COSC 6377 – Project Report

Name: Hai-Y Michael Tran Nguyen | Peoplesoft: 0925358 | Date: 11/26/17

Number of Bytes sent to the Raspberry Pi per one image frame

To determine the number of bytes sent between all the moving parts, I designed the python scripts to log everything that is sent and received based on a WRITE_LOG flag. Logs are stored in "./Log/*.txt". Since I am using sockets that sent the packets through TCP, we can then determine the network overhead of each image frame.

To determine the number of packets sent, we need the MTU, which is determined through some simple commands (like: ip ad | grep mtu). With the MTU we can determine the number of packets sent per image, and with that we can determine the total overhead. With those values, we can determine the overhead %.

For the following experiments, I took 10 examples in a row (10 images in a row, or 10 verifications in a row).

Image Data

Raspberry Pi to Server (Image)

Image	Send Bytes	MTU	Packets CEILING.MATH(Send Bytes/MTU)	Send Overhead (TCP+IP) * Packets	Overhead % (Overhead/Send+Overhead) * 100
1	25843	1500	18	720	2.710537213
2	25507	1500	18	720	2.745262516
3	23263	1500	16	640	2.677488181
4	21747	1500	15	600	2.684924151
5	19439	1500	13	520	2.605340949
6	17287	1500	12	480	2.701637868
7	19851	1500	14	560	2.743618637
8	21723	1500	15	600	2.687810778
9	23071	1500	16	640	2.699169162
10	24575	1500	17	680	2.692536131
Average	22230.6	1500	15.4	616	2.694832559

Server to Android (Image)

Image	Send Bytes	MTU	Packets CEILING.MATH(Send Bytes/MTU)	Send Overhead (TCP+IP) * Packets	Overhead % (Overhead/Send+Overhead) * 100
1	25047	1280	20	800	3.095136766
2	25243	1280	20	800	3.071842722
3	25083	1280	20	800	3.09083182
4	25299	1280	20	800	3.065251542
5	25139	1280	20	800	3.084158988
6	25031	1280	20	800	3.097053927
7	25075	1280	20	800	3.09178744
8	24911	1280	20	800	3.111508693
9	25063	1280	20	800	3.093221977
10	25075	1280	20	800	3.09178744
Average	25096.6	1280	20	800	3.089258132

Android to Server (Image Request)

Image	Send Bytes	MTU	Packets CEILING.MATH(Send Bytes/MTU)	Send Overhead (TCP+IP) * Packets	Overhead % (Overhead/Send+Overhead) * 100
1	206	1500	1	40	16.2601626
2	158	1500	1	40	20.2020202
3	158	1500	1	40	20.2020202
4	158	1500	1	40	20.2020202
5	158	1500	1	40	20.2020202
6	158	1500	1	40	20.2020202
7	158	1500	1	40	20.2020202
8	158	1500	1	40	20.2020202
9	158	1500	1	40	20.2020202
10	158	1500	1	40	20.2020202
Average	162.8	1500	1	40	19.80783444

Server to Raspberry Pi (Image Acknowledgement)

Image	Send Bytes	MTU	Packets CEILING.MATH(Send Bytes/MTU)	Send Overhead (TCP+IP) * Packets	Overhead % (Overhead/Send+Overhead) * 100
1	37	1280	1	40	51.94805195
2	37	1280	1	40	51.94805195
3	37	1280	1	40	51.94805195
4	37	1280	1	40	51.94805195
5	37	1280	1	40	51.94805195
6	37	1280	1	40	51.94805195
7	37	1280	1	40	51.94805195
8	37	1280	1	40	51.94805195
9	37	1280	1	40	51.94805195
10	37	1280	1	40	51.94805195
Average	37	1280	1	40	51.94805195

Verification Data

Raspberry Pi to Server (Request Verification)

Image	Send Bytes	MTU	Packets CEILING.MATH(Send Bytes/MTU)	Send Overhead (TCP+IP) * Packets	Overhead % (Overhead/Send+Overhead) * 100
1	52	1500	1	40	43.47826087
2	52	1500	1	40	43.47826087
3	52	1500	1	40	43.47826087
4	52	1500	1	40	43.47826087
5	52	1500	1	40	43.47826087
6	52	1500	1	40	43.47826087
7	52	1500	1	40	43.47826087
8	52	1500	1	40	43.47826087
9	52	1500	1	40	43.47826087
10	52	1500	1	40	43.47826087
Average	52	1500	1	40	43.47826087

Server to Android (Verification Acknowledge)

Image	Send Bytes	MTU	Packets CEILING.MATH(Send Bytes/MTU)	Send Overhead (TCP+IP) * Packets	Overhead % (Overhead/Send+Overhead) * 100
1	163	1280	1	40	19.7044335
2	215	1280	1	40	15.68627451
3	215	1280	1	40	15.68627451
4	215	1280	1	40	15.68627451
5	215	1280	1	40	15.68627451
6	215	1280	1	40	15.68627451
7	215	1280	1	40	15.68627451
8	215	1280	1	40	15.68627451
9	215	1280	1	40	15.68627451
10	215	1280	1	40	15.68627451
Average	209.8	1280	1	40	16.08809041

Android to Server (Verification)

Image	Send Bytes	MTU	Packets CEILING.MATH(Send Bytes/MTU)	Send Overhead (TCP+IP) * Packets	Overhead % (Overhead/Send+Overhead) * 100
1	162	1500	1	40	19.8019802
2	162	1500	1	40	19.8019802
3	162	1500	1	40	19.8019802
4	162	1500	1	40	19.8019802
5	162	1500	1	40	19.8019802
6	162	1500	1	40	19.8019802
7	162	1500	1	40	19.8019802
8	162	1500	1	40	19.8019802
9	162	1500	1	40	19.8019802
10	162	1500	1	40	19.8019802
Average	162	1500	1	40	19.8019802

Server to Raspberry Pi (Verification)

Image	Send Bytes	MTU	Packets CEILING.MATH(Send Bytes/MTU)	Send Overhead (TCP+IP) * Packets	Overhead % (Overhead/Send+Overhead) * 100
1	87	1500	1	40	31.49606299
2	87	1500	1	40	31.49606299
3	87	1500	1	40	31.49606299
4	87	1500	1	40	31.49606299
5	87	1500	1	40	31.49606299
6	87	1500	1	40	31.49606299
7	87	1500	1	40	31.49606299
8	87	1500	1	40	31.49606299
9	87	1500	1	40	31.49606299
10	87	1500	1	40	31.49606299
Average	87	1500	1	40	31.49606299

Observations

As we can see, the total overhead of sending an image was higher, but the overall overhead % when comparing it to the total bytes sent was only around 2-3%. The total overhead of everything else (requests, responses, acknowledgements, etc) ranged from ~20-50%. Therefore, we can see that the overhead % was much higher when the payload was smaller, and much smaller when the payload was larger.

Network Tradeoff between OpenCV and AWS Rekognition

OpenCV

OpenCV must be trained before you can start face recognition. This process takes approximately 7 seconds with 26 faces. As more faces are added, the training time will increase.

Image	Time Elapsed (s)	Bytes
1	0.080661	18839
2	0.053607	17801
3	0.1385	20821
4	0.148823	21459
5	0.151474	21468
6	0.142346	21454
7	0.130221	21656
8	0.168614	21215
9	0.150761	21062
10	0.123114	21052
Average	0.1288121	20682.7

AWS Rekognition

Image	Time Elapsed (s)	Bytes Call 1	Bytes Call 2	Total Bytes Sent
1	1.956108	19985	20020	40005
2	1.559946	19970	20005	39975
3	1.68897	20194	20229	40423
4	1.610744	20118	20153	40271
5	1.60587	20854	20889	41743
6	1.566456	20998	21033	42031
7	1.431174	18668	18703	37371
8	1.566421	20462	20497	40959
9	1.54109	20323	20358	40681
10	1.498116	20285	20320	40605
Average	1.6024895	20185.7	20220.7	40406.4

Observations

As we can see from the above results, OpenCV has a faster overall execution time and does not require any network usage. While AWS Rekognition does have to send about 40000 bytes of data per image, and takes much longer to execute. There is some processing time for OpenCV, like training the face recognition software, and the processing time to compare faces, but for a small amount of faces, it should be fine.

One problem with using OpenCV versus AWS Rekognition was that OpenCV sometimes outputted low confidence when comparing faces. It also had some face detection issues (like not being able to see a face when there was clearly a face in the image).

Overall for a project of this magnitude, I think OpenCV wins out as it is more efficient in terms of processing and network to use it over AWS Rekognition. (Plus, it’s free compared to AWS Rekognition – Which is only free for 5000 API calls)

Fastest Image Rate

These were the following conditions/settings used to get the fastest image rate:

- Home Network
- Logging Disabled
- Using OpenCV
- Using Non-polling

The fastest image rate from Raspberry Pi to Server to Android that I was able to achieve was approximately 6 FPS.

The following are the bottlenecks that limited the FPS that I observed:

- OpenCV frame capture took approximately 0.1 seconds to execute on the Raspberry Pi, which means in an ideal network situation (everything over the network was instant), the maximum FPS could only be 10 with OpenCV.
- Image from Server to Android was a big bottleneck, as the android application could only get a frame as fast as it was able to render the image on screen. I tried asynchronously fetching the image but that led to some weird situations where it would slow down in FPS and suddenly spike in FPS constantly therefore I stayed with synchronous calls.

Polling vs Non-polling

With the polling approach, it would run on a timed loop (every 2 seconds) to send a 52-byte request to the server and receive a 48-byte response from the server if no authorization currently. If there was an authorization it would receive an 87-byte response.

For non-polling on the other hand, it would send a 52-byte request to the server and then just wait for the server to respond. Once the server responds with an 87-byte response, (could be instantly, or could be after x number of seconds) then it sends another request to the server and waits for it again.

The pros of using non-polling was that it did not unnecessarily waste network time constantly sending a request to the server. Instead it only sent one request per one verification. Whereas non-polling sends one request and got one response every 2 seconds no matter if there was no verification.

The con of using non-polling was that there still needs to be a heartbeat request to the server every now and then to make sure the connection is still alive. It also tied up a background thread on both server and raspberry pi to wait for the response.

DIY Tutorial

Goal

The goal of this project is to build a physical authentication system that will stream a series of images (or a video) to an android app and use face recognition to inform an admin of who is currently in the picture. This admin will be able to authenticate, reject, or send a custom message to the raspberry pi and it will output it through speakers. The admin will also be able to program a new user into the face recognition software on command.

System Requirements

There are several requirements to have before you can start:

- Raspberry Pi 2B
    - AWS, VL53L0X, MPG123, OpenCV
- USB Webcam (connected to Raspberry Pi)
- VL53L0X Distance Sensor (connected to Raspberry Pi)
- Speaker (connected to Raspberry Pi)
- Windows Server from AWS EC2 with an Elastic IP (Static Public IP)
    - Ports 5000 and 5001 should be open to the public
    - AWS, pyodbc
- Windows Server from AWS EC2 with an Elastic IP (Static Public IP)
    - Microsoft SQL Server with the correct ports open to the public
- Microsoft Visual Studio with Xamarin
- Android Phone and/or Android Emulator

What you will learn

You will learn about all the moving parts (Raspberry Pi, Sensors, Webcams, Main Server, SQL Server, Android, AWS, etc.) that have to work together in unison to provide a simple video stream with authentication functionality to a user. With further usage, you will discover the limitations (of the FPS) of this program due to network and processing times.

Setup: Raspberry Pi

First you should connect the USB Webcam and Speaker to the Raspberry Pi. Secondly you should connect the VL53L0X Distance Sensor to the Raspberry Pi (See http://www.st.com/en/imaging-and-photonics-solutions/vl53l0x.html on how to connect the sensor to the Pi).

The following Interfaces should be enabled on the Raspberry Pi: Camera, I2C. You should also enable SSH and VNC as it will make life easier.

Install AWS, VL53L0X, MPG123, OpenCV onto the Raspberry Pi. Note: OpenCV installation is a time-consuming process and should be done ASAP.

Now get the folder "./RaspberryPi" from the git repository at https://github.com/cosc6377/project-m1-michaeltran.

Modify the following scripts:

- Client/client.py
    - HOST: The Main Server PUBLIC IP Address
    - PORT: The Port
    - WRITE_LOG (True/False): Turn logging on or off
    - COLLECTION_ID: The AWS COLLECTION_ID

Setup: Server

You should obtain 2 servers from the AWS EC2 service. One will be for the main server, and one will be the SQL Server that contains the database of names and distances.

On the server for SQL Server, create a database called FACEDB. Run the following script from the git repository on FACEDB: "./Database/DBSchema.sql".

On the main server, install AWS and pyodbc.

On the main server, get the following folder from the git repository "./Server".

Modify the following scripts:

- server.py
  - HOST: The Main Server PRIVATE IP Address
  - PORT: The Port for Server/Pi Communication
  - LONG_POLL (True/False): Turn long poll on or off
  - WRITE_LOG (True/False): Turn logging on or off
- httpserver.py
  - HOST: The Main Server PRIVATE IP Address
  - PORT: The p
- customsqlserver.py (connection string)
  - Server: Public IP Address of the SQL Server
  - UID: Username
  - PWD: Password

Setup: Android

On any PC with visual studio and Xamarin, get the following folder from the git repository "./Android".

Open the project and find all references to `CustomHTTPServer` and change the first parameter to the main server public IP address and the second parameter to the port.

Build and deploy the application on either your android phone or emulator.

Setup: AWS

Assuming you have installed and configured AWS on the server (if you haven't, see http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/get-set-up-for-amazon-ec2.html), run "./Server/Amazon Api/uploadimage.py" to do an initial setup for the face collection.

Setup: Final

Now that all 3 parts are setup, we should be able to run the application. Follow these steps:

1. Ensure Main Server and SQL Server are up
2. On the Main Server: run the following scripts on two different command prompts: server.py and httpserver.py
3. On the Raspberry Pi: run client.py -f <option> (option can be either AWS or OPENCV, default is AWS)
4. On the Android: start the application and click on of the options

System Performance

To deliver a clear image for the given FPS to the user and face recognition software, a good internet connection is required for all 3 main parts (Raspberry Pi, Server and Android). Without a good connection, frame rates and image quality can and will suffer dramatically.

A faster android phone will also help considerably in maintaining the FPS.

OpenCV training time will also increase as more faces are added to the database.

Conclusion

In conclusion, I learned a great deal of things while working on this project.

This was the first time I have ever used sockets to implement a server. I actually got stuck on sockets for the longest time as I had (wrongly) initially thought you could send as much data as you wanted through a socket and the receiver would magically know when it received everything. I got around this by prefixing each message with a 4-byte length of the message onto the front of the message, and the receiver would read that 4-byte length and then read that length from the buffer.

This was also the first time I had ever used a raspberry pi. It was really cool to use something as little as that to perform these operations. I also had very little Linux experience before this, so learning what Linux has to offer was quite interesting.

Fortunately, I had experience with using windows servers, SQL servers, and android development, so I was able to apply that knowledge to this project. Which made setting up those parts easier.

It was quite tough making all these moving parts work together into more or less a singular application, but in the end, it was a great learning experience.