an RtpSession has to be started using start() method. Later it can be stopped using stop() method. An RtpSession uses an initial remote IP address and a port number to send first RTP packets before any RTP packet has been received from remote RTP party. Once a packet has been received, if the source IP address or the port differs from the previous received packet, the remote IP address and port are updated with latest one. This enables NAT traversal in a few cases.

5.6.5. GUI

Last but not least, graphical user interface. Peers is based on swing for gui management. Once more, swing is already integrated in oracle JRE. It's light, efficient and uses staight-forward development methods. Peers uses OS native look and feel instead of sun's metal look and feel. OS look and feel will probably be moving faster than metal look and feel and OS integration is thus improved.

GUI is based on a class which manages all events coming from SIP layer and from user and dispatches events amongst GUI components. This class is EventManager. This class receives invocations from sip layer using its SipListener interface. It then dispatches events to the appropriated frames. Several frames are implemented in gui (from the most important one to the least important one):

- MainFrame
- CallFrame
- AccountFrame
- AboutFrame

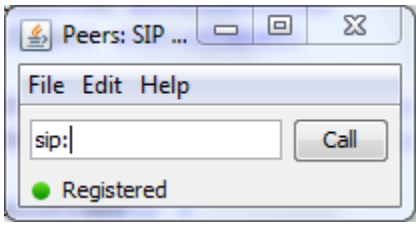
Some of those frames have been designed using netbeans [http://netbeans.org/] with an empty project and then integrated in peers main source code, implementing only triggers on appropriated events (typically button clicks). Thus, graphical design has not been too harsh.

5.6.5.1. MainFrame

MainFrame is actually the class that contains the main() method of peers, so this is the first class to be instantiated. MainFrame creates and keeps a reference to EventManager. Then EventManager creates and manages other frames and their corresponding events.

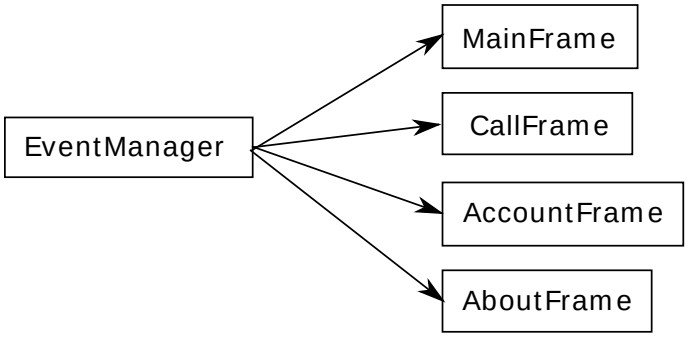
Here is a screenshot of MainFrame:

Figure 5.24. MainFrame in action



5.6.5.2. EventManager

Figure 5.25. GUI event manager

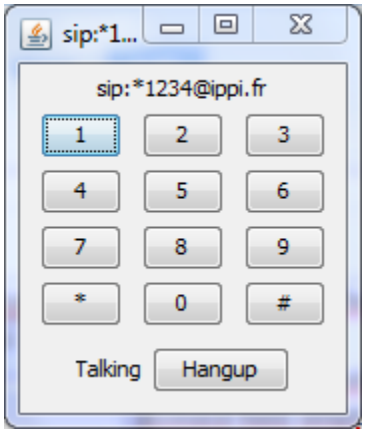


EventManager is listening to frames through listeners interfaces: MainFrameListener, CallFrameListener. EventManager also implements ActionListener to receive events from MainFrame JMenu. Thus when user is clicking on Edit > Account, the corresponding event reaches EventManager and EventManager eventually instantiates a new AccountFrame to enable user configure sip account.

5.6.5.3. CallFrame

When user places or receives a new call, a new CallFrame is created and displayed to user. Here is an example after callee pickup:

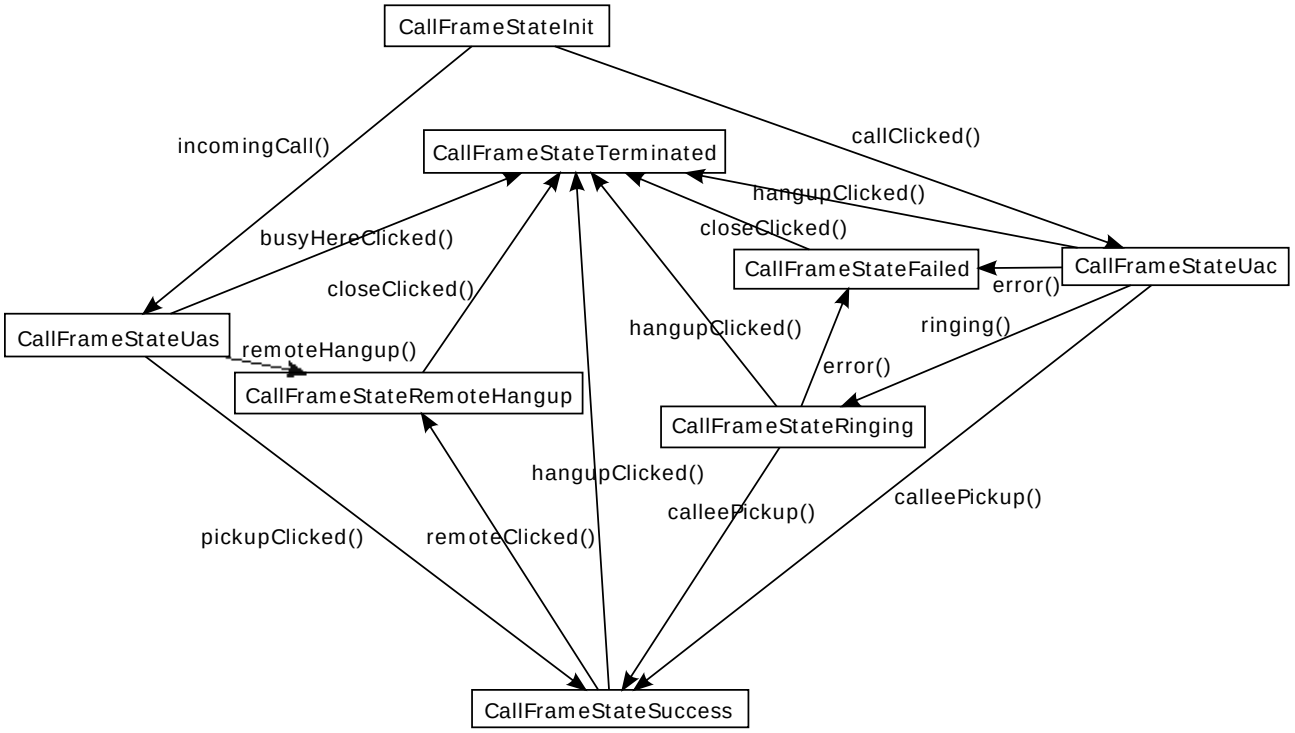
Figure 5.26. CallFrame in action



Peers defines a state machine to maintain the state of this window. This state machine uses the standard model employed in peers (one class per state and a parent class for all states). The same state machine

is employed for calls originating from peers user and terminating towards peers user. Here is this state machine:

Figure 5.27. CallFrame state machine

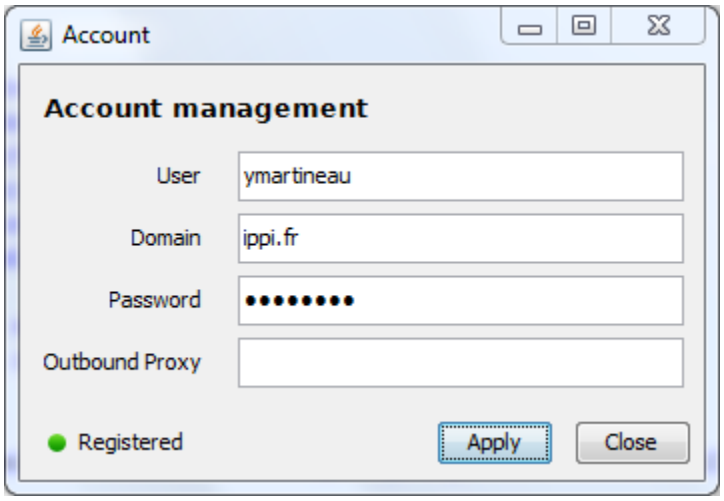


The keypad displayed in CallFrame has its own Keypad class to improve CallFrame readability and share this simple component from core CallFrame code. This Keypad component is visible in previous CallFrame screenshot.

5.6.5.4. AccountFrame

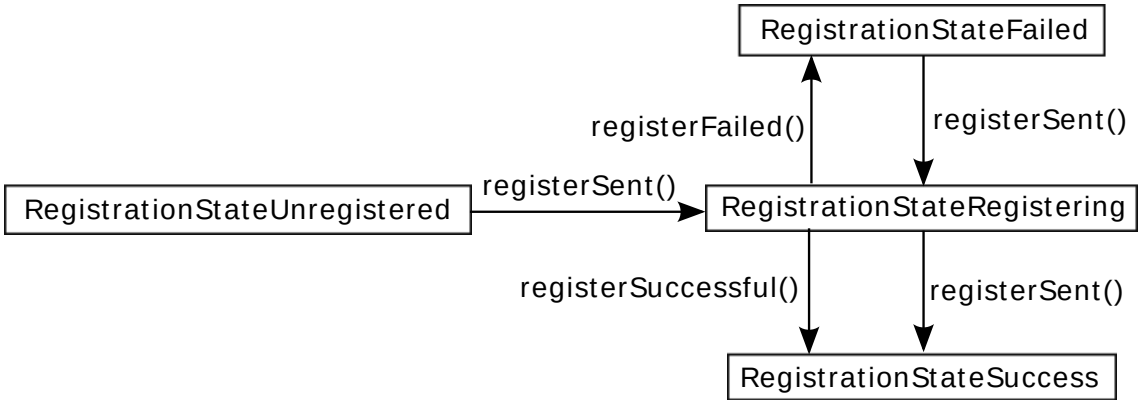
This frame enables peers user to configure his or her sip account with a minimal interface:

Figure 5.28. AccountFrame in action



This class also displays the registration state of the corresponding account. Actually, this registration state is also displayed in MainFrame to let user know that his or her account is registered on regular peers startup. As this "registration state" component had to be displayed on several frames, a specific component has been written in Registration class. This class also makes use of a state machine, because of asynchronous registration events coming from sip stack.

Figure 5.29. GUI Registration state machine



5.6.5.5. AboutFrame

AboutFrame is not that interesting, it's just useful to display license to user.