CSE520S - Final Report
Home Monitor System

Haobo Xing Xueting Yu Yue Hu

1. Introduction

1.1. Project Background

What happens at your apartment or house when you hang out with friends? Do you want to check if your pet is doing well when staying home alone? Have you ever worried about if your apartment gets theft or fire when you not at home?

Here is the Home Monitor System to help. People are able to use iOS device to monitor their house as live video stream from everywhere as long as you connected to Internet.

1.2. Raspberry Pi & Webcam

The Raspberry Pi is a series of credit card-sized single-board computers developed in the United Kingdom by the Raspberry Pi Foundation to promote the teaching of basic computer science in schools and in developing countries. The original model became far more popular than anticipated, selling outside of its target market for uses such as robotics. Accessories including keyboards, mice and cases are not included with the Raspberry Pi. Some accessories however have been included in several official and unofficial bundles. [1]

We designated raspberry pi as our client platform for two reasons. First, it is economical enough for home using; and second, the raspberry pi has corresponding raspberry camera module, which is super compatible to the pi. We didn't find any other set of equipment more compatible than this one.

The Raspberry Pi camera module can be used to take high-definition video, as well as stills photographs. It's easy to use for beginners, but has plenty to offer advanced users if you're looking to expand your knowledge. There are lots of examples online of people using it for time-lapse, slow motion and other video cleverness. [2]

Webcam is super compatible with Raspberry Pi just by connecting them with a cable, the CSI interface and some configurations on Raspberry Pi.

1.3. EC2 Instance

Amazon Web Server (AWS) is no doubt a good option for start-ups, as it provides a wide range of low cost and extendable cloud computing services. The Amazon Elastic Compute Cloud (known as EC2) is one of the central products of AWS, which allows users to run their programs on the virtual machines provided by Amazon. In this project, we deployed web server on EC2 Linux instance on Amazon Web Server. [3]

1.4. iOS Application

iOS is the mobile operating system that runs on Apple's mobile devices. Nowadays, when it comes to mobile device operating system market share, Apple takes 43.6 percent in United States till January 2016. We decided to build our mobile application fit in iPhone 6s for this project. [4]

2. System Design

2.1. Project Plan

We planned to design a system which can help us monitor the activity at home when we are not in home based on Raspberry PI and camera modular, since we want to see activities anywhere and anytime, we also developed an APP on smartphone, right now, we only have IOS app on iPhone, in future, Android and Windows phone version may come out.

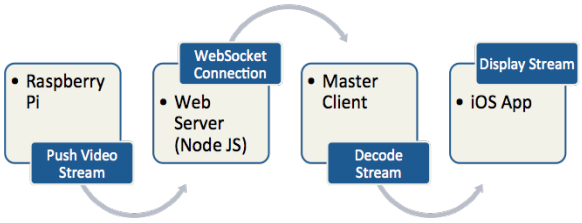
Our system contains three parts.

The first part is Raspberry PI which is used as a camera monitor in our project, we use it to transcribe live stream and push it to our server, right now, we only have one raspberry PI and one camera, we can extend our project by adding several Raspberry PI cameras in different room at home, living room and bedroom etc.

The second part, server, which is based on Amazon Web Service EC2, contains a node JS server and a decoder used for decoding video stream, this decoder is format related. Server opens port listening for socket connection from clients, every client communicates with server through one socket, and server also opens a port listening for video stream pushed from Raspberry PI. [5]

The third part is client part, unlike most of the parallel client, we design our client part a two-level client, the master client and the end client based on master client, there are several reasons that we design the client part this way, first is if all clients fetch stream from server, it would be a heavy load at server (server also need to listen for stream pushed from Raspberry PI), second is having a master client and make the end client which is IPhone in our project can make our system extensible, for example, if a master client can support a certain number of end client with high performance, when the number of end client becomes large and we can add one master client to support the additional end clients. What's more, it can be easy for us to manage end clients based on their locations by having one or multiple master client for each location.

Figure 1 Home Monitor System Structure



3. Implementation

3.1. Build Process of PI & Camera

To begin with, we should initialize the raspberry pi. And then connect the camera module to the pi. Eventually we configure the webcam and test on both of still photo and video monitor.

The initialization of raspberry pi includes several following considerations. For the operating system, we choose the rasbian, which is popular among pi players so that if we encounter some problems, we can easily get strong supports from the rasbian community. For we use limited-memory microSD card, we decided to apply light version, which just supports command line interaction instead of both of CMD and graphic interaction. The difference between the size of a complete version and that of a light one has at least 3 GB, which really influence our card and the performance of raspberry pi.

The installation of camera module includes two sections. For hardware, the setting up details will be found at our first demo slides. It will be shown that we connect the raspberry camera to the pi's CSI port with its cable. For software, due to we use CMD version, we typed "raspi-config" and then access to an ugly GUI where we choose camera bullet then activating it.

After all of these, typing "sudo reboot" will leads to a successful installation. Then we used mainly "raspistill" and "raspivid" to test our camera side client. It worked. [6][7]

3.2. Build Process of EC2 Instance - Server

In our project, we need two separate Amazon EC2 instances, one is used as server and the other one is used as master client. Since the master client should be able to handle many end clients which are IPhones, we chose the same type of AWS EC2 for both server and master client, which is a T2.micro type, it has one virtual CPU and 1GB memory, based on the description from AWS, its network performance is low to moderate. And the versions of Linux are both Amazon Linux AMI.

For the server side, we configured it as a node JS server, listening to connections from Raspberry PI and clients, also with a decoder.
Installing Node.JS:

The easiest way to install Node.JS is installing by yum-get, but due to the upgrading policy of Amazon AWS, installing Node.JS with this way may install the out-of-date version of Node.JS, note that Node.JS is a rapidly changing language. So we choose use-nave-no-shell method to install it with the following command:

```
$ mkdir ~/.nave
$ cd ~/.nave
$ wget
http://github.com/isaacs/nave/raw/master/nave.sh
$ sudo ln -s $PWD/nave.sh /usr/local/bin/nave
$ chmod a+x nave.sh
$ sudo chown -R $USER:$USER /usr/local
$ nave usemain stable
$ curl -L https://www.npmjs.org/install.sh | sh
```

After successfully running the above command, check the Node.JS version with the following command:
$ node --version

Choose format:
We choose MPEG-1 as the format of our video stream, there are two reasons, first one is the provided decoder online, and the second one is it's easy to push and fast, since we want the live stream, it won't make server and client have too much work to do.

Implementing the server:

First, we initialize some arguments such as the width and height of video stream and some essential variables for establishing connections. Second, we create a event called connection (Node.JS is event driven) used for listening for connection from clients, it is a websocket server, every time one client connect or disconnect from the server, the server will print information on the terminal.

Third, creating a http server to accept incoming MPEG stream, we can either specify the height and width from the command line or use the default parameter set in our program.
Note: don't use the port which conflict with other running protocol, you can check which port are using currently by netstat command.

Source code for server would be attached.

3.3. Build Process of EC2 Instance - Master client:

First, we need a decoder, which can decode videos in MPEG-1 format. It has to be put in the same folder as client.html.

Second, we want to display the video stream pushed from Raspberry PI on a web browser, so we develop a HTML file used for displaying video. It starts a socket connection to server (need server's IP address to do this) and also it can be used to display any resolution pushed from Raspberry PI without changing the code.

Attached source code for master client.

3.4 Build Process of iOS Application

Since the codec for raw image from EC2 Server is in MPEG-1 and iOS only support MPEG-4, we establish a master client embedded with MPEG-1 decoder by a Javascript library. The iOS

application will connect to master client for data transaction of the video stream through HTTP protocol and display in iOS as a web view when user choose to display specific stream.

Also, user is able to take photo of specific video and save into photo album.

Attached source code for iOS client.

4. Experiments

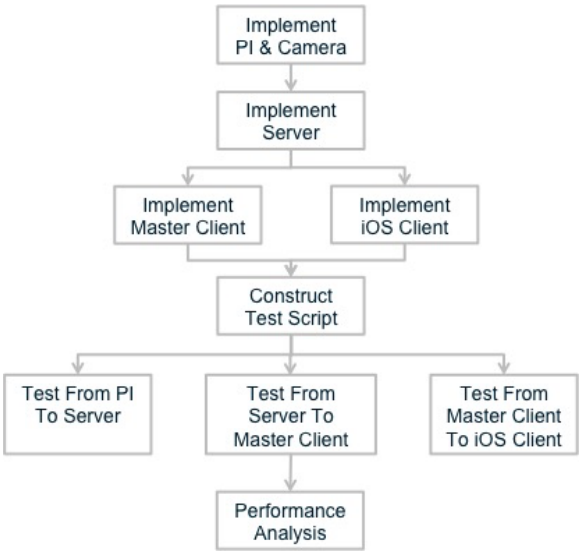
4.1. Test Plan

We need to evaluate the round trip latency from PI to Mobile Devices. However, there is no existing tool to test the latency of this end-to-end task. Hence, we plan to break down our test into three parts. First part is to test the latency from PI to EC2 Server. Second part is to test the latency from EC2 Server to Master Client. The last part is to test the latency from Master Client to Mobile Client.

Since it's pretty hard to synchronize the times of Raspberry PI, server, master client and mobile devices, we use the following method testing the round trip latency between master client and mobile devices. We run a server on master client which is a simple server, just echoing back what is received, and on mobile devices, we send requests and set two timers, one get the CPU time when it sends a request, the other one get the CPU time when it receives a response from master client (all code will be attacked), we compute the difference between these two times and we can achieve multi-user and multi-thread by controlling the sending part.

Within the same test scenario, we will add up all the three part of latency data to measure the end-to-end round trip latency.

Figure 2 Latency Test & Stress Test Workflow



4.2. Test Instance

To do this evaluation, we install several instances. One Raspberry Pi 3 instance, to capture video stream, encode data and upload to Cloud Server; two AWS EC2 instance, one for Server and one for Master Client; one iOS simulator, to receive the video stream and display for users.

Figure 3 Raspberry PI 3 Hardware Feature

EC2 Server & Client Test Bed Info	
CPU	Intel(R) Xeon(R) CPU E5-2676 v3 @ 2.40GHz
Memory	1 GB
System	Amazon Linux AMI

Figure 4 AWS EC2 Instance Feature

EC2 Server & Client Test Bed Info	
CPU	Intel(R) Xeon(R) CPU E5-2676 v3 @ 2.40GHz
Memory	1 GB
System	Amazon Linux AMI

Figure 5 iOS Simulator Feature

iPhone 6s Test Bed Info	
CPU	1.85 GHz dual-core 64-bit
Memory	2 GB LPDDR4 RAM
Storage	128 GB

4.3 Testing Tool

4.3.1 Ping

Since our Raspberry PI, server and master client all have linux-based operating system, we can use the tool provided by linux to test the round-trip time of them. We choose because it can set the number of requests and it will automatically compute the average/min/max round trip time for us, which is really convenient. Another reason is that, right now, we only have one set of Raspberry PI and camera, so there is no concurrency in this part. So ping is excellent for this part of latency testing. [8]

4.3.2 Swift Script Within Timer

There is no existing tool for benchmark test in iOS SWIFT. We programmed in SWIFT to capture the time interval between the time when program send out an HTTP request to master client and the time when we get the call back from Master Client. The difference between this two times is the latency we expect from Master Client to iOS client.

In order to simulate multiple iOS mobile device send HTTP request the same time, we programmed in SWIFT with pre-defined amount of concurrent threads represents certain amount of devices. Each thread will send out same amount of HTTP requests concurrently. After all threads send out request, get response and calculate the specific time interval of their own task, the main thread will be invoke and calculate the average time interval of all the devices. This is the estimated latency from Master Client to iOS client.

Attached source code for master client to iOS client latency test.

4.4. Test scenario design

4.4.1. Normal Data Range

4.4.1.1. Single request per device test
We conducted tests at three different times in one day, which is 10 a.m., 4 p.m. and 10 p.m. In every time slot, we set up 4 scenarios regarding number of devices which simulate 1, 3, 5, 10 devices for concurrent requests.

4.4.1.2. Multi Requests Per Device Test
We conducted tests at three different times in one day, which is 10 a.m., 4 p.m. and 10 p.m. In every time slot, we set up 4 scenarios regarding number of devices which simulate 1, 3, 5, 10 devices to access the same Master Client. Also, we set up 4 scenarios regarding number of requests send out of each device. It includes 1, 3, 5, 10 http requests per device to Master Client the same time.

4.4.2. Extreme Data Range

4.4.2.1. Single Request Per Device Test
We conducted tests at three different times in one day, which is 10 a.m., 4 p.m. and 10 p.m. In every time slot, we set up 6 scenarios regarding number of devices which simulate 50, 100, 250, 500, 750, 1000 devices for concurrent requests.

4.4.2.2. Multi Requests Per Device Test
We conducted tests at three different times in one day, which is 10 a.m., 4 p.m. and 10 p.m. In every time slot, we set up 4 scenarios regarding number of devices which simulate 1, 3, 5, 10 devices to access the same Master Client. Also, we set up 4 scenarios regarding number of requests send out of each device. It includes 50, 100, 250, 500 http requests per device to Master Client the same time.

5. Test Analysis

In this part, we analyze two parts of the result of latency test and stress test, latency test is the normal test, and stress test is focused on the bottleneck of our system.

5.1. Latency Analysis

Based on the design of our system, there are two main factors which can have great influence on latency, first one is the number of devices connected to master client and the second one is the number of requests per device. We test latency at different times of a day.

Note: the unit of y-axis of all pictures is in millisecond.
First, we test single request per device case. The result is shown below:

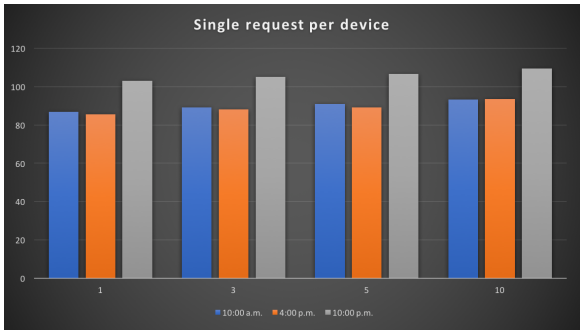


Figure 6 Single Request Per Device Test Result

As we can see from the picture, latency at 10 p.m is obviously larger than latency at other two time periods. The possible reason could be network latency at 10 p.m is larger than network latency in other two times.

Beyond single request per device, we move our attention on multiple requests per device, and the result is shown below (result at 4 p.m):

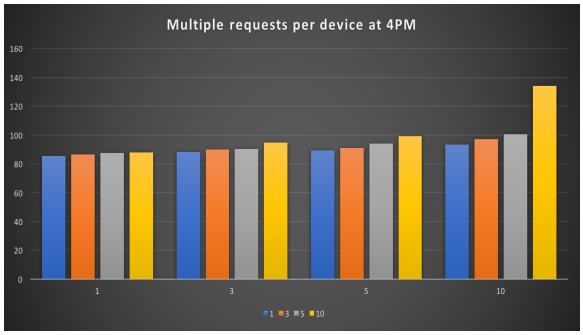


Figure 7 Multiple Request Per Device Test Result

The other two times' result are the same in trend, from these pictures, we can see that with the increasing of devices and requests per device, latency increases, but what is more interesting is which factor can have a bigger influence on latency, so we fix either number of devices or number of requests, we did bunch of this kind of test and most of them seem to draw the same conclusion, which is number of devices has a bigger influence on latency than requests per device.

Next, we find that the latency trend of multiple-requests test is the same as the result of single-request test, latency at 10 p.m is larger than other two, the same reason as before, network is much more busy at 10 p.m.

Finally, we want to know which part of our system has the largest latency. We test with multiple data and use the average number. Result is shown as below:

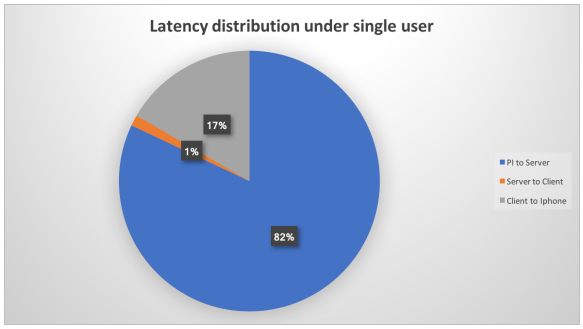


Figure 8 Latency Distribution Under Single Request

The blue part represents latency from Raspberry PI to server, and the other two parts seems

negligible compared to this part. There are probably two reasons for this, first is Raspberry PI has slower CPU and less memory compared with server and master client which are both based on Amazon EC2, the second one is huge amounts of data (video stream) has been pushed from Raspberry PI to server. Due to the above two reasons, latency from Raspberry PI to server can be significant.

5.2. Stress Test Analysis

5.2.1. Query Per Second for Single Master Client

In order to calculate query per second for Master Client, we extract data from single request per device test within the range from 50 devices to 1000 devices. When we simulate 750 devices concurrent requests, we have an average response time around 1000 ms. When simulate 1000 devices scenario, the average response time will be over 1400 ms at any test time. Hence we conclude that 750 devices is the capacity for one Master Client to process request and send response in 1 second.

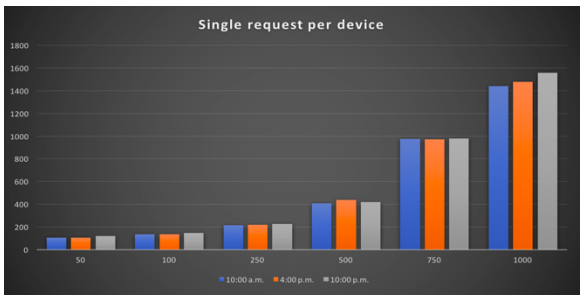


Figure 9 Single Request Per Device Stress Test

5.2.2. Network Latency At Different Time

Under stress scenarios, we found the average response time at different time of the day don't have significant difference. Hence, we conclude that network latency barely have impact for performance.

5.2.3. Decomposing Latency: Number of Device versus Number of Request Per Device

When comparing the multi-request per device scenarios, we found neither the number of device nor the number of request per device don't have significant impact in the response time. In other words, the key factor for the latency is the total number of request for the Master Client.

The average response time within 50 devices and 10 requests per device is very similar to 100 devices and 5 requests per device. Also, the average response time within 250 devices and 10 requests per device is very similar to 500 devices and 5 requests per device. Hence, we conclude that under extreme data scenarios, the total amount of request to Master Client has significant impact to the response time.

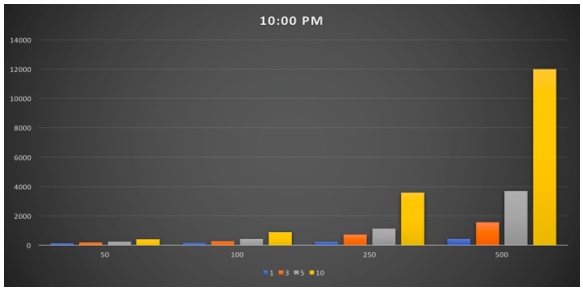


Figure 10 Multiple Request Per Device Stress Test

6. Conclusion

Eventually, we built a real-time monitor system, which supports several cameras at different locations at the apartment. For the limit of budget, we have only one running camera, but the whole system is proven to work and be extendable.

The latency test mainly shows that first, we could improve our software system performance in the evening. And second, the latency between the pi and the EC2 server really take up a lot of whole latency. To enhance the system performance for future development, we could work on this part to reduce this latency.

7. Reference

[1] Wikipedia, “Raspberry Pi” [Online]. Available: https://en.wikipedia.org/wiki/Raspberry_Pi

[2] Raspberrypi Official Website, “Camera Module” [Online]. Available: https://www.raspberrypi.org/products/camera-module

[3] Wikipedia, “Amazon Elastic Compute Cloud” [Online]. Available: https://en.wikipedia.org/wiki/Amazon_Elastic_Compute_Cloud

[4] comScore, “2016 U.S. Smartphone Subscriber Market Share” [Online]. Available: https://www.comscore.com/Insights/Rankings/comScore-Reports-January-2016-US-Smartphone-Subscriber-Market-Share

[5] Node JS Official Website, “APIs documents” [Online]. Available: https://nodejs.org/en/docs/

[6] Raspberrypi Official Website, “Raspbian” [Online]. Available: https://www.raspberrypi.org/downloads/raspbian/

[7] Ffmpeg Official Website, “Documentation” [Online]. Available: https://www.ffmpeg.org/documentation.html

[8] Linux Ping Manual, “Documentation” [Online]. Available: https://linux.die.net/man/