Implementing a RFC 3428 SIP IM client on Android
A quick-starter's guide

by Michael Maretzke

30th October 2009

Background

A good friend of mine presented a HTC Hero to me one day. He had a very impressive application augmenting the live video image of the phone's camera with some rendered information. It caught my attention – and curiosity – immediately. The same evening, I downloaded the Android SDK and did my first steps. I ordered one of these phones and started to look for an interesting application to implement – ideally with some telephony background.

It came the same friend asked for a SIP based IM application on Android – the challenge. Some research produced Jean-Luc Deruelle[1] ported the JAIN SIP stack already to Android and Emmanuel Proulx[2] did a basic AWT SIP IM client already.

The ingredients for this SIP IM client – and an interesting journey to get in touch with Android programming paradigms.

Setting up the environment

The first website to visit when looking for Android development information is definitely http://developer.android.com/.

From this website you can download the SDK[3] which nicely integrates with the Eclipse IDE. The installation of the SDK and setting up the Eclipse IDE is described here: http://developer.android.com/sdk/installing.html.

During the development of this application the Android SDK 1.6r1 in combination with Eclipse 3.5.0 (Galileo) on a Ubuntu 9.04 (Jaunty) system was used. Final testing was done on a HTC Hero device.

To test the Android based IM client another Java based AWT client named "SIP IM AWT" was used. The AWT version is an adapted version of "TextClient" done by Emmanuel Proulx.

After unpacking the archive[4] the root directory of "sip-im-awt" contains an ANT build file. With Java and ANT installed the following commands brings up the AWT SIP IM client.

```
ant package
java -jar sip-im-awt-1.0.jar michael_awt 6060
```

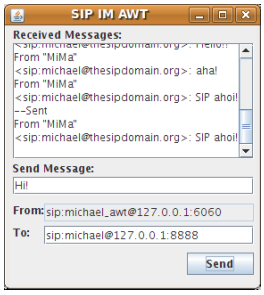


Figure 1 SIP IM AWT client

The root folder of the AWT SIP IM client contains an Eclipse project as well. Simply select "File" → "Import..." → "Existing Project into Workspace" and select the root folder of the just extracted AWT SIP IM client.

The Android based SIP IM client needs some further work to be done before executing it on the emulator – or the device.

First, ensure to have Eclipse with the Android SDK installed. Download[5] the Android SIP IM client. Unpacking the archive results in the root directory named "sip-im-android". The root-folder contains an Eclipse project. Import the project as described above for the AWT client. Assuming a correctly setup working environment, clicking on the "Run as..." button allows you to select "Android Application" as the way to execute the project.

Eclipse takes care of packaging, signing and starting the emulator for you. Starting the emulator takes quite some time …

Meanwhile switch the perspective of Eclipse to the DDMS perspective. This is a collection of useful debugging views of your Android emulator – containing the "LogCat" called system log of your emulator.

After a while, your emulator should look like this.

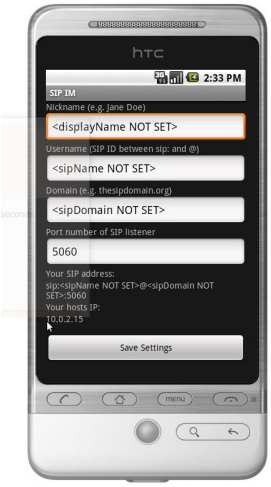


Figure 2 SIP configuration UI on Android

The client started for the first time and didn't have any SIP relevant parameters stored. Now, it's time to enter them.

- Nickname: MiMa
- Username: michael
- Domain: thesipdomain.org
- Port: 8888

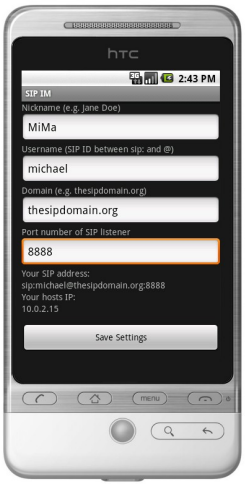


Figure 3 filled in parameter settings

The nickname corresponds to the DisplayName in the SIP message to be used to send IM – SIP MESSAGE – requests. The username corresponds to the portion between "sip:" and the "@" in your SIP address. The domain is the portion between the "@" and the ":". The port will be used to setup the SIP UDP and TCP listeners for the client. Pressing on "Save Settings" leads to the IM UI screen.

To switch on and off the on-device debug log, press "MENU" and select "Debug Window". The lower part of the screen now shows debug information produced by the client. Scrolling down unveils that a SIP stack is now listening on port 8888 – what a surprise!

Now, send a message from the Android device to the AWT IM client.

Put "From Android to AWT!" in the "Send Message..." named text field and "sip:michael_awt@10.0.2.2:6060" in the "To" text field. Press the "Send" button.

Figure 4 Sending IM from Android to AWT

Ideally, the two clients on your system look like the ones above. A remark on the 10.0.2.2 IP address: the emulator takes 10.0.2.2 as a synonym for the emulator hosting system's 127.0.0.1 address[6].

Before we send a message from the AWT client to the Android Emulator we need to configure a port forward for port 8888 from the hosting system towards the emulator.

```
telnet localhost 5554
```

In the emulator's console, type

```
redir add udp:8888:8888
```

Now, type "Back from AWT to Android!" in the message field of the AWT client and "sip:michael@127.0.0.1:8888" into the address field.

Great – it seems to work on the emulator. Let's have a look at the workflow when deploying the application to the device.

Figure 5 Sending IM from AWT to Android

The funny thing about putting the application on a real device is – it's really easy. And it's ways faster than working with the emulator.
To setup a connection to the device, follow the description in the SDK documentation[7].

Well, that's it! Shut down your emulator and deploy your application using Eclipse. Now, you can utilize the DDMS view to monitor your real device. That's really cool!

Can't over trump this? Well, try debugging on the device … Set a breakpoint and click on "Debug as ..." … That is really impressive!

Architecture

The architecture of the IM client is surprisingly simple: a layer takes care of the UI interactions and the other of the SIP communication.

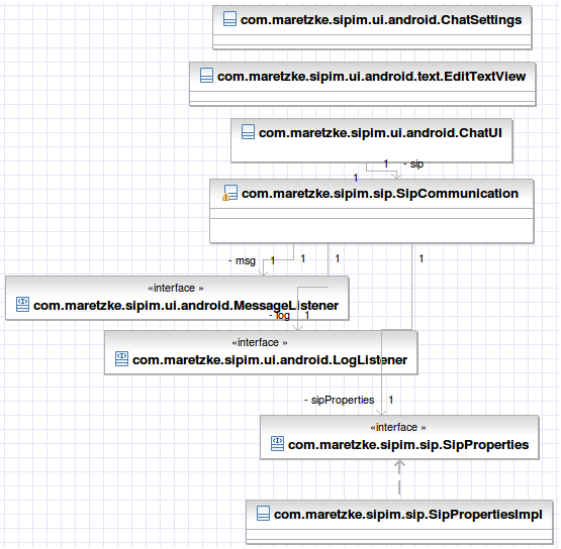


Figure 6 Architecture overview

The classes `SipCommunication`, `SipProperties` and `SipPropertiesImpl` are part of the communication layer. `ChatUI`, `ChatSettings` and `EditTextView` are part of the Android UI layer. `MessageListener` and `LogListener` are interfaces to connect both layers.

The Android UI

The heart of the application is the class `ChatUI`. It is called during creation of the overall application and takes control over the major control flow.

ChatUI

Walking through the code, initially some variables are defined. The section …

`ChatUI.java: 102ff`

```
// handler to update the UI layer from a non-UI thread
// this is necessary to update the log messages and chat
// messagesfrom the SIP communication layer (a non-UI
// thread)
final Handler updateUIHandler = new Handler();

// this Runnable will be associated to the above mentioned
// updateUIHandler to update the UI elements within the
// scope of the UI managing thread
final Runnable runnableUpdateUI = new Runnable() {
  public void run() {
    updateLogUI();
    updateChatUI();
  }
};

/**
 * This method is called from the UI Handler runnable to
 * update the log message UI element.
 */
private void updateLogUI() {
  logBox.setText(contentLog);
}

/**
 * This method is called from the UI Handler runnable to
 * update the chat message UI element.
 */
private void updateChatUI() {
  chatBox.setText(contentChat);
}

…

/**
 * Print a log message in the UI element for log messages.
 * Implementation of LogListener interface.
 */
public void logMessage(String msg) {
  contentLog += msg + "\n";
  updateUIHandler.post(runnableUpdateUI);
}

/**
 * Print a chat message in the UI element for log
 * messages. Implementation of MessageListener interface.
 */
public void messageReceived(String from, String msg) {
  contentChat += from + ">> " + msg + "\n";
  updateUIHandler.post(runnableUpdateUI);
}
```

… defines an interesting element. The SIP communication layer receives incoming messages and wants them to be shown in the UI. Same applies for log messages. Android, however doesn't allow non-UI threads to update UI elements. Hmmm.

The solution is to implement a `Handler` in the UI thread to trigger the UI update. The methods `logMessage()` and `messageReceived()` utilize the `Handler` to post a message into the UI thread containing the runnable. The runnable simply calls the methods `updateLogUI()` and `updateChatUI()`. These methods update the UI.

The very same principle is applied to show the progress dialog when the application saves the user preferences to a file. The `ChatSettings` Activity is started as an asynchronous action:

ChatUI.java: 188 and 277

```
Intent intent = new Intent(this, ChatSettings.class);
startActivityForResult(intent, CHAT_SETTINGS);
```

After finishing, the onActivityResult() method is invoked and allows the ChatUI to act on the results.

ChatUI.java: 298ff

```
public void onActivityResult(...) {
  if (requestCode == CHAT_SETTINGS) {
    progress = ProgressDialog.show(this, "", "Saving
    settings... Please wait!", true);
    …
    Thread saveData = new Thread() {
      public void run() {
        …
        save information to file
        setting up the stack
        …
        updateUIHandler.post(hideProgressDialog);
      }
    };
    // start the above defined thread
    saveData.start();
  }
}
…
final Runnable hideProgressDialog = new Runnable() {
  public void run() {
    progress.hide();
  }
};
```

The I/O intense portion is outsourced to a thread to keep responsiveness of the UI high. The method starts a progess dialog and then defines and starts the I/O thread. The final line in the thread informs again the `Handler` to put the `Runnable hideProgressDialog` into the UI message queue. The progress dialog disappears.

Another nice snippet of code is the one to determine the non-loopback IP address of your Android device. The current Android SDK implementation of `InetAddress.getLocalHost().getAddress()` always (!) returns the loopback address (`127.0.0.1`) – no matter if there's a routable address available. This turns out to be an issue when needing a routable IP address. The internet, however, produced a solution[8]:

ChatUI.java: 442ff

```
private String getLocalIpAddress() {
  try {
    for (Enumeration<NetworkInterface> en =
        NetworkInterface.getNetworkInterfaces();
        en.hasMoreElements();) {
      NetworkInterface intf = en.nextElement();
        for (Enumeration<InetAddress> enumIpAddr =
          intf.getInetAddresses();
          enumIpAddr.hasMoreElements();) {
          InetAddress inetAddress =
          enumIpAddr.nextElement();
          if (!inetAddress.isLoopbackAddress()) {
          return
          inetAddress.getHostAddress().toString();
          }
        }
      }
  } catch (SocketException ex) {
    logMessage("getLocalIpAddress - Exception caught.");
    ex.printStackTrace();
  }
  return null;
}
```

ChatSettings

The information transfer between the various Activities is done utilizing the `SharedPreferences` class.

To read the values it's enough to get a reference to them like this:

ChatSettings.java: 47

```
SharedPreferences sp =
getSharedPreferences("sip-im-android", MODE_PRIVATE);
```

Modifying access needs the `Editor` being involved – and don't forget to `commit()` your changes!

ChatSettings.java: 97ff

```
SharedPreferences.Editor spe =
getSharedPreferences("sip-im-android", MODE_PRIVATE).edit();
…
spe.commit();
```

EditTextView

The `EditTextView` class extends `TextView` to become an editable version of `TextView`. This version allows additions to the text and is able to scroll content.

The Communication layer

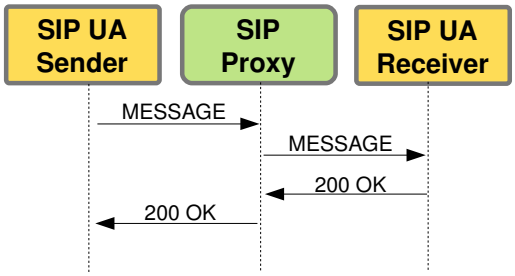
The RFC 3428 "(SIP) Extension for Instant Messaging"[9] defines an IM extension for the SIP signalling protocol.

An example message according to RFC 3248 is shown below.

```
MESSAGE sip:michael_awt@10.0.2.2:6060;transport=udp SIP/2.0
Call-ID: 8280d71d051470fede6c78a52cffc638@10.0.2.15
CSeq: 1 MESSAGE
From: "MiMa" <sip:michael@thesipdomain.org>;tag=sip-im-
android-v1.0
To: "michael_awt" <sip:michael_awt@10.0.2.2:6060>
Via: SIP/2.0/UDP 10.0.2.15:8888;branch=branch1
Max-Forwards: 70
Contact: "MiMa" <sip:michael@thesipdomain.org:8888>
Content-Type: text/plain
Content-Length: 12

The Message.
```

The message flow as defined in RFC 3248 is shown below and implemented by the two SIP IM clients. They only omit the SIP Proxy shown in between.



The implementation of `SipCommunication` is straight forward. The constructor takes over `SipProperties` which hold important information about the SIP stack to create – but also information about the user sending messages – and call back methods to inform the UI about log messages (`LogListener`) and incoming IM messages (`MessageListener`). The constructor instantiates a SIP stack instance with UDP and TCP listen points.

The methods `sendMessage()` and `processResponse()` implement the above drawn message flow according to RFC 3248.

The method `processRequest()` deals with incoming IM SIP messages and hands them over to the UI thread to be displayed.

The `setNewListenerPort()` method is invoked by the UI whenever the listening port of the IM client is changed by the user.

`SipCommunication.java: 308ff`

```
public void setNewListenerPort(SipProperties  sipProperties)
… {
    log.logMessage("Recreating SipStack.");
    sipStack.stop();
    createSipStack(sipProperties);
  }
```

The method basically stops the current SIP stack and re-instantiates a new one matching the new port number.

Limitations

This prototype implementation obviously has some limitations.

First of all, it seems a bit odd to enter IP addresses in the SIP address to have some form of routing at all. Ideally, a SIP Proxy took over the address resolution.

Second, the current contact management implementation is not connected to anything inside the phone. It's a hand-made and very simplistic solution to save some time typing. Furthermore, contacts are only stored to the user preference file (which in turn is only stored when the user presses the “Save Settings” button on the ChatSettings UI.

Last, the implementation is straight forward and not tested to whatever error conditions – hence may crash quite easily (run it with no IP address … having mobile and WiFi switched off …).

Lessons learned

Android UI

The handling of the UI was quite a surprise. Knowing MS Windows, Java and some other Windowing toolkits it turned out to be quite complex to notify UI threads about information changes happening in non-UI threads. But in the end it seems logical – since the phone should be able to respond to asynchronous events like phone calls any time. So, the above described mechanisms utilizing the `Handler` class seemed like a good way forward.

UI Responsiveness

In the spirit of responsiveness it seems logical as well to outsource long-running or I/O intense processes to separate threads. I learned my lesson during the progress dialog exercise.

Information share

Information sharing between Activities happens via the `SharedPreferences` class. Any other idea?

Emulator and networking

It is important to understand the Emulator and how it solves networking. Especially, knowing how to set the port forwarding (redir) and the link to the host's local loopback address (10.0.2.2) is quite useful.

Credits

I'd like to say thank you especially to Mudumbai Ranganathan (better known as Ranga) from the NIST institute for the excellent JAIN SIP reference implementation – a true reference! Thanks as well to Jean-Luc Deruelle for proving

that the RI implementation works on Android. This helped reassuring quite a lot! And last, thanks to Emmanuel Proulx creating the "template" for this application – the TextClient called SIP IM client.

Important links

The SIP IM client can be downloaded from:

http://www.maretzke.com/pub/howtos/sip_im/index.html

For questions, discussions, improvements feel free to contact me via michael@maretzke.com.

[1]Jean Deruelle's blog entry on his nightly port of the JAIN SIP stack to the Android platform: http://jeanderuelle.blogspot.com/2008/10/jain-sip-is-working-on-top-of-android.html

[2]Emmanuel Proulx introduces the JAIN SIP API on the Oracle (ex-BEA) developer's portal: http://www.oracle.com/technology/pub/articles/dev2arch/2007/10/introduction-jain-sip.html

[3]Android SDK download link: http://developer.android.com/sdk/index.html

[4]Download the sources from: http://www.maretzke.com/pub/howtos/sip_im/index.html

[5]Download the sources from: http://www.maretzke.com/pub/howtos/sip_im/index.html

[6]See the remarks on emulator and networking: http://developer.android.com/guide/developing/tools/emulator.html#emulatornetworking

[7]SDK documentation on connecting a real Android device to the development environment: http://developer.android.com/guide/developing/device.html

[8]Get the ip address of your device by „Martin": http://www.droidnova.com/get-the-ip-address-of-your-device,304.html

[9]See e.g. http://www.rfc-editor.org/rfc/rfc3428.txt for the RFC 3248