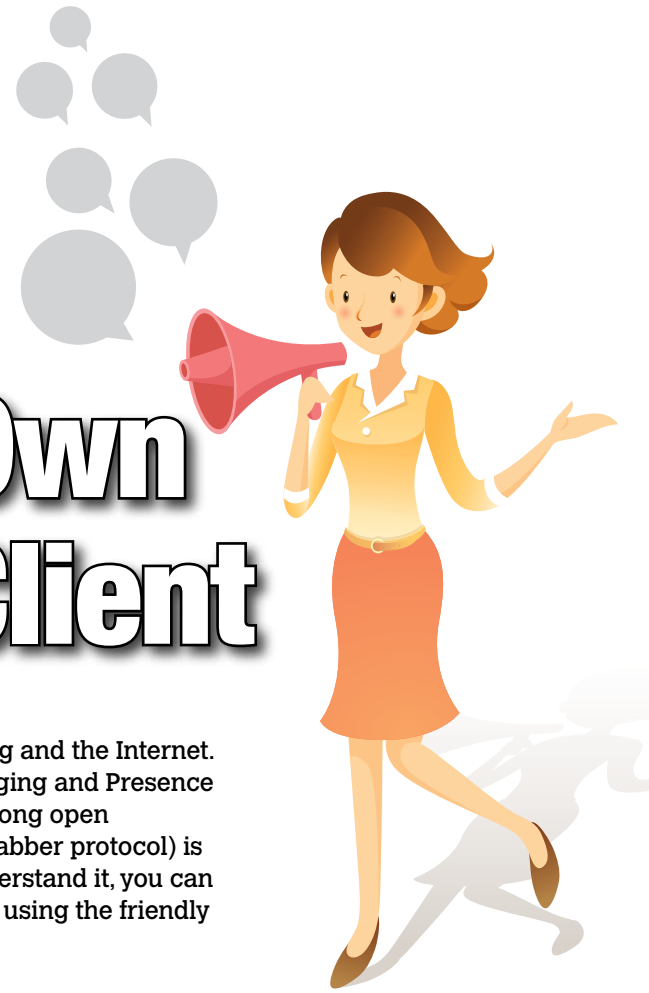
Use XMPP to Create Your Own Google Talk Client



Instant communication is now the essence of social networking and the Internet. The popular Google Talk, which uses XMPP (Extensible Messaging and Presence Protocol), made this Instant Messaging protocol prominent among open standards protocols. Exploring XMPP (formerly known as the Jabber protocol) is fun—it is a transparent and simple architecture. Once you understand it, you can easily write your own XMPP/GTalk clients from the ground up, using the friendly and powerful Python scripting language.

XMPP is an XML-based open standards protocol. It uses XML streams to implement the entire message communication system. XMPP is used in a de-centralised client-server architecture, in which a server acts as an intermediary for the message transfer, and also manages services like the user account registration, authentication, buddy list database, etc. Since our primary focus is on clients, we won't dig any more into the server part—we will just consider the server to be a 'black box' entity available at a specific IP address/hostname and port number, which meaningfully responds to our XMPP requests.

We connect to the server using a TCP socket in our program. An explanation of networking, TCP/IP, IP addresses, ports and sockets is too much information to put into this article, so if these concepts are new to you, you could visit http://en.wikipedia.org/wiki/Internet_socket for some quick reading.

Since our purpose in this article is to code our own Google Talk client, the hostname of the server we will use is gmail.com, and the port we will use is 5222, the default port for the XMPP service.

The sequence of the initial interaction between client and server is as follows:

1. The client connects to the server and sends credentials like the username and password.
2. The server validates the received credentials against its user database and sends a response to the client.
3. When authentication is successful, the client receives a response containing *presence notification* data. This is a collection of *presence* data from different buddies of the user, which the client authenticated to the server. Presence is explained below.

As already noted, the communication between client and server is in the form of

XML streams, over the connection created at the beginning of the interaction.

Before we get down to the code, we need to define some terms and underlying concepts that are involved with the use of the XMPP protocol. To write an XMPP client, we don't need a thorough understanding of the XML streams that are exchanged between client and server, since we use libraries that abstract away the complexities of the underlying protocol, and provide an API to us. In this article, we'll be using Python to write our code, and the python-xmpp library provides us a neat API. We need to understand the types of communication and interaction involved, however, so let's begin.

Note: The XML streams shown below as examples are **not** as they would actually be in a live XMPP session, since I have omitted attributes of some elements, and omitted portions of the XML stream that are not required to illustrate the concept. Since the python-xmpp library handles the nitty-gritty of building, sending, and receiving the XML streams, you don't need to memorise the XML samples—just look at them as illustrations, and **not** as code that you have to write yourself.

Resource

The XMPP client could be running on any of several types of computing devices, ranging from mobile phones and embedded devices to laptops and full-fledged desktops. The type of devices on which the client is running can be exposed in the *from* attribute in XML streams sent by the client, and this information is termed 'resource'. The *from* attribute is in the format *from="userid@domain/resource"*. For example, a client running on an Android phone could send something like "*userid@gmail.com/Android*". This identifier can be very useful in many ways: administration tools that manage many clients could segregate the clients on this basis. The server could adjust its responses to the type of resource being used—for example, if connected from a mobile phone, the length of responses, the size of images, etc. could be held to a minimum to avoid long download times on a slow GPRS connection.

With this information exposed to the clients that buddies are running, it is then possible for clients to also treat different buddy resources in a different manner. For example, I might want to send the message "Hello Android guys!" to all my buddies who are connected from an Android device, but send "Hello, netbook guys!" to all buddies connected from netbooks.

Stanza

A stanza is an atomic command in XMPP, and one of the fundamental structures in the protocol layer. You can send an unlimited number of stanzas over an established connection between server and client.

Once an authenticated connection and XML stream is established between the server and the client, the client and the server can exchange (as part of the messaging service) three basic XML stanzas in their communications—<*message*/>, <*presence*/>, and <*iq*/>. We'll look at each of those in turn.

Presence

As the word signifies, *presence* is a method by which a client/user notifies buddies and the XMPP server of its current status—whether the user is online or offline. When a client authenticates, it sends a presence notification stanza to the server, with some metadata that describes the client—the current status text, whether Busy, *Available* or *Away*, the resource parameter, user nickname, the client name, etc. When the server receives the presence stanza, it sends a copy of the data to all users who are in the buddy list of the user who sent the presence stanza. If you use the Pidgin messenger to connect to GTalk, hover the mouse pointer over an entry in your buddy list to see some of that metadata.

Here's a sample of an XML presence indication stanza:

```
<presence from="slynux@slynux.com/Android">
  <show>xa</show>
  <status>Writing for LFY</status>
</presence>
```

Message

Message is an XML stanza used to send messages between users. It looks like what's shown below:

```
<message from="slynux@gmail.com/Home"
         to="slynuxguy@gmail.com"
         type="chat">
  <body>Hey, Whats up ?</body>
  <subject>Query</subject>
</message>
```

IQ

IQ (Info/Query) is an XML stanza that is similar to GET and POST requests in the HTTP protocol. We use IQ to request for some information from the server, and collect the response for further use. If the request is invalid or cannot be processed, the server returns an error stanza. The XMPP protocol states that every *iq* stanza should contain an *id* attribute, whose value is generated by the client (the XMPP library used). This *id* attribute is returned in the *iq* response from the server, and can be used by the client to match the received response with a specific *iq* request it sent (useful in the case of the client sending multiple *iq* requests in a batch). Since the underlying details of generating a unique *id* are dealt with by the library (in our case, python-xmpp), we don't have to worry about that in our code.

For example, the following snippet is a roster query (more about that below) as sent to the server:

```
<iq from="slynuxguy@gmail.com"
    id="7"
    to="slynuxguy@gmail.com/Pidgin"
    type="get">
  <query xmlns="jabber:iq:roster"/>
</iq>
```

Shown below is a sample response from the server:

```
<iq to="slynuxguy@gmail.com/Pidgin"
    id="7"
    type="result">
  <query xmlns="jabber:iq:roster">
    <item jid="user3@gmail.com/Home"/>
    <item jid="user4@gmail.com/Adium"/>
    <item jid="user4@gmail.com/Android"/>
  </query>
</iq>
```

You'll see multiple entries for *user4—user4@gmail.com/Adium* and *user4@gmail.com/Android*. *Adium* and *Android* are resource parameters, as discussed earlier. This tells us that *user4* has logged in from two clients.

Roster

A roster in XMPP is basically a buddy list, which contains a presence attribute for each user item. Your roster contains a list of Jabber user IDs (called JIDs) and the state of your presence subscriptions with those entities. When you come online, your client announces your presence to the server, and it handles the rest—both notifying your contacts that you are online, and fetching their current presence to display in your client interface. Your roster is updated when you send a presence stanza.

Here is a sample of an XML stream that combines each of the three types of stanzas in a session with the server. Note that this is not a debug output—the actual debug output streams will have many more details. This is just a structural example of an internal XML stream, intended to conceptually demonstrate interaction between client and server, and is taken from XMPP documentation. In this sample, the client (identified by “C:” in the sample) has authenticated with *gmail.com* as *user5@gmail.com*.

```
C: <stream:stream>
C:   <presence/>
C:   <iq type="get" id="1">
       <query xmlns="jabber:iq:roster"/>
     </iq>
S:   <iq type="result" id="1" >
       <query xmlns="jabber:iq:roster">
         <item jid="user1@gmail.com"/>
         <item jid="user2@gmail.com"/>
         <item jid="user3@gmail.com"/>
       </query>
     </iq>
C:   <message from="user5@gmail.com"
              to="user2@gmail.com">
       <body>Hello world!</body>
     </message>
S:   <message from="user3@gmail.com"
              to="user5@gmail.com">
       <body>Kudos to you</body>
     </message>
C:   <presence type="unavailable"/>
C: </stream:stream>
```

There can be many kinds of requests like the buddy add request, buddy remove request, group chat request, etc. All of these are handled by XML streams. XMPP includes a method to secure the stream from tampering and eavesdropping. This channel encryption method makes use of the Transport Layer Security (TLS) protocol, along with a "STARTTLS" extension that is modelled after similar extensions for the IMAP, POP3 and ACAP protocols.

Hands-on client code

We will now take a look at how we can write our own Google Talk XMPP client from scratch using Python, a simple yet powerful language that includes a staggering amount of functionality in its standard library—often referred to as ‘batteries included’. In addition, there are plenty of separately installable extensions and libraries for Python—for almost everything you can do with other languages.

For coding the client, we will use the XMPP module for Python, which you will probably need to install. If you are using a Debian-based distribution like Ubuntu, run the following command:

```
$ sudo apt-get install python-xmpp
```

For other distributions, if a package is not available in the repositories, download the tarball and install it as follows:

```
$ wget http://downloads.sourceforge.net/project/xmpppy/xmpppy/0.5.0-rc1/
xmpppy-0.5.0rc1.tar.gz?use_mirror=nchc
$ tar -xzvvf  xmpppy-0.5.0rc1.tar.gz
$ cd  xmpppy-0.5.0rc1
$ sudo python setup.py install
```

Now let us write the base client code. The code below will connect the client to the server and authenticate. A base client in just 12 lines of code—can you believe it?

```
#!/usr/bin/env python

import xmpp

user="username@gmail.com"
password="password"
server="gmail.com"

jid = xmpp.JID(user)
connection = xmpp.Client(server,debug=[])
connection.connect()
result = connection.auth(jid.getNode(), password,"LFY-client")

connection.sendInitPresence()
```

```
while connection.Process(1):
      pass
```

Save the code in the file *base.py* and run it:

```
$ chmod a+x base.py
$./base.py
```

Now run a Pidgin instance and this client at the same time. Use two different GTalk accounts to log in with Pidgin and this client. After running this script, in Pidgin, hover the mouse pointer over the script user (the user under whose account the script authenticated itself). Check the status—"LFY-client" — that has been set by the script. Try sending messages to the script user account— of course, it will not respond, because this is still a very basic client and we haven't specified what to do when some message arrives.

Now, let's find out how to look into the XML stream. Switch the debugging on: change the *connection = xmpp.Client(server,debug=[])* line to *connection = xmpp.Client(server)* by removing the *debug=[]* Parameter. Now when you run this script, you can see the XML stream, as shown in Figure 1.

Writing a GTalk bot

You might have used or encountered GTalk bots that auto-respond to messages—for example, Google's own transliteration bots. If we add a transliteration bot as a buddy and send a phonetic word to it, it will send back a Unicode string transliterated to the corresponding Indic language. It isn't hard to develop something like that if you have a back-end application that can be used for transliteration. Now, we will modify the current *base.py* by adding a message handler. When a message is received, this handler will reply with the message "Welcome to my first GTalk Bot :)".

```
#!/usr/bin/env python

import xmpp

user="username@gmail.com"
password="password"
server="gmail.com"

def message_handler(connect_object, message_node):
        message = "Welcome to my first Gtalk Bot :)"
        connect_object.send( xmpp.Message( message_node.getFrom()
,message))

jid = xmpp.JID(user)
connection = xmpp.Client(server)
connection.connect()
result = connection.auth(jid.getNode(), password, "LFY-client")
connection.RegisterHandler('message', message_handler)

connection.sendInitPresence()
```

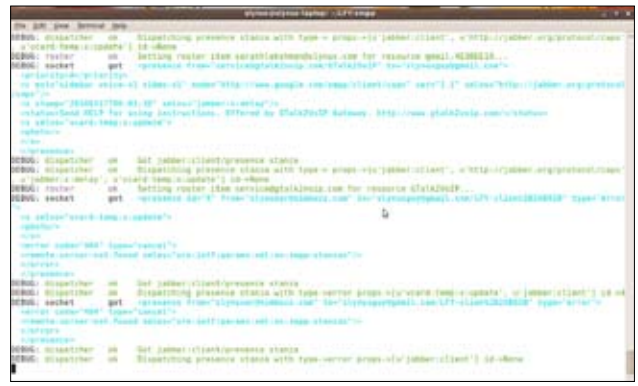


Figure 1: Debug information (XML stream) in the terminal window

```
while connection.Process(1):
      pass
```

Here, *connection.RegisterHandler('message',messag e_handler)* is used to specify that the *message_handler()* function must be called when a message stanza is received. The two arguments passed to the function are a connection object, and the message stanza node. By using the attribute function *getFrom()*, we obtain the JID of the user who sent the message stanza, and we send our reply to that user. To obtain the text of the received message, use the function *message_node.getBody()*.

Remote-control shell bot

You have seen how to write a very simple bot in a few lines of code. Now let's play with another idea: remotely controlling computers via an XMPP bot. Normally, we use SSH (Secure Shell) for remote administration—it gives us a shell at the remote machine, so we can execute commands on that computer. Can we do something similar with an XMPP bot? Yes! Let's modify our simple bot to act as a remote-controlled shell bot. Replace the simple bot's *message_handler()* function with this new one:

```
def message_handler(connect_object,message_node):
        command = str(message_node.getBody())

        process = subprocess.Popen(command,shell=True,stdout=subprocess.
PIPE, stderr=subprocess.PIPE)
        message = process.stdout.read()
        if message=="":
            message=process.stderr.read()

        connect_object.send( xmpp.Message( message_node.getFrom()
,message))
```

Note: You will need to add *import subprocess* at the top of the program file, since the module is now used in the *message_handler()* function.

In this message handler, we retrieve the text of a message received from a sender, and run it as a command

using the *subprocess* Python module. We check the file descriptors *process.stdout* (standard output of the sub-process) and if that does not return any data, then *process.stderr* (the standard error of the sub-process). We then send the data returned from the sub-process to the sender of the message.

Like before, from Pidgin (logged in as a different user), try sending commands like *ls*, *cat /proc/cpuinfo*, etc, to this bot's GTalk ID. You should see the expected results returned as messages in your Pidgin window.

This bot has no access-control—any user in its buddy list could send commands and have them executed. Let's add a simple check for the message sender's ID, so we can restrict command execution permission to a single buddy. Any other users sending commands to the bot should receive an "Access denied" type of error message. Replace the *message_handler()* function with this new one:

```
def message_handler(connect_object,message_node):

        admin = "admin_user@gmail.com"
        from_user = message_node.getFrom().getStripped()

        if admin == from_user:  # allow to execute command only if admin
requested
             command = str(message_node.getBody())

             process = subprocess.Popen(command,shell=True,stdout=subproces
s.PIPE, stderr=subprocess.PIPE)
             message = process.stdout.read()
             if message=="":
                  message=process.stderr.read()
        else:
                  message="Access denied!\nContact system admin"

        connect_object.send( xmpp.Message( message_node.getFrom()
,message))
```

Find invisible users

Most instant messaging protocols support invisible users, so that anyone can remain online without being noticed by anyone in their buddy list. GTalk is no exception. When I started playing with XMPP, I noticed an interesting thing about GTalk's implementation: it implements user 'invisibility' on the client side, not the server side. We can easily find such invisible-yet-online users in our buddy list by listening to the presence notifications. (As explained before, whenever a client joins a server/network, it sends a presence notification for the user.) When a user authenticates to the server and becomes invisible, the client sends an 'unavailable' presence to all buddies in the roster. Thus, presence notifications with a presence type of 'unavailable' means that the user is in invisible mode. All GTalk clients ignore this type of presence; hence, those users are not shown in the buddy list. But we can write our own Python program to grab the list of invisible users:

```
#!/usr/bin/python -W ignore::DeprecationWarning

import xmpp

user="user@gmail.com"
password="password"
server="gmail.com"

def presenceHandler(conn, presence):
      if presence:
         if presence.getType() == "unavailable":
             print presence.getFrom().getStripped()

print "Invisible users:"

jid = xmpp.JID(user)
connection = xmpp.Client(server,debug=[])
connection.connect()
result = connection.auth(jid.getNode(), password,"Client Name")

connection.RegisterHandler('presence',presenceHandler)
connection.sendInitPresence()

while connection.Process(1):
      pass
```

Writing a GUI front end for your client

So far, our simple Python GTalk client using the XMPP module was all about command-line programs that must be run in a terminal. You can develop a Graphical User Interface to make your client more user-friendly and increase usability. The two most popular GUI toolkits for use with Python are Qt and GTK. Qt is the base of the KDE desktop environment, while GTK is the base for the GNOME desktop environment. While writing the GUI, it is important to remember that you need to use threading to keep the XMPP and GUI part in separate threads. The *connection.Process(1)* function is to be called in an infinite loop. In the above programs, we used a *while* loop. In the case of Qt or GTK, both maintain a window by using an event loop—so using another infinite loop inside the Qt/GTK window classes will, at some point, result in 'frozen' and unusable windows. Instead, use the Python threading module to run the XMPP part in another thread.

Hope you enjoyed the hacks around Google Talk and XMPP. Happy hacking till we meet again. END

By: Sarath Lakshman

The author is a Hacktivist of Free and Open Source Software from Kerala. He loves working on the GNU/Linux environment and contributes to the PiTiVi video editor project. He is also the developer of SLYNUX, a distro for newbies. He blogs at www.sarathlakshman.info