Raspberry Torrent

CS 401 Final Project

Jaleesa Dmytrow, Ashley Herman, and Rooshan Aslam
https://github.com/rooshan1234/Project-JAR
4/10/2016

Table of Contents

Introduction

This paper is used to summarize and describe our Computer Science 401 final project. Within this paper we will talk about our original project ideas, overview of torrenting, overview of Raspberry Pi, and the overview of Google Cloud Messaging. We will also talk about the constraints with each of these components of our project. This paper also includes each team member's contributions. Lastly, we will explain in detail, how our project was set up and problems we ran into while setting up the project. The conclusion of this paper will summarize all topics covered.

Thought process – first ideas

When we first discussed the project we came up with a couple of ideas. One of our ideas was creating a key logger from an Arduino, connected to a computer (a Raspberry Pi) and analyzing the results within a cloud platform such as keen.io. When we started to look at this project we realized that this idea would not be enough work for a group of three people. Another problem we came across with creating a key logger was the legality of creating a device that records key strokes of a user.

The next idea that we came up with was to create a torrent box out of a Raspberry Pi that is controlled by a phone. With this idea we had to figure out how the phone would connect with the torrent box which is where Google Cloud Messaging (GCM) is introduced as the connecting link. We also thought about adding keen.io, as a future development, to analyze the data for the torrent box. Because this project has a lot of components we felt that it was large enough for a group of three and would allow us to customize and split up the work. When trying to look for a similar project to our idea we found that a Raspberry Pi, GCM and a cloud have not been connected to a torrent box before. Therefore, this project idea is unique.

Other projects similar to this idea

Although our project is unique, we looked at projects that created a torrent box out of a Raspberry Pi for similarities. One of the projects that is similar to our group project was a torrent box created on a Raspberry Pi that is accessed via a web

browser, using the Pi's IP address [3]. To create the torrent box on a Raspberry Pi the project installed Raspbian as the operating system and downloaded the transmission daemon to allow for the torrent box to work. Another project that created a torrent box out of a Raspberry Pi was created in a similar way as the first project but used a different daemon to provide the access to the torrent box. This other project also allowed a user of the torrent box to be accessed by windows explorer. To provide this option the Pi and the computer that would like to access the Pi would have to be on the same network or connectivity issues could arise. Once we looked at these two projects we realized that our torrent box project would be much more complex than a simple Raspberry Pi torrent box.

History of Torrents

Torrents are shared using the peer to peer (P2P) protocol. Within a peer-to-peer network many peers (or end hosts) are connected to other peers (or end hosts) [1]. File sharing uses a P2P network as the users are sharing files with each other. A P2P file sharing network is highly scalable as a new peer can join the network without crashing the network [1]. The new peer just adds a connection within the network, there is no reconfiguring or re programming. P2P networks are faster than server to client networks because the server is doing all of the work; when a client requests a file the server has to provide the file. If there is more than one client asking for the file the server is very busy which could cause some back log [1]. In a P2P network when one peer asks for a file the other peers in the network can provide the pieces of the file they have which speeds up the download [1]. The idea of torrenting is based on file sharing. The earliest version of file sharing is using the Bulletin Board System created by Ward Christensen [4].

One of the most popular peer to peer protocols for file distribution is BitTorrent created by Bram Cohen [1]. BitTorrent allows a peer to download chunks of the file they require to form the entire file from other peers instead of a file server [1]. A typical size for a chunk of data is 256 Kbytes [1]. A peer that uses BitTorrent has the option of getting the file they want from the P2P network and then leaving or staying in the

network. If a peer leaves the torrent they just downloaded will not become a seed from their computer. On the other hand, if the peer stays within the network chunks of the file they downloaded will be uploaded for other people to download from.

BitTorrent has many complicated pieces involved to allow for people all over the world to become a peer and upload chunks of files. This P2P protocol network only works because it is a tit-for-tat system which means that when you download files you are also uploading the chunks of files you already have to benefit someone else in the network [me 1]. If BitTorrent was not a tit-for-tat system there would be many people that just downloaded the file then left the system. Although this does happen there are enough peers that stay within the network to keep BitTorrent alive.

Raspberry Pi overview

The unique property of a Raspberry Pi is that the whole system is on one chip. This means that everything a computer will need to run is on one chip, you will not need to connect memory, a CPU, or a GPU. The first Raspberry Pi was created by the Raspberry Pi Foundation, based in the United Kingdom. The main reason for creating the Raspberry Pi was to promote teaching of basic Computer Science within schools and developing countries. Raspberry Pi's has made this task easier as they are affordable, easy to use, and are portable to an extent. The Foundation that created the Pi provides Debian and Arch Linux ARM distributions for download but is not the only distributions that can be used with the Raspberry Pi. The Foundation also promotes Python as the main programming language to be used with the PI but as with the distributions, Python is not the only option. Raspberry Pis are popular and successful because the cost is considerably less than a home computer that has separate parts. For example, a regular home computer will need the bare necessities such a CPU, RAM, motherboard, and a GPU. All of these separate parts for a computer each cost money. On the other hand, a Raspberry Pi is a system on a chip; the Pi has everything it needs on one board. The lower price for a Pi is beneficial for students or developers that may want to try something new but do not want to wreck their current computer.

The Raspberry Pi is an embedded system that has been improved with new features since the first model was released [7]. There have been many improvements since the first Raspberry Pi. When comparing the Raspberry Pi 2 to older models the boot up time is much faster (Raspberry Pi 2 has the boot up time of 20.26 seconds compared to 35.72 seconds for Model B+ and 34 seconds to Model A+) [2]. With Raspberry Pi 2, Raspbian has been updated to be more lightweight, load applications more quickly and allow for less resources to be used up [2].

Although a Raspberry Pi is a system on a chip that is amazingly powerful for a computer that costs so little, the chip does have some drawbacks [5]. One of the most noticeable drawbacks is the speed of the Pi compared to other devices. The Pi does allow a user to use it as a computer but the power and speed is closer to a tablet or a mobile device) than a regular home computer [6]. The one difference that shows the capability of the Pi compared to a mobile device is the graphics are sharper on the Pi [6]. Another drawback is the limited memory on the Pi. The memory is limited because of the amount of space on the chip. This limitation would restrict the capabilities of the Pi. Lastly, the Pi only has 1 GB of storage which does not allow for large data to be stored on the Pi directly, although an external drive could be connected to increase the storage.

GCM overview

Google cloud messaging or GCM for short is a service provided by Google to help decrease the discrepancies for messages passing between two devices connected to the Internet. Google cloud messaging server handles the queuing of the messages and delivery to the client applications. For our purposes, we set up a server that will communicate with an android device. There are three required components that are needed to use GCM: application server, GCM connection server, and client application. It is important to mention the following terms can be used interchangeably for our project: ‘Raspberry Pi’ can interchangeably be used with ‘application server’ because the Raspberry Pi is running our XMPP server. Also, ‘client application’ can be used interchangeably with ‘android application’ because our client application is an app that

runs on the android OS. The application server is used to communicate with the GCM connection server via HTTP or XMPP. The GCM connection server receives messages from the application server and sends them to the client application (known as down streaming). The GCM connection server could also accept messages sent from the client application and send them back to the application server (known as upstreaming). Lastly, the client application equipped with GCM capabilities will send and receive messages from GCM directly. Without the three components listed above, GCM would not work (as shown in figure 1).

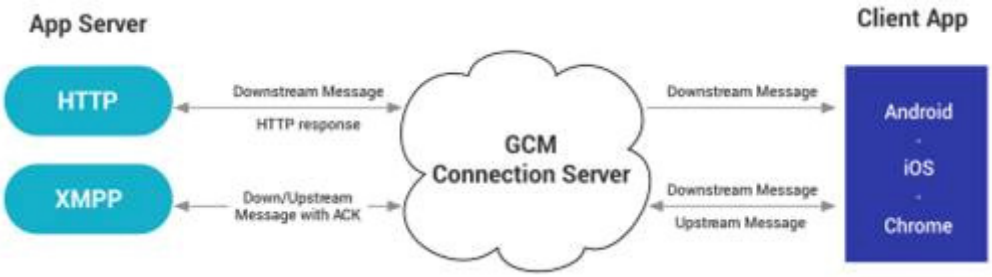

Figure 1

To communicate with GCM there are four credentials needed that are used for both the application server and the client application: sender ID, API key, application ID, and registration token. The sender ID is used to generate the registration token for each device that the client application is installed on. Each GCM enabled application has a unique sender ID so that the GCM connection server has a pool of the application clients under its domain registered to a single sender ID. The API key is used to talk from the application server to the GCM server by sending HTTP/XMPP requests.An application ID is the unique id which specifies to the application that GCM is enabled on. The application ID is generated based on the name of the application that will be deployed on the android operating system. Lastly, a registration token is used to talk to the specific device GCM wishes to communicate with and is used every time a message is sent from the application server to the client application. This token could be returned to the application server to be stored in a database to keep track of the devices and their corresponding users linked to their token, but this functionality is optional.

To send downstream messages through GCM servers to the user application, there are two different approaches: HTTP post request and XMPP. HTTP post request is what the application server uses to send notifications and/or messages to the client application. There are three different parts required in an HTTP post request: URL, header, and data XMPP (Extensible Messaging and Presence Protocol) on the other hand is a protocol, much like HTTP, but specifically designed for handling messages via XML (Extensible Markup Language). As we are using XMPP instead of HTTP post request this paper will not cover the code required for this approach. XMPP will be talked about in more detail in the upstream paragraph.

How XMPP relates to GCM

For our project we will be using Python XMPP. GCM uses XMPP to send messages from the application server to the user devices. XMPP also is used to send upstream messages from a user's device to the application server, which includes the acknowledgment messages. GCM servers run CSS (Cloud Connection Server) which is an XMPP endpoint. This XMPP endpoint provides persistent, asynchronous, bidirectional connection to Google's servers. This connection can then be used to send and receive messages between your application server to the GCM connected devices you have. CCS uses XMPP as an authenticated transport layer. Some libraries which implement this built in authenticated transport layer support include: libstrophe for C, Smack for Java and XMPP Python. To get CCS to work two additional parts, which differ from HTTP, are required. An initiated Transport Layer Security (TLS) connection that you as the developer must create. The second part that is needed is a Simple Authentication and Security Layer (SASL) which is done by providing your username as the sender ID and the API key as a password, which would be in the form <GCM sender ID>@gcm.googleapis.com. CCS XMPP endpoint runs at gcm.googleapis.com on port 5235. An example of XML code for a CCS XMPP is shown below.

XML code example:

```
<auth mechanism='PLAIN' xmlns='urn:ietf:params:xml:ns:xmpp-
sasl'>base64Enc(\0[userName]\0[password])</auth>
```

To send upstream messages from the client application to GCM servers the only viable option is XMPP because upstream functionality is not supported in HTTP post requests.

CONSTRAINTS

Constraints for Torrenting

Torrent machines hover on the legal/illegal line. A torrent machine that offers free software/updates that are already open source are legal as they are not promoting or allowing illegal downloads. However, a user should download from the original site of the file instead of a torrent machine. On the other side, torrent machines that allow for music or movies to be downloaded are illegal as these files should be purchased. Torrents were created to help share files to other peers. The majority of torrents are illegal as they break the copyright law. Torrents allow for peers to work together so that if another peer needs a file they will not have to slowly download it from a file server that may be overloaded.

Some constraints of a torrent application (for example BitTorrent) include:
If there are no users that are seeding a document there would be nothing to torrent. If the torrent application is disabled, no one can access the files even if another person on the network has the file. The torrent box connects everyone using the network. Therefore, if the torrent application (machine) is disabled so is the network.

Constraints for Raspberry Pi

A Raspberry Pi is a great tool for creating a device such as our project but there are some constraints associated with the Pi that need to be noted. One of the main constraints is the fact that the Pi needs to always be plugged in to a power source. In most cases this means using a wall plug unless you have a portable power source that is not directly connected to the wall such as a battery pack. For our project we need to use a wall plug-in as our power source as we do not have the resources to make a battery pack available.

Another constraint for the Raspberry Pi is that we needed to keep the Pi connected via Ethernet at all times. The solution to this constraint would be a Wi-Fi dongle which we were not able to use due to lack of resources. There is no onboard Wi-Fi so the only two options if you want to connect the Raspberry Pi to the internet are a Wi-Fi dongle or to have the chip always connected via Ethernet.

Lastly, the constraint of memory will cause problems when creating a device that needs to store more than the 1 GB of storage the Raspberry Pi has. One way to combat this constraint is to connect an external storage device, such as a USB thumb drive or an external hard drive to the Raspberry Pi. This solution is only possible if you have available resources.

Constraints for GCM

When talking about constraints in concern to GCM connection server a couple of main concerns come up. The maximum payload (or package) size is only 4 KB per message, this is for both sending and receiving messages. The maximum number of instantaneous downstream requests that GCM can process is limited to 1000.The maximum pending upstream messages is 100 and the maximum pending upstream messages from a single device is only 20 can be sitting in GCM. The longest a message with stay in GCM is 4 weeks, after this time the message dies. So if GCM is over loaded and the message is not taken care of fast enough the device will have to be resent. The connection server has to send ACK messages to the GCM endpoint when you are using XMPP, this causes extra messages to be sent.

Some of the constraints when using a phone (such as an android phone) includes phone resource usage, scheduling tasks, message failure and battery saving that are all handled with the time to live/priority settings in the JSON object that is sent to the GCM server. To help with the phone resource usage constraint GCM implements (for an android phone) battery saving, network optimization and screen on time features using GCM Network Manager. This feature of GCM is useful so that your phones resources are not being used constantly by GCM and can still be used for other tasks. If this GSM feature was not available the battery would be drained quite rapidly as the

phone would continuously be waiting for messages from GCM. Another constraint that GCM has regarding a connected phone is when to send messages or do a task. There are two options: schedule one-off or periodic tasks. Schedule one-off will do everything that GCM needs to do at once so that the phone is only busy for a brief amount of time and then is free for other tasks. Periodic tasks allow GCM to perform tasks or send messages in small chucks of time on a periodic basis to not overload the phone with messages for an amount of time. Although the periodic tasks option looks like the better options it would depend on how often GCM interrupts the phone to send a message. If this happens frequently the phone will be slower as GCM is using more resources than usual. Another constraint regarding a connected phone is GCM's automatic back-off and failure retry. This feature is used when the message sent has an error or is corrupt. Once a message is labelled corrupt GCM will withdraw the message and re-send a clean copy. The last constraint that is not a problem is that if a large amount of messages need to be sent, but do not need to be sent immediately, GCM could send these messages when the phone is charging. This would prevent the battery from being drained.

Constraints for the Project

This project does have a couple of constraints that could have hindered implementation. One of the constraints regarding a torrent box would be that if you use a regular server or home computer the amount of power consumption would be excessive. For example if you use a regular computer for a torrent box keeping the computer on all the time would cost approximately $200 per year [5]. To prevent this constraint from affecting our project we decided to use a Raspberry Pi which consumes less power than a regular computer, approximately $3 per year [5]. This is a considerable difference, which is why the Raspberry Pi is the best option for our project.

Another constraint that could affect our project is how to get the Pi to communicate with the phone. An android phone connects with GCM easily which is how we will connect our whole project. Connecting a Raspberry Pi to the android phone would be quite difficult to implement. Therefore, we connect the Pi to GCM and GCM to the android phone. By using GCM as a 'middle man' we get to take advantage of

GCM's features. The main feature is that GCM handles the message queuing which allows the phone to save power until GCM is active. GCM automatically handles the power management of the phone.

The Raspberry Pi torrent box can support multiple users. If a mobile device can connect to GCM with the android app, then it can send a magnet link to the torrent box. We can think of the transmission daemon itself as being multi-user, because we configured it so that it does not require authentication and will begin a torrent on any magnet link that is sent to it under the default username. There are a few constraints involved with transmission itself. The Pi needs to use a 32 bit version of transmission, which is not as well seeded as the 64-bit version. Also, because it runs as a background process, the daemon will not immediately notify the user when something goes wrong. A few times, we ran into an error where the server had dropped, which we solved by restarting transmission from the command line. The fact that it runs in the background means that it is easy to tell whether a torrent started successfully, and more difficult to tell whether it has finished successfully. Currently, the only thing a user can do when texting the Pi is start a torrent; there is no text command to stop the torrent or check its status (although given more time, these commands could be implemented).

Overview of our project

Our project is to create a torrent box out of a Raspberry Pi that communicates with GCM to send messages to an android phone. To create this project we set up a Raspberry Pi as the server side of our application. The Raspberry Pi is our torrent box. We also set up Google Cloud Messaging (GCM) to communicate with the Pi and the android phone. GCM is essentially the middle service that connects the 2 devices. The client application, which we have named Raspberry Torrent, is the application that the user will interact with. The following diagram (figure 2) is how our project looks overall with all pieces of the project.

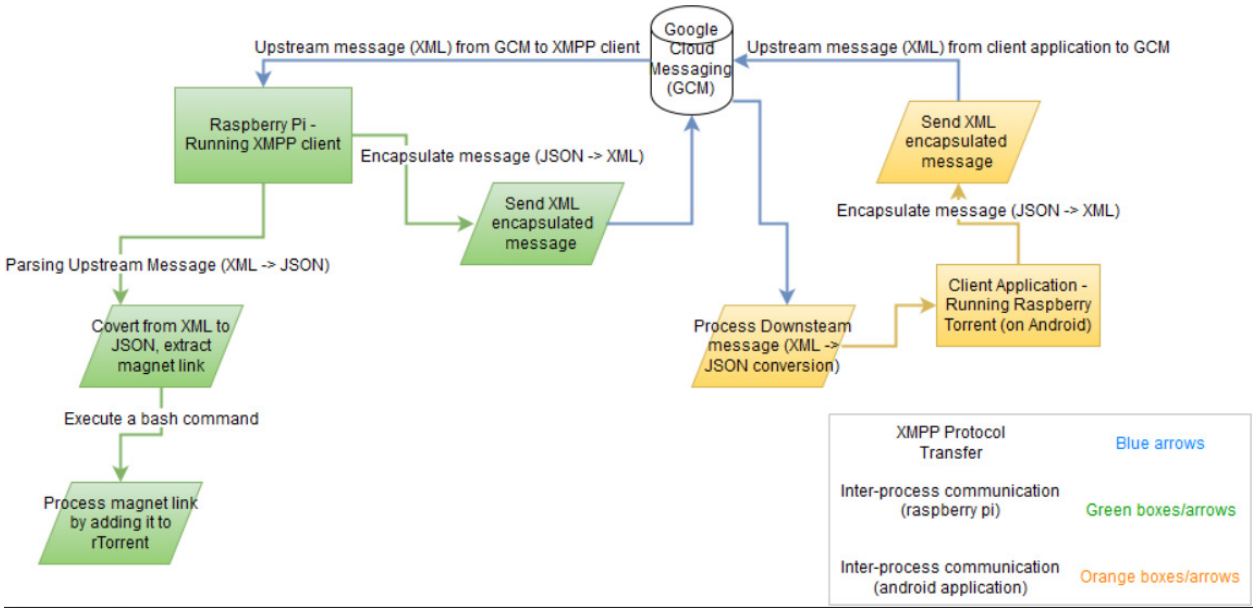

Figure 2

Setting up the Pi

For our project we are using a Raspberry Pi 2 model B. This model of the Pi has 1 GB of RAM, QUAD core broadcom CPU, 40 pin extended GPIO, 4 USB ports, HDMI, 4 pole stereo output and composite video port, CSI camera port and DSI display port.To be able to use configure the Pi we connected a USB mouse and keyboard to the Pi as well as an HTMI monitor and Ethernet cable. The Pi is also always plugged into a wall outlet as we do not have an alternate power source. To access the Raspberry Pi remotely we set up Weaved [10]. Weaved is an application that would allow us to work with the Pi by SSH protocol. Once everything was working, some of the peripheral devices could have been disconnected. However, we left them plugged in to periodically check that everything with the Pi was working correctly.

To set up the Raspberry Pi we used a 16 GB microSD card for more storage. The operating system we installed was Raspbian which was installed using Noobs. Noobs, or New-out-of-Box Software, is an operating install manager that we installed onto the microSD card by using Ashley's laptop. Noobs has Raspbian included in its installation, so installing the OS was very straightforward. Once the operating System, Raspbian, was installed we set up the torrent box using sudo-apt get transmission-daemon.

By using this command you get transmission CLI and transmission remote as well as all other files that are needed. Once everything was installed we changed the settings file, located at /var/lib/transmission-daemon/info/settings.json. This file contains all the default settings for transmission. Before editing this file, it is important to remember to stop the daemon so that the process does not overwrite the changes. In this file, we changed “auth-required” to false so that we didn’t need to use the credentials of the transmission for everything. To work with the transmission we created a bash script that passed a magnet link to start a torrent which is shown in figure 3. The bash script is called from a python script.

```
#!/bin/bash
## Note that the magnet link must have quotes around it.

echo "Running the bash script"

transmission-remote -a "$1" -s

echo "Bash script completed"
```

Figure 3

There are four of five components that are required to communicate with GCM, two components are application specific and the other two are universal constants. The two application specific components required are: USERNAME (Sender Id of the project), PASSWORD (API Key), and REGISTRATION_ID (Registration Token of the device). The two universal constant components required are: SERVER ('gcm.googleapis.com') and PORT (5235). A connection needs to be established to the GCM servers with the five required components. This connection is done with the following code that is located in the python script, gcm_script.py, running on our application server:

```
client = xmpp.Client('gcm.googleapis.com', debug=['socket'])
client.connect(server=(SERVER,PORT), secure=1, use_srv=False)
```

```
auth = client.auth(USERNAME, PASSWORD)
```

After the connection has been made, gcm_script.py (the XMPP application server script) runs indefinitely in a loop. The script switches between three different tasks: receiving a message (done through the message_callback function), handling a sent message from the XMPP application server script to GCM server (this is done through flush_queued_messages()) , and sleeping for a certain period of time (this is executed using the bash script in Figure 4).

After a connection is successfully established, a message listener is added to tell the python script what to do once the message has been received. The listener is added by the following code in the gcm_script.py python script:

```
client.RegisterHandler('message', message_callback)
```

Receiving a message (from GCM server)

There are two types of messages that are received from the GCM server. The first message type is a data message which has no message type or a null message type. The second type is an acknowledgement (ACK) or not acknowledgement (NACK) message and has the message type ACK or NACK. In either instances ACK or NACK types tell the XMPP application server script that the message sent to GCM was received. In the case of NACK, it just lets the application server know that the message was incorrectly received, or corrupt, but still received.

The message_callback function is called when a new message is received from the established client connection. In the message_callback function we parse the incoming XML message into JSON using the following code:

```
gcm_json = gcm[0].getData()
msg = json.loads(gcm_json)
```

Handling a message with no type (null message type)

After parsing, the message type is determined, if the message does not have a message type then we know that it is a message from the client application, and thus contains data to parse, the magnet link in our case. If the message is from the client application then an acknowledgement message is sent to tell GCM that the message from the client application has been received successfully. This acknowledgement is sent using the following code:

```
send({'to': msg['from'], 'message_type': 'ack', 'message_id': msg['message_id']})
```

A message from the application server is then queued on our server. The message that is queued becomes the message to be sent to the client application. This message is sent to the client application to notify the user that the torrent has been uploaded correctly. This queued message is later dequeued when the function "flush_queued_messages()" is called. Next, the magnet link is parsed from the 'data' JSON object of the message with the following code:

```
magnet link = msg['data']['data']
```

With that code as the parameter, the bash script (see figure [1]) is called with the following code:

```
subprocess.call(["./download_magnet_link.sh", magnetlink])
```

Handling a sent message from the XMPP application server (script) to GCM server

In the indefinite loop, after client.Process(1) (or message_callback) is called, flush_queued_messages() is called. This function handles the queued message, in our case the message "Torrent has been uploaded" is sent back to the client application. The following code allows the message to be sent back to the client application:

```
send_queue.append({'to': msg['from'],'message_id': random_id(), 'data': {'message':"Torrent has been uploaded"}})
```

This function sends the queued message by popping the first message off the queue and increasing the total number of unacknowledged messages by decrementing unacked_messages_quota (which determines how many unacknowledged messages the application server can hold). The total number of unacknowledged messages keeps track of the messages that have been sent but have not yet been acknowledged. For example, if unacked_messages_quota is at 99, then flush_queued_messages() was just called then the total number of unacknowledged has increased (or the total number of queued messages the server can hold has decreased).

Handling a message with type ('ACK' or 'NACK' message type)

When client.Process(1) (or message_callback) is called again, after having completed flush_queued_messages() in the scripts indefinitely loop, message_callback with receive a message or type ACK or NACK. This ensures that when the message_callback function is called again, it goes to the else if statement and decrements the total number of unacknowledged messages by incrementing unacked_messages_quota. This function still decrements the total number messages (even if a NACK message is received) because of the nature of the NACK messages described above. A NACK is not that the message has not been received; it is that it got corrupted or damaged during transmission to GCM, but was still received, and can be cleared as received from the XMPP application server script. Compared to a NACK message, an ACK message ensures that the received message to GCM was not corrupted or damaged during transmission and can be cleared by the XMPP application server script.

For example, if flush_queued_messages() is called then we know that the XMPP application server (script) can only hold 99 more unacknowledged messages (thus unacked_messages_quota is 99). Then when an ACK or NACK message is received unacked_messages_quota is bumped back to 100, letting the application server know that it can once again let 100 messages be queued up without acknowledgement. By incrementing the amount of messages to be queued we essentially tell the application server that the last sent message was received.

Sleeping for a certain period of time (last function of the indefinite loop)

This puts the entire python process (gcm_script.py) running to sleep for the specified number of minutes using the bash script (Figure 4). The code is as follows:

```
subprocess.call([“./sleep.sh”, “1m”])
```

This calls the bash script sleep.sh for 1 minute then get up again and check for newer messages. This is used so that it doesn't constantly check for messages while the python script runs. The script is used so that the Pi isn't constantly using power all the time.

```
#!/bin/bash
## Sleep.sh: simply executes sleep command for amount of time passed to it

echo "Sleeping"

sleep "$1"

echo "Woke up!"
```

Figure 4

Instructions for running gcm_script.py

The python script is a script that is always running as a background process. To start the python script you would enter the command “python gcm_script.py&”. To kill the python script you would need to find its process ID and kill the process ID.

Setting up GCM

To set up Google Cloud Messaging (GCM) we registered for an account to be able to access Google's developer's console. Once we had made an account for GCM we were required to create an application name and application package to target the client application, which is our project, called Raspberry Torrent, that GCM is going to be enabled on (shown in figure 5). This step is needed or the GCM servers will not know which GCM application it should be communicating with [11]. The parameters that we entered for App name was “gcmimplementation” and the android package name was “android:com.example.rooshan.gcmimplementation”. This step basically tells GCM that we would like to use their services with our application.

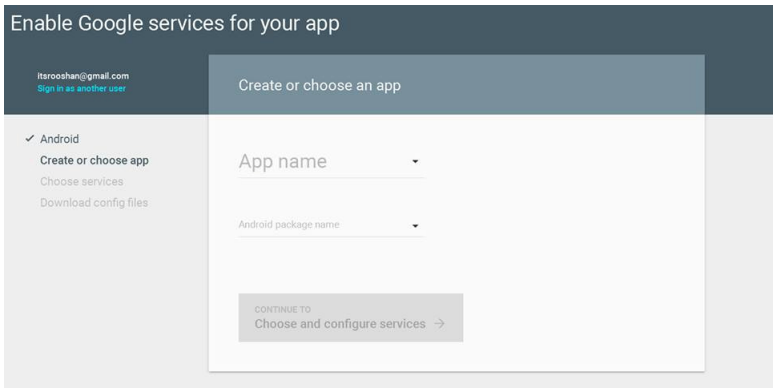

Figure 5

Once we had entered our application name and application package name, GCM was enabled for our application to use. As shown in figure 6, GCM then provided us with two of the three required key components that are used to communicate with GCM, the API key and the sender ID. Once GCM is enabled a configuration file is generated from the required app name and package name by GCM (figure 7). This is the final step of getting GCM registered to use with our application. Once GCM is registered the generated configuration file is placed in the root directory of the client application.

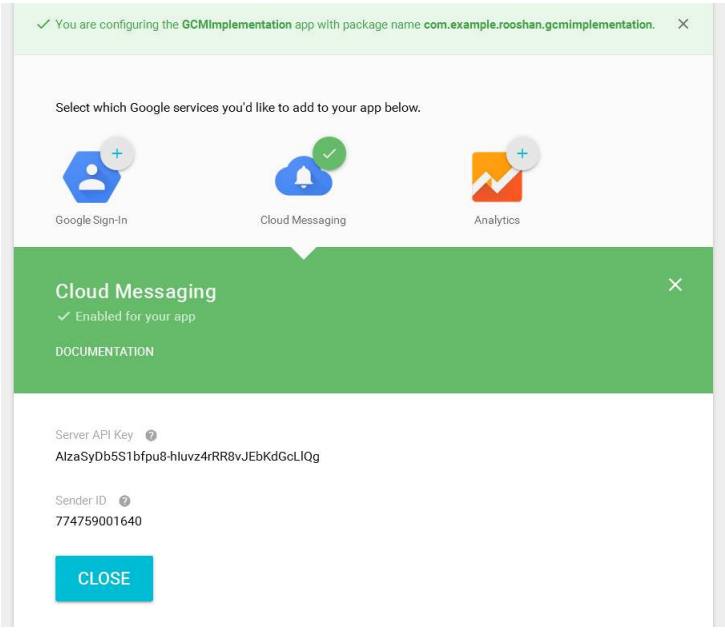

Figure 6

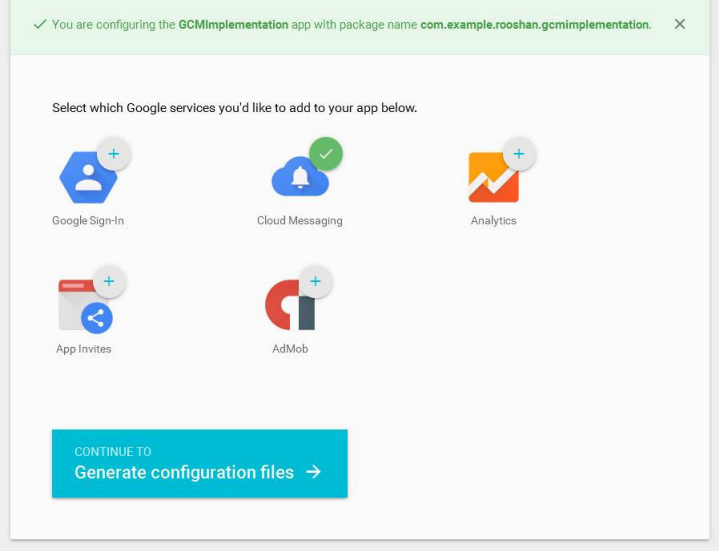

Figure 7

The next step in setting up GCM is to tell our client application to talk to GCM. To enable GCM on the client application we first needed to enable it in the project structure (as shown in figure 8).

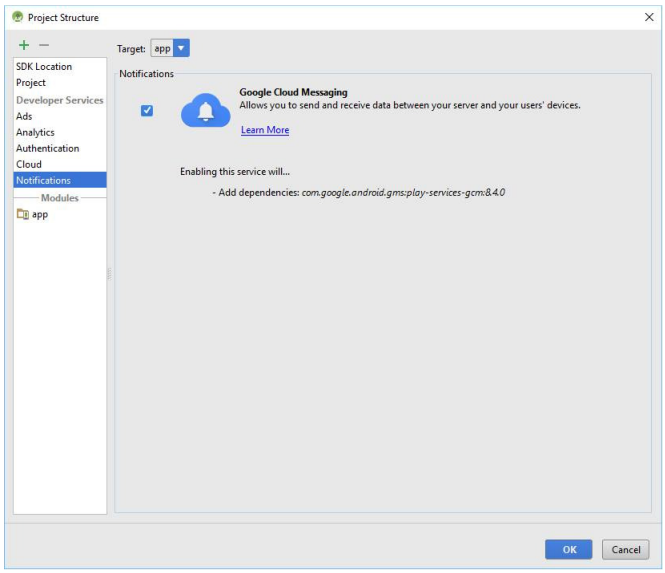

Figure 8

The next step is to modify our client application's configuration file, androidmanifest.xml. We modify this file to tell our client application that we want GCM

to be able to communicate with our application; the term for this is called permission enabling. In other words, we are telling our client application to give permission to GCM so that it can run without problems. Within the file androidmainfest.xml, we put the following code to allow for GCM to have permission to work with our application:

```
<uses-permission android:name="com.google.android.c2dm.permission.RECEIVE" />
```

The next step is to tell our application to listen for messages from GCM, which is done in androidmanifest.xml. The code to achieve this is:

```
<action android:name="com.google.android.c2dm.intent.RECEIVE" />
```

Setting up the Client Application

To set up the client application we will use the configuration file that GCM generated that is located in the root directory of the client application. To start this setup we needed to generate a registration token. This registration token is generated with an intent service from the main activity when you first open your application. The intent service which is known as RegistrationIntentService in our application uses the gcm_ defaultSenderId from the configuration file from GCM, to generate the registration token. Within our application the registration token is printed on a text field within the application in the main activity. In a real application we would send this registration token back to the application server to store in a database.

Receiving a message

In order for GCM to receive a message a service in the file androidmanifest.xml is started. To start the service the following code is added to androidmanifest.xml:

```
<service android:name="com.example.rooshan.gcmimplementation.MyGcmListenerService" </service>
```

Once this code is added MyGcmListenerService class will implement the GcmListenerService. GcmListenerService is important to implement so that the following function, onMessagedRecieved, will perform a set of actions when a message is received from GCM to the application. These actions include parsing the incoming

data, sending the parsed data to the main activity, and creating a notification from the parsed data. Parsing data is done with the following code, where the JSON object 'message' is extracted from the incoming message:

```
String dataFromGCMSever = data.getString("message");
```

The parsed data is then sent to the main activity using an intent service using the following code:

```
Intent message_recieved= new Intent("MESSAGE_RECIEVED"); message_recieved.putExtra("Message", dataFromGCMSever);
```

Lastly, a notification is created from the parsed data which contains the messages and is sent to the phone using the following function:

```
sendNotification(dataFromGCMSever);
```

Sending a message

Sending a message is done when a user clicks on the send button on the main activity. An instance of GCM is added to the main activity using the following code:

```
final GoogleCloudMessaging gcm = GoogleCloudMessaging.getInstance(this);
```

After the user clicks the send button the text from the main activity “MessageToSend” is grabbed and sent to an asynchronous task where GCM preps the message for sending. The asynchronous task is created to off load the work from the main UI thread. We want to provide this asynchronous task to help the main UI thread, if it is not provided your UI will become unresponsive. The asynchronous task then encapsulates messageToSend into a JSON object using the following code:

```
data.putString("data",messageToSend);
```

Lastly, gcm_defaultSenderId is once again grabbed from the configuration file that is in the root directory of the application and the message is sent using the following code:

```
gcm.send(getString(R.string.gcm_defaultSenderId) + "@gcm.googleapis.com",
Integer.toString(messageId.incrementAndGet()), data);
```

The atomic integer within the code assures that two messages do not have the same message ID. After the message is sent the asynchronous thread is removed and returns to the main thread. If the message is not sent successfully, the asynchronous task runs “onPostExecute” to log the failure of the event. The code for “onPostExecute” is as follows:

```
protected void onPostExecute(String result)
 {
        if (result != null)
        {
                Log.i(CLASSTAG, "send message failed: " + result);
         }
}
```

Problems and how we solved them

One of the original problems we had was with the transmission daemon because we were connecting to the Raspberry Pi remotely using Weaved [10]. While we were using Weaved to work on the Pi we ran into permission problems when trying to get the torrent application to work. To see if this problem existed locally, we tried to run rTorrent on the Raspberry Pi itself. For this test the same permission errors continued to happen. Although when we tried rTorrent on debain, rTorrent picked up the Linux ISO we were testing with from the watch folder and started to download it. Once we ran this test we realized that the .sh script that we were using on the Pi was not generating the torrent file correctly. The .sh script was giving an authentication error saying we didn’t have

permission. To try to fix this problem we went into the settings and changed the authentication to false. However, once we restarted the transmission the default setting for authentication was set back to true. To fix this problem instead of trying to set the authentication to false, we left the settings.json file as it was and changed the bash script to provide the credentials. With the credentials in the bash script there was no reason to go to the settings.json file to authenticate. Once the work around was implemented we realized why the settings.json file would not keep our changes. To change any setting of a transmission daemon you need to stop the daemon or the default settings will always override your changes.

We were going to use rTorrent for the torrent client but have since switched to Transmission CLI because of our first problem with the transmission daemon. One of the main reasons we switched to Transmission CLI was because receiving meta data from rTorrent was bounded by the number of trackers who had the meta data for the torrent. In order for the torrent to begin downloading, meta data had to be pulled by the trackers and thus caused performance issues. By changing to a different daemon to handle transmission we fixed the issues that rTorrent had caused us.

Team Member's contributions

The parts of the project were equally distributed between the three team members. Ashley was responsible for setting up the Raspberry Pi. This included: installing the OS, setting up the transmission daemon, creating the two bash scripts, and writing the python script. Rooshan was responsible for setting up GCM and the client application, Raspberry Torrent. Rooshan and Ashley worked together to connect the Raspberry Pi with GCM. Jaleesa was responsible for the project paper which included: research for history of Torrenting, taking Rooshan and Ashley's notes and putting them into paragraphs and editing the paper. We helped each other out when someone needed help. There was not just one person working on the whole project, we were a team.

Conclusion

To sum everything up we managed to implement our project by breaking up each aspect of the project, getting everything to work individually, and then put all of the pieces together. We set up GCM to work with the android phone and the Raspberry Pi so that we would have a connection that worked. The Raspberry Pi was set up with Transmission CLI to allow for the Pi to become a torrent box. We also created a client application, which we named Raspberry Torrent. This client application is where the user would interact with our torrent box. The user does not directly contact the Pi but contacts GCM which in turn talks to the Pi on behalf of the user. This is also the case when the torrent is finished, the Pi would contact GCM which will pass on the message to the user's android phone. With this set up all the user needs is the client application, Raspberry Torrent, downloaded on their phone to start a torrent on the torrent box. Overall, this project was unique as no other projects we looked at implemented a torrent box connected to an android phone. We believe the project was a success as we were able to accomplish what we set out to do.

References

[1]J. Kurose and K. Ross, *Computer networking: a top-down approach*, 6th ed. Boston: Pearson, 2013, pp. 144-151.

[2]"The Raspberry Pi 2 Is Faster, More Powerful, and Available Right Now", *Lifehacker.com*, 2016. [Online]. Available: http://lifehacker.com/the-Raspberry-pi-2-is-faster-more-powerful-and-availa-1682814956. [Accessed: 10- Apr- 2016].

[3]"Raspberry Pi TorrentBox: Build a Always-On Torrent Machine", *Pi My Life Up*, 2015. [Online]. Available: http://pimylifeup.com/Raspberry-pi-torrentbox/. [Accessed: 10- Apr- 2016].

[4]"The BBS Corner - Main Page", *Bbscorner.com*, 2016. [Online]. Available: http://www.bbscorner.com/usersinfo/bbshistory.htm. [Accessed: 10- Apr- 2016].

[5]"How to Turn a Raspberry Pi into an Always-On BitTorrent Box", *Howtogeek.com*, 2016. [Online]. Available: http://www.howtogeek.com/142044/how-to-turn-a-Raspberry-pi-into-an-always-on-bittorrent-box/. [Accessed: 10- Apr- 2016].

[6]"Raspberry Pi: What are its limitations? | ITProPortal.com", *ITProPortal*, 2013. [Online]. Available: http://www.itproportal.com/2013/04/25/Raspberry-pi-what-are-its-limitations/. [Accessed: 10- Apr- 2016].

[7]R. improved, "Raspberry Pi 2 vs Raspberry Pi: How the DIY computer has been improved", *TrustedReviews*, 2015. [Online]. Available: http://www.trustedreviews.com/opinions/Raspberry-pi-2-vs-Raspberry-pi. [Accessed: 10- Apr- 2016].

[8]"Raspberry Pi • View topic - Adding RAM to Pi", *Raspberrypi.org*, 2016. [Online]. Available: https://www.Raspberrypi.org/forums/viewtopic.php?f=63&t=14664. [Accessed: 11- Apr- 2016].

[9]"Steves Computer Vision Blog: Controlling Raspberry Pi via text message", *Stevenhickson.blogspot.ca*, 2016. [Online]. Available: http://stevenhickson.blogspot.ca/2013/03/controlling-raspberry-pi-via-text.html. [Accessed: 10- Apr- 2016].

[10]"Remote Manage Networked Devices Anywhere", *Weaved Inc*, 2016. [Online]. Available: http://weaved.com. [Accessed: 10- Apr- 2016].

[11]"Add Google Services", *Google Developers*, 2016. [Online]. Available: https://developers.google.com/mobile/add?platform=android&cntapi=gcm&cnturl=https:%2F%2Fdevelopers.google.com%2Fcloud-messaging%2Fandroid%2Fclient&cntlbl=Continue%20Adding%20GCM%20Support&%3Fconfigured%3Dtrue. [Accessed: 10- Apr- 2016].

[12]"TransmissionHowTo - Community Help Wiki", *Help.ubuntu.com*, 2016. [Online]. Available: https://help.ubuntu.com/community/TransmissionHowTo. [Accessed: 11- Apr- 2016].