# Google Cloud Messaging

I suggest following Google's tutorials to get this working if you don't already know how:

http://developer.android.com/google/gcm/index.html

You'll probably want to replace the APIKEY constant in the Constants class on the ChatServer and the SENDER\_ID constant in the Constants class on the ChatClient with your own values.

# ChatServer

## Running

For the server to run, 4 libraries must be present and on the classpath:

* The GCM server library, which can be downloaded through the Android SDK Manager under extras. The appropriate jar can be found at sdk/extras/google/gcm/gcm-server/dist/gcm-server.jar, wherever you have installed the Android SDK.
* the JGroups library, e.g. jgroups-3.2.7.Final.jar
* JSON Simple, e.g. json-simple-1.1.1.jar
* Library containing the com.hchi590.chatclient.Message class, e.g. message.jar.

For JGroups multicasting to work, IPv6 has to be disabled for Java. This can be done by setting this option on the command line or in the Eclipse project run settings.

The option would be:

"-Djava.net.preferIPv4Stack=true"

Also, the port that the server listens for mobile clients on is given as a command line argument. The main method is located in the hchi590.server.Launcher class. An example run command that starts a server listening for mobile clients on port 31000 would be:

java -Djava.net.preferIPv4Stack=true -cp jgroups-3.2.1.Final.jar;gcm-server.jar;json-simple-1.1.1.jar;message.jar; hchi590.server.Launcher 31000

The config.ini file is read to determine the cluster that the server joins with/creates, as well as which protocol stack will be used by the JChannel instance.

The .xml file to load the protocol stack from is specified by writing "config:" before the file and the cluster name is specified by writing "cluster:" before the cluster name. Refer to the JGroups documentation for more information in writing the .xml files to specify the protocol stack.

## Data Storage

Data storage is handled through the DataStorageSingleton class. You'll probably want to replace the data structures here with something persistent.

## Push Notifications

The PushNotification will generate a GCM multicast and add data given a String array. Data from the String array is added with the keys "data0", "data1", ... and so on until the end of the array. The push notification will also add the current time with the key "time" to the message. data0 is always a String constant describing the sort of data the push notification is supposed to inform the client of; e.g. POST for a new message having arrived on the server.

I would suggest running the PushNotification class on its own thread as performing a send can take a rather long time.

## ChatHandler

ChatHandler is a singleton class that defines all communications between servers. This class extends the org.jgroups.ReceiverAdapter class, which importantly has the receive() callback which runs when a message is received. It also defines methods for getState and setState which allow a server to transfer over the current state (for example, messages and online client list) of the server if requested.

## Sending Messages between servers

All messages between servers use the class org.jgroups.Message. Each message has a payload which can be a byte array or a Java object; in ChatServer, it is a HashMap<String, Object>. Data is stored in the HashMap along with a String constant value with the "HEADER" key (String constant) that identifies the type of data contained in each message to the receiver. The receive callback will then process the message according to which header is contained in the HashMap.

By default messages will be sent to all members of the cluster, but specific servers can be selected as destinations either in the constructor itself or presumably through setAddress() (not tested!).

## Handling incoming communications from mobile clients

Incoming communications will spawn a new ChatServerThread, which reads the 1kB header defined in the ToServer class using a BufferedInputStream, then converts that header to a string. The ChatServerProtocol class will then return the appropriate command based on the header. Commands all implement the Command interface, and each one corresponds to a different request coming from the client.

# ChatClient:

## Installation

Install as per usual via Eclipse or whichever preferred method.

## Running

### G:\Summer Scholarship\Screenshot_2013-03-02-03-21-52.pngG:\Summer Scholarship\Screenshot_2013-02-27-02-25-43.pngUsage

Have the ChatServer and NameServer running first before starting ChatClient. (Not entirely necessary, but makes the startup smoother.)

Spinner to select recorded audio for upload

ListView with custom ArrayAdapter

Figure : Main screen

Figure : Retry connecting to server... dialog

The send button and upload buttons will enable once the client has made a successful connection to a ChatServer (the first connection is always an attempt to register the GCM ID with the ChatServer).

Bringing up the menu with the menu button will bring up options to bring up four options:

1. Send thirty messages - just sends the numbers 0-29, one number per message to the server. For testing purposes only.
2. Retry connecting to server... - brings up a dialog where a server hostname and a port number to connect to can be set. Will send a request to register the GCM ID upon pressing OK.
3. Start Recording - starts recording from the microphone. Pressing again stops recording. The menu text changes to "Stop Recording" as well and changes back if pressed again.
4. Set to download mode - sets the client to download audio files completely and play them once notification is received of a new audio file. By default, the client will attempt to stream the file instead. Pressing again will switch back to streaming mode.

The spinner underneath the entry field for text messages contains a list of the audio files that have been recorded thus far; pressing the "Upload" button will upload the selected audio file.

### Getting the server

Upon startup, the client will attempt to contact a name server (NameServer) with a fixed hostname to retrieve a ChatServer address to connect to. This action is defined in the GetServer class. The GetServer command expects a String written over an ObjectOutputStream back from the name server, with which it sets the address of the ServerAddress singleton instance.

The current implementation just blocks the main thread with wait() for up to 5 seconds or until it has successfully retrieved an address from the server, so a smarter implementation or some sort of loading animation would be preferred.

### ToServer

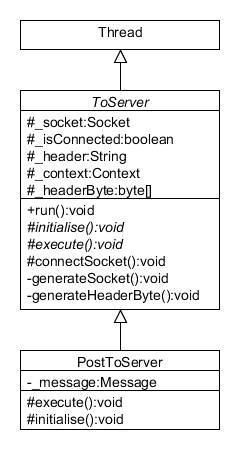


Figure : ToServer class with concrete subclass PostToServer shown.

The ToServer class serves as the base class to all communications to a ChatServer; It will generate the header that the ChatServer will use to recognise the type of request being sent by the client. It also instantiates the socket to use in generateSocket() as well as starting the connection to the server. It also defines what happens when a connection is unsuccessful in the connectSocket() method. Currently, an attempt at a connection will be made with a 5 second timeout before throwing an IOException. This class will then send a broadcast to the main Activity, so it can respond appropriately to the failure. Currently, the main activity will display a Toast indicating a failure to connect.

A subclass must implement the initialise() method, where the subclass will append the type of request to the \_header field. The \_header field always contains a timestamp, taken in the constructor of ToServer, and the type of request being made. This will be a string constant defined in the Constants class. The ChatServer will be expected to use the same string constant to identify the header was it receives it.

A subclass must also implement the execute() method, where data is written to and received from the ChatServer. The OutputStream of the \_socket field must always be wrapped by a BufferedOutputStream, and the first thing that is written is always the \_headerByte field (a 1 kB byte array generated from the \_header field using the .getBytes() method in generateHeaderByte()). Once the \_headerByte is written and ideally flushed, any additional data can be written/input from server read.

The outgoing communication to the ChatServer which does not subclass ToServer is when an audio file is requested to be streamed; the actual connection and request is handled by the Android Media Player, which sends an HTTP request to the server instead of following the format outlined above.

## Handling GCM notifications

The GCMIntentService class has callbacks for GCM events;

onMessage(Context context, Intent intent)

is the callback called when a GCM message received by the phone. intent contains the payload of the message and context refers to the application. Upon receiving a message, data is extracted from the payload of the GCM message and is stored in a new Intent, which is then broadcast. A BroadcastReceiver, messageReceiver, in ChatActivity will then receive the intent and decide what to do from there.

## Messages

Message data is stored in the Message class which has fields for text, author and time. In the UI, messages are displayed in a ListView.

### MessageAdapter

MessageAdapter is a custom ArrayAdapter that allows messages to take different layouts depending on who the author of the message is. It also sets the data that appears in the TextViews of the layouts. Both getViewTypeCount() and getItemViewType() methods should be overriden from ArrayAdapter otherwise the layouts returned for the ListView end up being randomly chosen.

There are two layouts that appear in the ListView: row\_other.xml and row\_self.xml. Essentially, they are both just RelativeLayouts with the alignments flipped to differentiate if the message came a different client versus a message that came from that particular client.

### Message Retrieval

Once the server has received a message, it will send a push notification to all of the clients that have their IDs registered to it, with the index of the message contained in the payload. Push notifications indicating a new message have headers using the String constant "POST".

1. Upon receiving a push notification with the header "POST", a new retrieve request is launched (the RetrieveMessageToServer class handles this).
2. This will send a request, along with the index of the last message the client has seen, to the server to retrieve any messages after the current message (that is, messages with a larger index) the client currently has.
3. The server determines that it is a retrieve request by reading the header of the request. ChatServerProtocol will return the appropriate command, RetrieveCommand, to handle the retrieval of messages.
4. RetrieveCommand writes back an ArrayList of hchi590.chatclient.Message over an ObjectOutputStream.
5. RetrieveToServer reads this ArrayList, and adds this information to an Intent.
6. This Intent is then broadcast to the application context, where the BroadcastReceiver, messageUpdateReceiver, retrieves the messages from the Intent and adds them to the message list and notifies the corresponding MessageAdapter that the content of the list has changed in order to update the UI.

Retrieve requests occur on a SingleThreadExecutor. Requests are only added to the executor if the client is not currently retrieving in an attempt to minimise the number of retrieve attempts the client makes.

## Audio

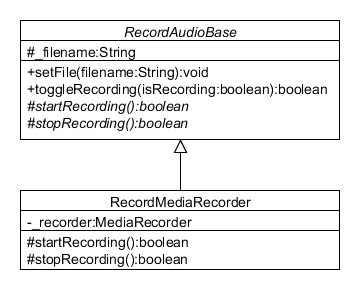
Audio handling is provided by classes in the com.hchi590.chatclient.audio package. A base class has been provided for recording audio (RecordAudioBase) and playing audio (PlayAudioBase). The base classes provide the interface through which ChatActivity can interact with the audio APIs, so one should be able to replace the currently implemented subclass if need be. 

Figure : Class hierachy for audio recording classes

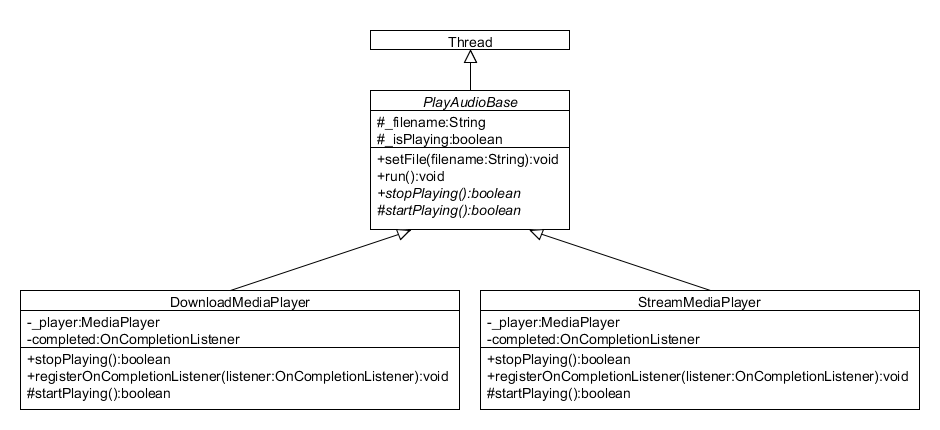


Figure : Class hierachy for audio playback classes