An Investigation into Infrastructure for Group Chat on Mobile Devices

# Student:

Ho Yeung Chin

# Supervisor:

Ian Warren

# Host Unit:

Department of Computer Science

# Degree:

BE (Hons) Software Engineering

# Career Development

This scholarship has provided an opportunity for me to apply some of the skills and knowledge that I have gained from various courses at university to put them to use in a project with a practical outcome.

This scholarship has also provided me the opportunity to acquire many skills that will no doubt be extremely valuable in the future.

I have had the opportunity to familiarise myself with the basic concepts of both developing a simple server as well as developing a network capable application for the Android smartphone platform. To do so, I also had to understand and utilise the basic concepts of multi-threading. I have also had the chance to use Java's high level Executors API to handle threading. These skills will be essential in any future career endeavours, especially as the usage of smartphones becomes ubiquitous, and applications which do not require network capabilities become increasingly rare.

Also, I have gained valuable experience in working with third party libraries, both those in available as part of the Android Development Kit as well as those in the open source project, JGroups.

Finally, I have been introduced to the importance of gathering empirical data to assess an application's performance, and not just whether the application functions correctly, which has been the focus in my studies so far.

# Summary

With the rapid increase in smartphone users in recent years, the demand for applications that allow a large amount of users to logon from their mobile devices and participate in a group conversation has grown. Currently, there is a huge variety of group chat applications available for smartphone platforms.

Often, it is not feasible for there to be only a single server handling such an application. A distributed server architecture provides advantages in both reliability and performance, especially if the number of users is large. However, a distributed server architecture can present issues not necessarily present with single server; one of these issues is guaranteeing message ordering between servers.

None of the current group chat applications available for smartphones advertise message ordering as a feature; The main objective of this project is to deliver a group chat application with a distributed server architecture that provides message ordering guarantees for the Android smartphone platform. Currently, it allows the for the transmission of both text and audio messages across the multiple clients attached to multiple servers, using both Wi-Fi and mobile data networks.

# Abstract

Group chat applications have become increasingly common on mobile devices. However, for the Android market, there is a lack of apps that provide message ordering guarantees. This project aims to deliver a group chat application with a distributed server architecture that provides message ordering guarantees for the Android smartphone platform. The open source project, JGroups, has been used to provide message ordering guarantees between servers. JGroups also handles the formation of the server clusters and multicasting messages to all cluster members. An Android client has been developed to leverage this architecture, and is capable of sending text messages, as well as uploading recorded audio. The server can then distribute the audio file with other servers within the cluster and notify their connected clients, who can then either download or stream that audio file. Google Cloud Messaging (GCM) has been used for push notifications to notify Android clients of new content on the server. Performance tests using Wi-Fi to connect to the server showed that the average latency time before play using streaming was 2374 ms while the average latency for downloading was 1128 ms. Over mobile data networks, the average latency time before play using streaming was 5431 ms while the average latency for downloading was 4466 ms. The average time it takes for a push notification via GCM to arrive to a client is 967 ms over Wi-Fi and 3140 ms over mobile data networks.

# 1. Aims

The main objective of the project was to develop a group chat application system with a distributed server architecture, along with a client application for the Android smartphone platform to utilise this architecture. Before development began, an investigation into existing mobile group chat applications was undertaken to determine the a set of standard capabilities common to all mobile group chat applications. In addition to these features, the system would need to provide a message delivery and ordering guarantee (e.g. FIFO or Total Order) across distributed servers. The system would ideally also be able to handle multimedia, with a focus on broadcast of recorded audio from one mobile device to many other devices.

As part of the development process, performance testing was needed to determine typical latency times as well as bandwidth usage, especially over mobile data networks.

# 2. Methods

## 2.1 Investigation into existing group chat applications

An investigation of apps available on the Google Play Store revealed no apps that featured message ordering guarantees. Two other applications, not on the Google Play Store, were investigated: ChatHoc, an app developed by Jonas Michel and Kyle Petre [[1](#_ENREF_1)], and an app developed by Jonas Adahl [[2](#_ENREF_2)]. While the end goal of these apps is similar to the goals of this project (that is, to develop an application that can guarantee message delivery order between mobile clients), both of these apps have the clients communicate directly with each other, unlike this project, which would have the clients communicate through a distributed server system.

## 2.2 Server Infrastructure

The open source project, JGroups [[3](#_ENREF_3)], was used to handle communications between servers. JGroups provides the message delivery and ordering guarantees between servers. Each member of the server group has a JChannel instance through which communications with other servers in the same cluster takes place. A JChannel can be configured with a protocol stack, with which the degree of message ordering required can be set, as well as various other network settings, such as whether the JChannel communicates through UDP or TCP.

The server group that the JChannel connects to can be set in the "config.ini" file located in the root folder of the server application. The protocol stack, defined in an XML file, can also be set in this file. There are preset protocol stacks already defined within the JGroups library (for example, "udp.xml" for FIFO ordered UDP multicast or "sequencer.xml" for Total Ordered UDP multicast).

An inter-server message is defined by the Message class in the JGroups library; the message will contain a list of destination servers, and can contain a Java object. Within this implementation, the object contained in each Message object is a HashMap<String, Object> with the string acting as a header to identify the Object in its paired value once the message has been received by another server.

## 2.3 Client-Server Communication Protocol

Communications between the client and server are performed directly via sockets; the OutputStream of each socket is always wrapped in a BufferedOutputStream first so a 1kB header can be written to the OutputStream identifying the type of request. Any additional wrappers, for example, ObjectOutputStreams, can be used on the already wrapped OutputStream provided the server unwraps the request in the appropriate order.

ObjectOutputStreams have been used when dealing with the transfer of structured data (e.g. ArrayLists of messages). This made it easier to develop both the server and client applications compared with a custom implementation using only the underlying BufferedOutputStream.

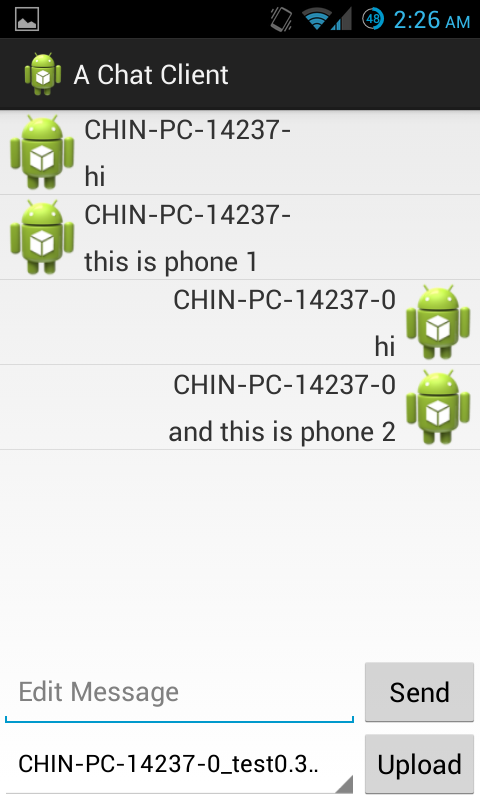
Each outgoing request from the client application to the server subclasses the ToServer class, with a corresponding class implementing the Command interface in the server application. Each request will first send a header of size 1kB describing the type of request before any other additional information is written. The header consists of the bytes taken from a string using String.getBytes() defining the type of request being made. All headers will contain at least two pieces of information: First, a timestamp taken using System.currentTimeMillis(), followed by a ":" then a string constant common to both the server application and the client application describing the type of request. Additional information can be attached, separated by ":". Any subclasses of ToServer must construct their own header string.

Google Cloud Messaging (GCM) has been used to implement push notifications to indicate whenever a message or file has been uploaded to the server a client is connected to. New messages and files are not delivered directly to the client using GCM, as GCM provides no ordering or message delivery guarantees, and has a maximum payload size of only 4kB. The GCM push notification instead contains information regarding what type of content has just been received by the server, and the client can then respond accordingly.

The GCM registration ID that each instance of the client application receives when it is first used on a phone is also used to identify each user on the server, as the GCM ID is unique to each particular instance of the client application on a particular phone.

Two methods have been explored for the broadcast of audio data uploaded by a client and assessed to determine which produces lower latency; both streaming the audio from the server and downloading the audio file to the phone and playing it directly.

## 2.4 Android Client Application

Currently, only a placeholder layout has been used, as shown in Figure 1 to the left:

The messages list is a ListView with two different row layouts defined in XML, one layout to display messages from the user and one layout to display messages to the user. A custom ArrayAdapter, MessageAdapter, is used determine which layout is used depending on who the author of the message is.

ListView for displaying messages with two different layouts

Messages are retrieved in ArrayLists from the server; once notified, the client sends a request to download messages from the server, which contains the current that the client has. The server returns any new messages after that index.

Spinner to select recorded audio file to upload

Audio recording is performed by a concrete subclass of the RecordAudio class. In the current implementation, the RecordMediaAudio class uses the built-in Android MediaPlayer to record data to a .3gp file. Streaming and playback of downloaded audio is handled similarly to recording audio. Streaming using MediaPlayer sends an HTTP request to the server, and the server has to respond with the appropriate HTTP headers before writing the audio data back over a BufferedOutputStream.

Figure : Current Layout of Android App

Message ordering for outgoing messages from a client have been ensured by making all outgoing sends execute on a SingleThreadExecutor and having each outgoing message block until the server confirms that it has received the message.

# 3. Results

## 3.1 Overhead of FIFO ordering versus Total Ordering

See Figure 2 below for server setup. Messages were text messages containing a number from 0 to 29.

Table : Average round trip delivery times in milliseconds, number of repeats in brackets

|  |  |  |
| --- | --- | --- |
|  | Total Order | FIFO |
| Phone A to Phone B | 543 (25) | 553 (26) |
| Phone B to Phone A | 509 (26) | 507 (26) |

## 3.2 Latency in Downloading versus Streaming

Refer to Figure 3 below for server setup and Figure 4 for recorded latency times. The latency time is measured from when one phone begins the upload to when the other phone is prepared to play the audio file.

Test was conducted with a 10 second voice recording with a file size of 20.9kB. Results showed downloading the file in its entirety before preparing to play was faster than streaming the file, both over Wi-Fi and mobile data networks. The average Wi-Fi download delay was determined after 30 attempts, average Wi-Fi streaming after 29 attempts. Over mobile data networks, the average download was determined after 30 attempts while the average streaming delay was determined after 27 attempts.

## 3.3 GCM Notification Delay

The time it takes for a GCM notification to arrive to the phone after it was first constructed on the server is highly variable, with an average time of 967 ms over Wi-Fi from 59 notifications and 3140 ms over mobile network from 60 notifications.

## 3.4 User Testing

A test chat with two devices containing 71 seconds of audio, performed over mobile data networks used around 400kB of data on each device. General impressions were that the length of the audio message has a much larger impact on the flow of a conversation than the latency involved in getting the content from the server to the device.

# 4. Discussion

While testing showed that there was no significant difference between the delivery times for a server cluster configured to use FIFO compared with one configured to use Total Ordering, further experimentation using a larger number of servers in the cluster would be preferable, as the overhead might not be noticeable if there are only two servers.

The lower latency experienced when downloading audio files compared with streaming was not expected; One would expect that streaming would experience lower latencies than downloading if a larger file was tested. However, a larger sound file would likely not be feasible for live chat situations, given the delay required to make the sound file.

The time it takes for a GCM notification to arrive takes up a significant portion of latency for the delivery of audio messages. Delays of 3.14 seconds over mobile networks would be unsuitable for a live-chat application. Another drawback currently is the lack of capability of live transfer of audio data between servers; only completed audio files can be transferred, which prevents a live broadcast between clients.

# Figures

Note: In Figures 2 and 3, Phone A is a Samsung Galaxy S2 running stock 4.0.3 software while Phone B is a Samsung Galaxy S2 running CyanogenMod 10.1 (4.2.1 equivalent).

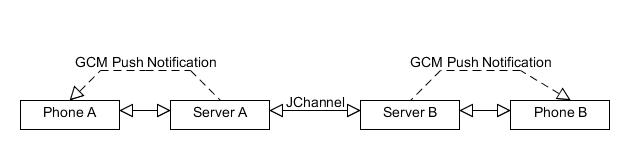


Figure : Overhead of message ordering testing server setup. One phone is connected to each server via Wi-Fi.

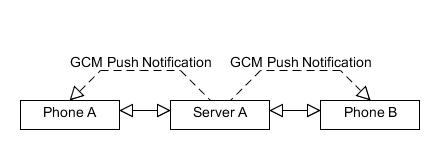


Figure : Latency testing server setup. Both phones are connected to the same server through either Wi-Fi or 3G.

Figure : Average Latency time for a 10 second audio file before play

# References

1. Michel, J. and K. Prete, *SkeenZone: A distributed Android chat application and extensible middleware.*

2. Ådahl, J., *Shared resource for collaborative editing over a wireless network.* 2011.

3. Ban, B., *JGroups, a toolkit for reliable multicast communication.* URL: http://www.jgroups.org, 2002.