The Current Condition of the Internet of Things

Alexander Oser
MS in Computer Science
Washington University in St. Louis
St. Louis, MO 63130
osera@wustl.edu

Arivan Thillaikumaran
MS in Computer Science
Washington University in St. Louis
St. Louis, MO 63130
a.thillaikumaran@wustl.edu

ABSTRACT

A real-time system has been described as one which controls an environment by receiving data, processing it, and returning the results sufficiently quick enough to affect the environment at that time. We use this idea and the suite of Amazon Web Services to create a mirror that responds to events in its environment and meets specified time constraints. Our mirror displays weather and news on its screen on command, can take pictures and send the pictures to the user via text, and increases home security by alerting the user of an intruder's presence. Through different latency tests, we desired to ascertain the ability of Amazon Web Services to act as a host for a complete Internet of Things device and thus describe the current ability of cloud computing to enable widespread use of Internet of Things devices.

1. INTRODUCTION

Cloud computing provides a shared pool of configurable computing resources to end users on demand. This allows users to avoid upfront infrastructure costs, such as purchasing servers and private networks. Different service models exist and in this paper we concentrate on the model called platform as a service. Cloud providers deliver computing platforms, which include virtual operating systems, middleware, databases, web servers, and much more. With platform as a service, application developers can develop and run the specific software solution without having to deal with the costs for configuring the underlying system.

1.1 Internet of Things

The Internet of Things (IoT) is an example of an application of Platform as a Service; it is a network of networks of physical devices that are embedded with computers, sensors, and network connectivity that allow them to exchange and collect data in real time.

1.2 Amazon Web Services

Amazon Web Services (AWS) offers a suite of cloud computing services, spanning areas such as computing, storage, networking, databasing, analytics, application services, and developer tools for Internet of Things.

Amazon Web Services' dominance in the current cloud computing market made it an ideal choice for our project and we thus take analysis of latency through the AWS platform as an approximation of the overall current state of cloud computing. Furthermore, the wide array of services offered by AWS made linking the front-end and back-end of our project much simpler.

We will briefly describe each of the Amazon Web Services used.

Alexa Voice Services (AVS)
Amazon AVS allows developers to add intelligent voice control to any product with an internet connection, a microphone and a speaker.

Alexa Skills Kit (ASK)
Amazon ASK is a collection of APIs that adds powerful capabilities to any connected device and facilitates the creation of new skills.

Lambda
Amazon Lambda makes cloud computing easy and cost efficient by only charging users for the amount of time the computation takes and automatically managing the required servers. Lambda functions can be set to be triggered on certain events, such as an upload to an Amazon S3 bucket or an addition to an Amazon DynamoDB table.

Internet of Things (IoT)
Amazon IoT provides a cloud platform that lets connected devices easily and securely interact and exchange information.

Simple Storage Service (S3)
Amazon S3 makes it simple to store and retrieve large amounts of data in the cloud and allows easy access to this data between other AWS services.

1.3 The Mirror

Our goal with this project was to test the current capabilities of IoT to see if IoT products are truly in a position to take over the marketplace. In order to accomplish this we decided to build a Smart Mirror that takes advantage of AWS to make people's lives easier. It accomplishes this goal by (1) displaying the time, date, weather and news for its user, (2) connecting to the existing suite of Alexa skills, and (3) providing a unique skill

designed to make the mirror itself a valuable and unique product.

Alexa skills are based on different "intents" which control Alexa's response and our skills has three primary intents.: One, the TakePhoto intent, triggers a listener on the Raspberry Pi connected to the mirror, prompting it to take a photo and then uses AWS to send it to the owner via a MMS message. The other two intents are EnterSecurityMode and ExitSecurityMode and as their names suggest control the activation and deactivation of the mirror's security mode. In security mode, the Pi will take a picture of the room every minute, and again with the help of AWS, send the picture to the owner via MMS message if it detects motion between the current picture and the one most recently sent.

In order to show that our application is a real-time system, we must realize that the mirror (technically, the Raspberry Pi) has to be able to interact with Amazon Web Services and send the user the image in a reasonable time frame; for example, if the mirror detects a change in the room, but the owner only receives the picture five minutes later, then there isn't much that the owner can do to make an impact on the situation. We perform latency tests to ensure that our application meets the deadlines that we desire, and go into more detail in Section 3.

1.4 Project Goals

During this project we set intermittent goals for ourselves to keep in line with our desired progress and we will outline those goals below.

I. *First Milestone*
   A. Construct physical mirror
   B. Setup Raspberry Pi
   C. Create front end interface for the mirror pulling information from available APIs
   D. Connect Pi to AWS and set up a simple MQTT publisher subscriber service

II. *Second Milestone*
   . Use Lambda to connect the Pi to Alexa and respond correctly to Alexa's response
   A. Begin development of custom skill in the Alexa Skills Kit
   B. Set up S3 buckets and experiment with bucket interaction

III. *Final Milestone*
   . Finish implementing security skill
   A. Use security skill to test the latency of the AWS suite
   B. Find bottlenecks in project stack

2. Design and Implementation

2.1 Hardware

In order to create the 'Smart Mirror' effect, we placed a piece of two-way glass over a computer monitor. Two-way glass reflects light that hits it from the brighter side and by displaying a black webpage with white text on it, information can be displayed to the user overlapped with their own reflection. A Raspberry Pi Zero (v1.3) was used to control the display of the mirror and to function as the audio input and the 8 megapixel Raspberry Pi Camera Board (v2) was used to take photos.

2.2 Software

Almost every part of the mirror is made possible through AWS and in this section we will describe the complete flow of information as is depicted in the figure below.

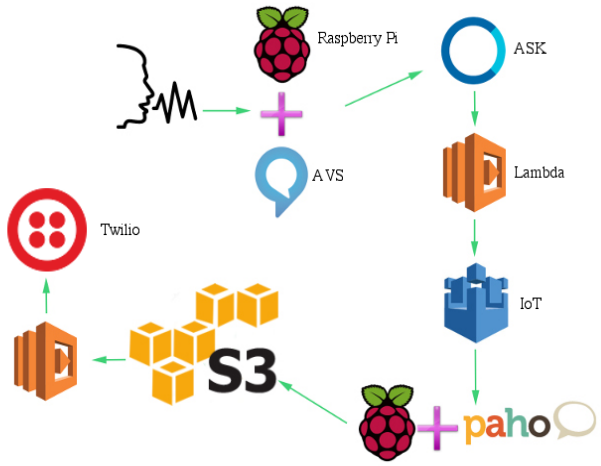

Figure 1: Smart Mirror flow diagram

The webpage which is displayed as the background of the mirror is hosted on AWS EC2. This page periodically pulls information from Reddit.com (for news stories) and OpenWeatherMap.org (for weather updates).

While the display itself is quite useful, the mirror has access to much more information through Alexa. Alexa is Amazon's voice assistant and any developer is allowed to add skills to Alexa, which has resulted in the production of hundreds of free skills that are currently available. In order to take advantage of these skills, we connected a microphone to our Raspberry Pi and established a secure connection to AVS. The microphone can then record a request which is sent to AVS from the Pi. AVS interprets the request and directs the transcribed audio towards the corresponding skill in ASK. Once selected, the appropriate skill uses a Lambda function to construct Alexa's response which is sent back to the Raspberry Pi and recited by Alexa. Our skill, in addition to constructing Alexa's response, publishes a message using AWS IoT as our MQTT broker. The Raspberry Pi is also connected to AWS IoT and uses a Paho client to subscribe to certain messages from the IoT broker. So, when the Lambda function interprets the request from ASK as either one to take a photo or enter security mode, a corresponding message is published. The subscriber on the Pi reacts to each incoming message and either takes a single photograph or enters a loop, taking periodic photos for the security mode. The Pi Zero does not have the computing power to detect motion

between two photos however, so the photos are immediately uploaded to S3. We have two buckets set up in Amazon S3, bucket A and bucket B. Upon upload to bucket A, a Lambda function is triggered. The photo that is uploaded gets put into bucket A, and if there is no picture in bucket B, then this was the very first picture taken, and there is no picture to compare it to for motion detection, and the photo that is uploaded to A gets uploaded to B, and finally, the photo in bucket A is deleted, all done by the Lambda function associated with bucket A. Next, the second photo gets uploaded to Amazon S3. Now, as the photo is uploaded to bucket A, the Lambda function checks and sees that there is indeed a photo in bucket B, and proceeds to run these two photos through the motion detection algorithm. Now, if motion is detected, the photo that was uploaded to bucket A will be sent to the Twilio client, which will then send that picture to the user via MMS message. If motion was not detected, then the photo that was uploaded to bucket A will be uploaded to bucket B with the same key name, so that photo in bucket B is overwritten. In this manner, photos are checked for motion detection, and bucket B only holds the most recently taken photo.

There are many, many ways to perform motion detection, tracking, and analysis. We use Python's OpenCV library to perform our motion detection. We initially resize the image to have a frame with width of 500 pixels, and convert it to gray scale since color has no bearing on our motion detection algorithm. We also apply Gaussian blurring to smooth our images. We do this because due to tiny variations in digital camera sensors, no two frames will 100% be the same, and some pixels will for sure have different intensity values. The Gaussian blurring helps smooth out high frequency noise that could throw our motion detection algorithm off. We compute the difference between two frames via simple subtraction, taking the absolute value of their corresponding pixel intensity differences. We set set a threshold value for changes in the frames to only reveal regions of the image that have *significant* changes in pixel intensity values. Given a threshold image, it's simple to apply contour detection to find the outlines of these regions. If the contour area is larger than the specified min area (in our case, 500,) then algorithm returns "Motion Detected."

3. Experiments and Analysis

In order for the main aspect of our application, the security mode, to be feasible, the user must receive a picture from the mirror in a reasonable amount of time. Since our mirror takes a picture once a minute, and to our knowledge there is no industry benchmark to compare our results to, we set a goal of 30 seconds. We feel that for the image file to travel from the Raspberry Pi to the Amazon Cloud, go through a motion detection algorithm in the cloud, be sent to Twilio services which then sends the MMS message over a mobile network service, thirty seconds is a reasonable round-trip latency.

As mentioned before, our application travels through various different modules in order achieve its goal; thus we can measure end-to-end latency and also isolate the time that the payload travels through each model. We use the help of Amazon Cloudwatch and a classic stopwatch in order to make our measurements. Amazon Cloudwatch is a monitoring service for AWS cloud resources and the applications that we run on AWS. Amazon Cloudwatch allows us to collect and track metrics, collect and monitor log files, set alarms, and automatically react to changes in our AWS resources. We use the log files that are provided by Cloudwatch in order to determine when each module of the stack is triggered.

22:02:59	START RequestId: 93de8292-bb36-11e6-8b8c-97bc80222020 Version: $LATEST
22:03:01	First picture, uploading to 2nd bucket
22:03:01	END RequestId: 93de8292-bb36-11e6-8b8c-97bc80222020
22:03:01	REPORT RequestId: 93de8292-bb36-11e6-8b8c-97bc80222020 Duration: 1777.39 ms Billed Duration: 1800 ms
22:04:00	START RequestId: b9dc2875-bb36-11e6-9071-9bf49512c9e9 Version: $LATEST
22:04:00	not first pic; downloaded pic
22:04:01	downloaded most recent pic, beginning motion detection
22:04:05	success frame1
22:04:05	success frame2
22:04:05	gray works
22:04:05	thresh work
22:04:05	got here
22:04:05	Motion Detected
22:04:06	pic is public :)
22:04:06	message sent yo
22:04:07	uploaded most recent pic to bucket
22:04:07	END RequestId: b9dc2875-bb36-11e6-9071-9bf49512c9e9
22:04:07	REPORT RequestId: b9dc2875-bb36-11e6-9071-9bf49512c9e9 Duration: 7203.57 ms Billed Duration: 7300 ms

Figure 2: An example log file from Amazon Cloudwatch

One of these log files is created for each Lambda function that we have implemented, every time that they are run. Thus with console logs in our code, we can isolate every module of our code due to the timestamps that are returned on the left side, as can be seen above. Every Lambda function request also ends with a report message in the log, shown in more detail below.

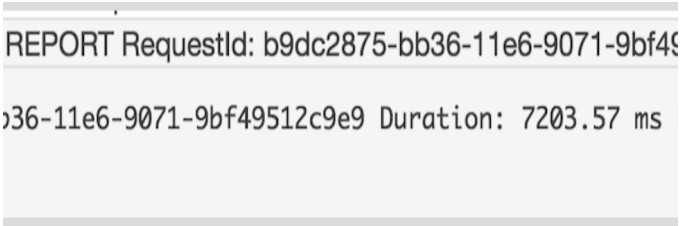

Figure 3: A report message shown in more detail. This particular report message was given by the Lambda Function that is triggered when the mirror is in security mode and sends a picture up to Amazon S3; thus it is the time from when a picture is uploaded to Amazon S3 until the time that the message is sent out by Twilio (not the time that the user receives the message.) In this example, this time is reported as 7.203 seconds.

In order to test the end-to-end latency of the process, as well as look at each module for potential bottlenecks, we conducted the following experiment. To measure overall

latency we used a standard stopwatch to measure the time from issuing the "Take a photo" command to AVS to receiving the resulting MMS message from Twilio. We then used the log files provided by Lambda to isolate exactly when the payload is reaching different AWS services, and how long it spends in each service (as shown in the report message above.)

In order to see how the payload size would affect processing time, we experimented with three different image qualities. The default image setting with 100% quality created images that were between four and five megabytes and with this setting we found that our results were not close to our goal. Over five different test runs the average latency was 40.12 seconds, with the shortest trip taking 33.4 seconds, and the longest 55.6 seconds. We then decided to tweak the image quality in order to try to improve the performance of our application. At 10% of the maximum quality, with photos between .7 and 1.5 megabytes, we got much more promising results. However, they still did not meet our goal with an average latency of 32.5 seconds, without too much fluctuation. Finally, when we set the quality of the image to 5% of the maximum the photo was reduced to around half a megabyte and the latency decreased to an average time of 27.66 seconds, reaching our goal.

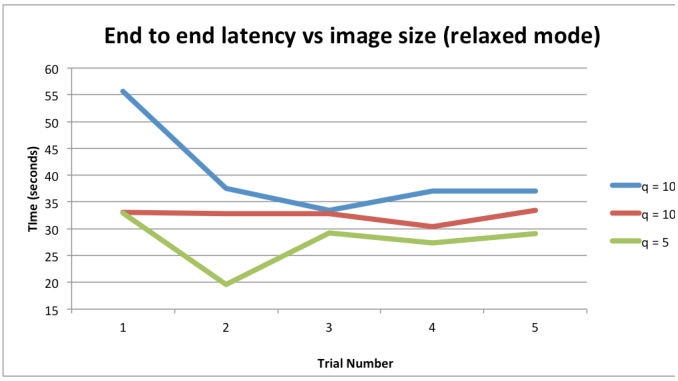

Figure 4: Results from each run of our experiment with q representing the percentage quality of the photo

Table 1: Raw data from latency experiments

Overall Selfie Latency Tests	q = 100					
Total Time	**55.6**	**37.5**	**33.4**	**37.1**	**37**	**40.12**
Alexa request to response	3.2	4.1	3.1	3.1	3.1	
Twilio Processing	0.9	0.7	0.8	0.7	0.7	
Camera speed	5	5	5	5	5	
Cell service latency	22	19	20	19	19	
	31.1	28.8	28.9	27.8	27.8	
Overall Selfie Latency Tests	q = 10					
Total Time	**33.1**	**32.8**	**32.8**	**30.4**	**33.4**	**32.5**
Alexa request to response	6.1	3.2	4.2	3.1	4.1	
Twilio Processing	1	0.8	0.7	1.2	0.9	
Camera speed	5	5	5	5	5	
Cell service latency	20	21	20	19	21	
	32.1	**30**	**29.9**	**28.3**	**31**	
Overall Selfie Latency Tests	q = 5					
Total Time	**33**	**19.6**	**29.2**	**27.4**	**29.1**	**27.66**
Alexa request to response	6.1	0.4	3.2	4.1	3.1	
Twilio Processing	0.9	1.3	1.1	0.8	0.9	
Camera speed	5	5	5	5	5	
Cell service latency	19	10	18	10	18	
	31	16.7	27.3	19.9	27	

We also tested our application with a picture quality of 1%, and were able to reduce the overall latency to 20 seconds. However, such a drastic reduction in quality led to almost unrecognizable photos. This lack of quality is obviously not ideal for the end user who simply wanted their photo taken, but also led to issues with our motion detection algorithm. While the previous experiment was conducted using the TakePhoto intent, we also tested each quality level with the SecurityMode functionality, and found that at 1% quality the blurriness of the photos led to the algorithm detecting motion for every single picture analyzed, even for what should have been identical pictures. Therefore we determined that we must sacrifice some speed in order to allow the motion detection algorithm to function correctly.

We next tested the latency of the additional lambda function that the application would trigger if security mode is active. This Lambda function consists of the time from photo upload to the moment that Twilio sends the text message (not necessarily the moment the user receives the text message.) As we adjust the quality of the image, we do find that the overall time decreases, as we predicted before when we tested the TakePhoto function separately. We then specifically look at the time that the motion detection method took during each trial, and confirmed that the image quality did have an impact on how fast the motion detection algorithm ran. These are shown in Figures 5 & 6 below.

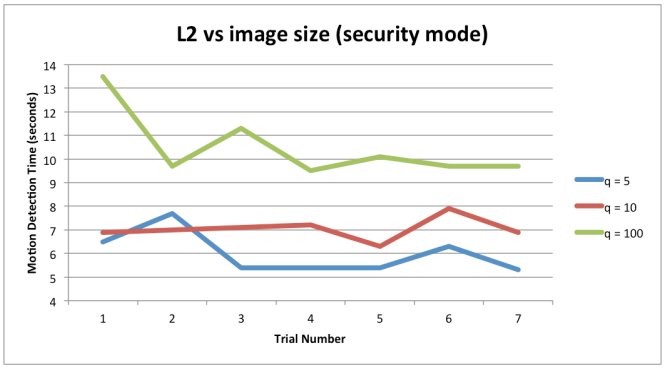

Figure 5: Lambda function (triggered when photo is uploaded to bucket in security mode) vs image size

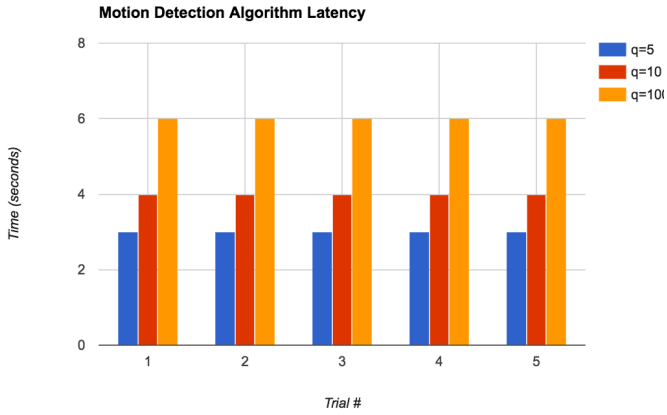

Figure 6: Motion Detection Algorithm latency vs image quality

4. Lessons Learned and Future Works

One of the biggest takeaways from the project was both how powerful and limiting the Raspberry Pi Zero was. We also learned what it takes to deploy an application on Amazon Web Services; using all the different services and tying them together was not exactly intuitive and took some time to wrap our heads around. However, as pieces started to fall together, the overall system made much more sense, and we were able to see it from an interactive, real-time system perspective.

All our work was done on a Raspberry Pi Zero. The Pi Zero is notoriously slow, with the minimum processing requirements needed to be able to run. If our application were built on a more powerful Raspberry Pi, end-to-end latency could potentially be improved and the motion detection algorithm could run on the Pi instead of in the cloud. We could also potentially run more effective motion detection algorithms; the two primary methods are forms of Gaussian Mixture Model-based foreground and background segmentation. Most of these methods are concerned with segmenting the background from the foreground (and Python's OpenCV even provides mechanisms to discern between actual motion and just shadowing and small lighting changes.) However, while powerful, these methods are computationally expensive, and would not be effective on the Raspberry Pi Zero. Also, as we mentioned before, a Raspberry Pi image of quality zero was not able to analyzed correctly by the way our motion detection algorithm is currently implemented. One of these more advanced motion detection techniques could be employed to potentially be able to analyze a Raspberry Pi image of quality value one, and thus would be the optimum size image to send as the cell service latency goes down as the size of the image goes down.

Amazon Web Services provides a Simple Notification Service; however, we chose to use Twilio mainly because this Simple Notification Service does not provide MMS messaging capabilities. Therefore, we believe that latency issues could be additionally be improved if Amazon adds MMS messaging capabilities to their notification service, eliminating the need for a third party messaging service.

If users wanted the mirror to take pictures more frequently than every minute, then Amazon Queueing Service can be used. As the overall end-to-end latency of our application is seen to be around 30 seconds, having the mirror take pictures that frequently, or even more frequently, would introduce confusion to the algorithm, as images in the buckets may be deleted by other calls to the Lambda Function while another thread is still utilizing that image. To avoid this, Amazon Queue Service will be able to queue the images that are uploaded to the bucket, but this will definitely add more overhead to the application and probably introduce more latency.

5. Conclusion

Amazon Web Services provides a very effective cloud-computing platform which is fully capable of producing innovative Internet of Things products. However, we came to realize that while AWS is ready, not every platform is as prepared. As we saw, the primary bottleneck in our application came from cell service latency, causing over half of the delay from Alexa to the user's cell phone. So where does the Internet of Things currently stand? Well, it is clear that many applications of it are already beginning to appear in our everyday lives, but until all necessary services, such as WiFi speed and cell service are improved, it is unlikely that they will become present in all parts of society.

6. Related Work

To our knowledge, there is no such "Smart-Mirror", voice controlled, that is on the market right now. We have seen some mirrors that display weather and news API's, but none that are voice controlled, or take picture and detect motion. However, as we are in the golden age of Internet of Things, many different IoT applications are being created, especially home surveillance/improvement products similar to our mirror, and it is only a matter of time before a something similar to our project is created. In the space of

Internet of Things applications, as an example, Nest, recently bought by Google, creates programmable, self-learning, sensor-driven thermostats, smoke detectors, and other security related systems. This addition of machine learning techniques to real-time cloud computing is another exciting facet of the deployment of real-time systems, and one that could definitely be applied to our application.

7. References

[1] Cannistra, Mario. Python and Paho MQTT for AWS IoT. *Hackster.io*. https://www.hackster.io/mariocannistra/python-and-paho-for-mqtt-with-aws-iot-921e41

[2] AWS Docs. "Tutorial: Using AWS Lambda with Amazon S3." http://docs.aws.amazon.com/lambda/latest/dg/with-s3-example.html

[3] Lu, Chenyang. CSE 520S Lecture Slides Fall 2016.

.