# Senior Project Final Report

BatChat - Mobile Communications Application

Fall 2014

Adam Jacques

## Introduction

In my senior project, I attempted to build a mobile application that would allow users of the immensely popular Android platform to connect to and communicate over the open and standardized XMPP/Jabber text based communications protocol. This protocol, which has been adopted by the IETF, defines a completely cross-platform and extensible network protocol that anybody can adopt and build real-time applications on top of. The protocol defines the complex endpoint discovery and message encoding rules that are required of all applications that wish to be able to communicate with other XMPP based applications.

This project was designed and developed in collaboration with Cody Martin. Throughout the entire development process we sought to emulate a real-world development environment much like you would see while working in a full-time employment position. Basically we wanted to use this project as a way to prove that we can work like a serious development team, which would use issue management systems to track regressions and features along with code reviews for all changes made on the code base.

One of the many different uses of the XMPP protocol, is to use it to transmit real-time communications between two or more different users. It works somewhat like email where there is no single centralized server and each user will instead connect to and use a server that they control or have authorization to use.

We chose to create this project after attempting to use one of the other popular XMPP clients for the Android platform. It did not offer a very good user experience as it had a very primitive user interface and did not support many of the more advanced features that the XMPP protocol allows for. This caused us to realize that this was an area of the market that was poorly covered by current applications as XMPP is used by many businesses to facilitate communication between internal employees and business to business communications, much like a real-time version of email.

The XMPP protocol specification represents a few base protocols that must be implemented by any endpoint and a multitude of optional extra protocols that can be implemented by any endpoint if they wish. To open up a usable XMPP stream, the client must first negotiate the message encoding and encryption protocols that will be used to transport the XMPP stanzas. Then you are required to authenticate yourself as a valid user with one of several different authentication mechanisms that will protect the user's passwords from eavesdroppers.

For this project we focussed only on implementing the required specifications to act as a user agent, which is a client that will connect to a server that implements the XMPP server specification and protocols. This would allow us to create a minimum viable product like many business articles tell you to develop first.

## Language choices

I had originally planned to implement the entire application including the networking subsystem in pure-Java with no other language dependencies. This would have been beneficial as it would have reduced the number of third-party libraries that we depended on, but as I began to design the general code architecture of the protocol system and message processing pipeline, I realized that to build a loosely-coupled class structure using a more functional and less-constricting language than Java would allow me to greatly reduce the complexity of the network stack. I settled on implementing the

networking subsystem in Scala because it's a JVM-based language meaning that it can compile to the Java byte code. This was very important because it allowed me to easily compile my Scala code using the standard Scala compiler and directly inject it into the compiled Android APK application files. Finally, the main reason I picked Scala was because that implemented functions as first-class objects in the language, which greatly benefited me when I was building the message parsing and processing routines in the network subsystem.

Scala is also a very compact language that allows for a lot to be represented in a small amount of code. I had partially implemented some of the different parts of the networking subsystem in Java before I switched to Scala and many of the functions were five to six times longer than their Scala equivalents. Scala required a lot less code to implement many features, partially since the language is compact and partially because Scala also comes with a very rich feature set that provided a lot of help in coding.

The Scala language and documentation promotes using immutable objects in place of mutable objects. While it did not require you to use immutable types, as that would not make sense in all scenarios, it had many features that allowed you to use them quickly and easily. I used as many immutable objects as made sense allowing me to be sure that types would never change while I was using them. This allowed me to take advantage of multi-threading and other concurrent models as was required in certain areas of the program.

## High-level overview

The XMPP protocol builds upon the OSI stack by adding more layers on top of the transport layer. A fully authenticated connection might resemble this stack:

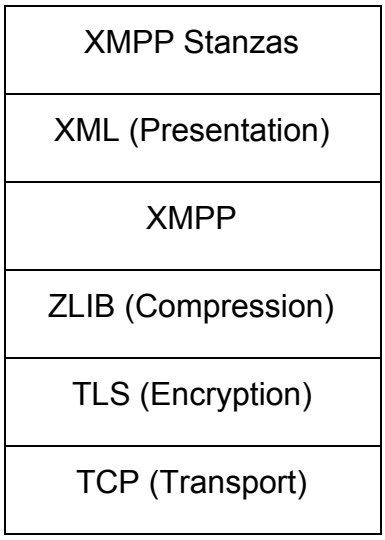


**Figure 1. The XMPP network stack**

In my project, I was responsible for implementing all of the protocols and interfaces for every layer positioned above the TCP transport layer. I designed a class structure that allowed for each layer to operate independently of the layer that is above or below it by exposing only Java-based InputStream and OutputStream classes to the classes responsible for the byte-oriented protocol layers. This allowed me to easily plug in new layers as I wrote them since developing the entire stack all at once would prove absolutely impossible to test and debug.

The message encoding layers (the ones responsible for encoding XMPP messages for writing to the network stream) are selectively inserted into the protocol stack only if the server announces support for it. It was tricky to design all of the different pieces to be extremely abstract from each other. If each of the layers were tightly coupled to each other, then testing would difficult and many of these protocol layers could be optionally inserted into the stack requiring them to have no knowledge of what specific layer is below or above them. The protocol layers expose different interfaces to the layer above. For example, the compression, encryption, and transport layers expose only a byte-oriented stream since they only know how to transform a

series of bytes. This is different than the higher-level protocol layers which don't know anything about what the underlying network protocol looks like and instead only know about the abstract concept of a message. This complete separation between messages and the underlying network encoding format allows me to implement custom network encoding schemes that may be more efficient or might even use a different network transport protocol like reliable UDP.

```
[...]
MSG1: <challenge
xmlns="urn:ietf:params:xml:ns:xmpp-sasl">[...]</challenge>
MSG2: <iq xmlns=”jabber:client” type="result">[...]</bind>
[...]
```

**Figure 2: Two sample XMPP-layer messages that might be sent or received**

Each XMPP message type that can be sent or received from the server is represented by a unique class that is capable of reading and writing that object. I separated the concern of understanding what a message looks like into the object that knows how do it best, itself. This helps keep related code together and ensures that new message types can be read or sent. In Figure 2 above, I show two types of messages that my client could receive from the server. I built a main message loop that is responsible for reading messages from the wire blocking until it receives a message from the server.

A message is marked by the start of a new XML element. When it receive a new message from the server, the thread will first read the message type and it's XML namespace. XML namespaces are used throughout XMPP to prevent name conflicts and to allow for extensibility. The message type, in this case might be `‘challenge’` or `‘iq’`, combined with the XML namespace, `“urn:ietf:params:xml:ns:xmpp-sasl”` or `“jabber:client”`, uniquely identify a specific message type. This must be globally

unique or the behavior that an endpoint that receives an ambiguous message is undefined. Once I have these two strings, I look up the correct message reader, then the main message loop hands control of the network stream over to the message reader. Only one thing is allowed to read the network stream at any given time. I chose to enforce this constraint because below the XMPP-layer, the transport medium is not multiplexed so allowing multiple readers in at the same time would not work. This also promotes power consumption efficiency by reducing the number of threads doing work at any given time, which is a great benefit when working on power constrained devices such as mobile phones.

## Other libraries

I ended up integrating two third-party libraries into my code base to help speed up prototyping and development. The first was an implementation of a DNS resolver in Java, dns4java. As recommended by the XMPP specification, to correctly perform service endpoint discovery I needed the ability to resolve SRV records for any given domain name. This was not supported by any of the built-in libraries in the Android SDK, as they only provide a way to resolve A or AAAA records for a domain name, which was not what I needed. Instead, I did a little research on how much work would be required to resolve them by myself. I discovered to build a fully standards-compliant DNS resolver would be almost an entire senior project by itself if I were to fully implement all of the security and traversal patterns. Since my senior project was oriented around building an XMPP client, not a DNS client, I chose to integrate this library into my program.

Another library that I integrated was used for the stream compression feature. One optional XMPP specification defines the ability to compress the entire network stream to save on bytes sent and received. It specifies that one compression algorithm that must be supported is the Zlib compression algorithm. I chose not to implement the compression algorithm itself at the time since it was quite complex. While the library did

indeed implement the Zlib compression algorithm itself, it didn't expose it in a way that I could integrate into my stack easily, so I had to modify the code a bit to ensure that it could operate on my streams. The library also was not written in a way that I could dynamically apply to an ongoing network communication stream and would block the thread until it received any communication from the server. This caused a dead-lock because the server was waiting for the client to send the compression handshake to the server. I solved this by modifying the library to lazily initialize the compression state tables when it did receive something from the server.

## Problems

I encountered a few different problems while working on this system. Many of these stemmed from small differences in how Android works compared to how the standard Java framework is implemented. I had originally wanted to implement the network stack in a way that was completely platform independent which would allow me to run my unit tests on Windows and know that they'd work on Android. This was quite useful for when I was first getting the stack to a point where it could be used. One thing I realized as I started to run it on our target platform is that android did not follow many of the same conventions that the standard Java framework followed.

One example of this would be the XML based stream readers and writers classes. In the standard XMPP encoding, all messages are encoded as a XML and must be streamed to and from the lower levels due to the the protocol. Document-oriented XML readers and writers won't work since we won't have the entire document to read in at once and we also depend on the context of different XML namespaces. I originally wrote my message writers to use the XMLStreamReader and XMLStreamWriter classes. When I went to run the test code that I had written on Android, I realized that none of these classes were implemented in the Android Java framework which meant that could not use the same classes on both desktop and Android platforms. Instead of reusing the classes already in the Java framework, Android decided to implement custom classes that were not compatible. These classes

also did not support all of the standard features. To account for this, I then abstracted out the XML reader and writer classes with a small facade object that would switch between the two classes based on the local execution environment. Unfortunately this solution did not help since the Android XML writer did not fully follow all of the XML serialization rules that the desktop followed but instead took many short cuts to allow their code to be smaller and force my code to be more complicated instead.

```
<stream:stream xmlns='jabber:client' ...
    <message from='romeo@example.com' to='juliet@example.com'
xml:lang='en'>
        <body>Neither, fair saint, if either thee
dislike.</body>
    </message>
```

**Figure 3: A sample expected output that is generated by the desktop Java framework and as implemented by the XMPP protocol specification**

```
<f0:stream ...
    <f1:message xmlns:f1='jabber:client'
f1:from='romeo@example.com' f1:to='juliet@example.com'
xml:lang='en'>
        <f2:body xmlns:f2='jabber:client'>Neither, fair saint,
if either thee dislike.</f2:body>
    </f1:message>
```

**Figure 4: A sample output generated by the Android based stream XML writers**

In figure 3, we have an example message that would be generated by the desktop framework. In this example, the stream element was generated before the message element and it defines a default XML namespace of ‘jabber:client’ which the message element will then inherit and use. This reduces network bandwidth required which is beneficial on mobile networks. In figure 4, what happens is that the Android XML writer will generate explicit XML namespaces on every single element and attribute which actually caused a few XML readers that I was using for testing purposes to choke and fail to parse the XML due to all of the extraneous XML commands.

This was not the only problem that was caused by the XML serialization libraries. The XML stream reader in Android was implemented in a way that caused slight differences in how elements were visited. In the desktop implementation, the element iterator would explicitly visit elements that had plain text in them and present that as a distinct XML text node, but in Android this was not the case and the parent XML element node would contain the body text. This meant that I had to break cross-platform compatibility for the time being to get it to work on the Android platform.

Platform incompatibilities were only one of the many difficulties I had to overcome while working on this project. Working with encrypted and compressed network streams made debugging issues difficult. Normally I would use packet capturing tools, such as Wireshark to monitor that everything that my protocol writers wrote to the wire was compliant with the protocol, but when I started implementing the encryption and compression features, I was unable to passively monitor these communications. Being able to passively decrypt the data would defeat the purpose of encryption, which is not something I wanted. This forced me to implement and use tools that would transparently intercept the connections between my client and XMPP servers to log everything, then re-encrypt it and send it off to the server. This only worked when my client connected to servers that I had full-control over because I could use a valid SSL certificate on the interception tool that would pass certificate validation on the

client, but wouldn't work on services that I did not control and did not have access to the SSL certificates for, such as third-party XMPP networks like Gmail. Transparently capturing traffic on a different host allowed me to fully test my code in a way that would not impact the program.

## Conclusion

Overall, the project has been a success and it served as a great environment for me to develop a serious Android application which could have a continued future. Scala was also a very useful language to learn as it's a language that is becoming more and more popular in certain areas, such as machine learning with the Apache Spark framework, which is something that I plan to do more of in the future. We also have plans to make further improvements to this project and application. While the work that we've gotten done for the senior project involved a lot of coding, it only laid the groundwork for future work. The networking subsystem was designed in a way that's fully extensible by allowing new message types and handlers to be registered and inserted into the chain. This means that we can implement even more of the XMPP standard which will allow a fully featured application to be completed.