IoT Application using Amazon Web Services IoT, Dynamo DB and Lambda services : Academic Project for CSE661

Ravichandra Malapati*
SUID: 22375-2155
Syracuse University, NewYork
United States-13210
rmalapat@syr.edu

Dr. Richard Tang†
Syracuse University
Syracuse, New york
United States-13210
ytang@syr.edu

Instructions to execute To successfully execute the this project you need to install openssl in ubuntu and MQTT protocol in ubuntu, after that go to folder /ACA_Proj/ Application/ subscribe_publish_ACA_Project and execute ./subscribe_publish_sample

ABSTRACT

Amazon Web services has released its latest version of Amazon Web Services Internet of Things with beta version, as part of the academic project, i select to setup a *MQTT* broker which acts as a server in AWS IoT. AWS IoT's ability to integrate with existing machine learning application from Amazon Web Services and the Databases like Dynamo DB coupled with the lambda service of the AWS give a great advantage to connect to wide range of application. For example A user can connect an IoT node which can be either a small embedded hardware device or a high performance computer to a twitter account or some other Internet applications.

Amazon AWS IoT also provides flexibility to scale the number of IoT from 0 to some thousands without no extra cost just by scaling up the number of connections in MQTT broker. AWS IoT also provides better security with X509.certificate, device authentication certificate and private key certificate. The experiments are conducted to test the AWS IoT suite and record the values with speed and scalability compared to the locally set MQTT broker. We explore the Mosquitto Mqtt broker, eclipse paho client and mosquitto publisher and subscriber tool in this project.

1. REQUIREMENT

To develop a prototype for IoT nodes which can send the sensor data to a centralized location. To notify users when an event happens in the IoT node. Compare the performance of the Mosquitto broker in the local machine and Mosquitto broker in the AWS IoT. As part of this project we demonstrate three AWS components like AWS IoT suite, AWS DynamoDB, AWS Lambda services. Analysis of how the AWS IoT can be used with AWS services like notification services etc. will be discussed.

*First Author
†Instructor

2. INTRODUCTION

Internet of Things is the word that has been heard by technology enthusiasts for almost 15 years. Even though, technology is readily available for implementation of IoT. The lack of data storage technologies and Internet connectivity are hindering the further development of IoT products. Recently Amazon released its first version of AWS IoT cloud suite and followed by Microsoft's version of Azure IoT cloud suite. Google, HP and other cloud service providers are already there in the field for quite some time. With over 50 billion devices to be connected with cloud by 2020 and the data that will be generated and the network bandwidth usage will be colossal. The applications of IoT are booming and will be used in every field. For example, smart homes, industrial automation, automotive, self-driving cars, medical appliances, social media, agriculture, environmental monitoring etc. collecting and securing such mammoth volume of private data transmission over networks is going to be a big challenge for cloud service providers.

The architects who design the cloud systems face the challenge of building applications that work within the Internet of Things and its devices. With the new PaaS architecture of AWS its quite possible to accomplish these challenges and help to ease the challenges and reduce the complexities. Though the cloud services for the Internet of Things is quite new and still in starting phase, managers and architects are learning at the fast phase. Since the market is new and the requirements are huge it is very challenging to divide the services and offer to the end customer. The development of best cloud based architecture depends on how loosely coupled are the application and services.

The IoT is the domain where the wide variety of applications come from for example, there may be real time applications where the devices want to stream their data in to the cloud as and when the data is generated in the IoT device. There are also applications which require just data storage in cloud, for example applications where the IoT devices have less amount of memory and will update the cloud database as scheduled. There are also applications which require real

time data analytics where the live data is streamed from all the IoT devices and data analytics is done in the cloud and the results are updated to the end customer. It is said that no single data base can solve all the IoT problems and hence there will be continuous development of cloud data bases for each use case in future once IoT is matured.

Keeping all these in mind the architect and project manager needs to design the cloud architecture where each service can be provided as a service and the customer can pay as per the usage. Especially the model of pay as you use has been successful and hence AWS IoT is designed keeping this in mind. In our project i discuss various features of the AWS IoT and how they can be used. And also i show how i can set up basic test bed for testing the AWS IoT using AWS IoT suite. MQTT provides user-name/password and SSL/TSL based authentication to secure IoT devices. We setup AWS IoT MQTT broker in section and use this to demonstrate how efficiently we can use AWS IoT rather than using the self developed MQTT broker locally. Designing the

3. REVIEW OF CLOUD SERVICES FOR IOT

I have studied the design documents of the Google Cloud services, Microsoft Azure and AWS IoT and found that only AWS IoT broker is best and well structured. We will provide a brief overview of all the three services below before we go further. The architecture of the AWS IoT best suits the IoT applications and hence we will use AWS IoT as part of this project.

3.1 Microsoft Azure for IoT

IoT Hub is the new entry in the Microsoft Azure offer; it's a service that enables bi-directional communication between devices and business engine in the cloud. The communication channel is reliable and secure and the authentication is per-device using credentials and access control. Thanks to its bi-directional nature, the messages between devices and cloud travel in both directions along the established channel. Each device has two endpoints to interact with IoT.

3.2 Google Cloud for IoT

From my study on Google cloud i see that Google supports the cloud based services for IoT but not in efficient way and the quality of service is not very good. The latency is high for the IoT cloud services. However there is no SDK support from Google for IoT services as of now. And also the Google IoT cloud services are not yet released officially. Google database support is not much appreciable for the IoT applications as it takes minimum 2 minutes to upload the data into the database and it takes around 90 minutes for the data become available for the export and import applications which means the real time operations can not be performed if we use google cloud services in the IoT applications. being said that the Google IoT cloud services can not provide the real time streaming of the data and it is way behind when compared to the AWS IoT suite.

3.3 AWS IoT suite for IoT

The AWS IoT suite beta version provides secure, bi-directional publish/subscribe communication between Internet-connected things (such as sensors, actuators, embedded devices, or smart appliances) and the AWS cloud. This enables the IoT applications to collect telemetry data from multiple devices and store, analyze the data.The AWS also provides the SDK to develop applications that enable IoT users to control IoT devices from their phones or tablets.

The best of the three major cloud service provides we have mentioned in the introduction of this document is AWS IoT, it provides real time streaming of the data. The AWS IoT cloud also provide the lambdas to trigger real time notifications. The machine learning applications can be easily integrated in the AWS IoT suite. Amazon Machine Learning is based on the same proven, highly scalable, machine learning technology used for years by Amazon's internal data scientist community. The service uses powerful algorithms to create ML models by finding patterns in the existing data. Then, Amazon Machine Learning tools uses the defined models to process new data and generate predictions for IoT applications. The main advantage with the AWS IoT is that it provides the security feature which comprises of three certificates using TSL1.2 or higher certificate version.

Device End Point: Device to Cloud.

The device can use this endpoint to send MQTT messages to the Microsoft cloud as telemetry data, outcome for a received command or request for execution etc

Device End Point: Cloud to Device.

The device receives commands on this endpoint for executing the requested action. The IoT Hub generates a feedback at application level to confirm that the command is acquired by the device and it's going to be executed.

Cloud End Point: Device to Cloud.

it's an Event Hubs compatible endpoint used by the back end system to retrieve messages from device (telemetry data or outcome for commands). On a different path, there are feedbacks on command delivery (successful or not). Event Hubs compatible means that we can use an Event Hub client to receive messages from this endpoint.

4. PROPOSED EXPERIMENT

As part of this academic project we develop a IoT client application using the AWS IoT SDK in C language. We use this application to publish and subscribe to the IoT data requests from the MQTT broker. We also set up MQTT broker in both AWS IoT cloud as well as in the local system. Now using the client IoT MQTT application we communicate with the locally setup MQTT broker and also with the AWS IoT MQTT broker. We compare the results at the end to compare the performance of the MQTT broker in AWS IoT as well as in the local system.

We also use the dynamo DB of the AWS to shohw that the stream of data can be stored in the cloud and later can bse used to perform machine learning applications.

As a server we user AWS IoT suite Mosquitto broker and also locally setup mosquitto broker for comparison. The database services are provided by the Dynamo Database while the event notification services are provided by the Lambda services of the AWS IoT lambda. while many other services can be integrated into this project, since it is academic project we keep it short for three services. The project is explained in the below paragraphs in detail.

The Mosquitto broker is set up in the AWS IoTserver, The paho client and Mosquitto MQTT client are set up in

Figure 1: Microsoft Azure reference IoT architecture

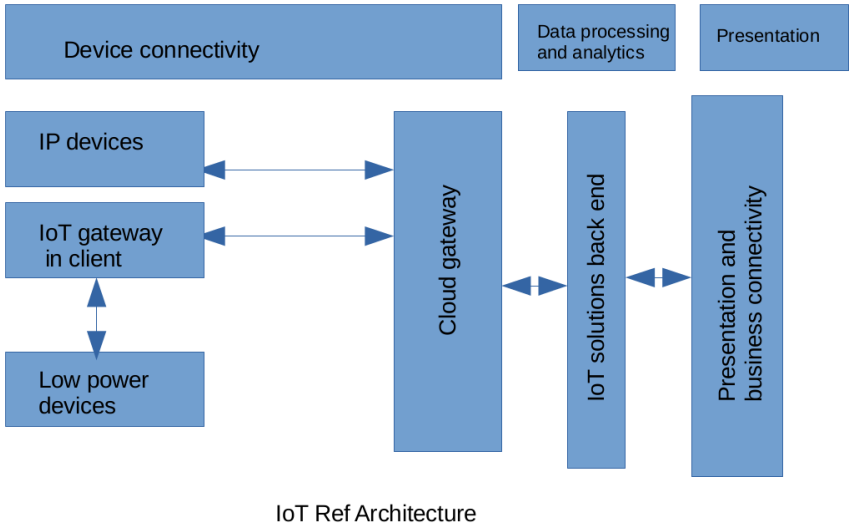


Table 1: Comparision of AWS IoT and Local system

Name of specification	**Cloud System Configuration**	**Local system configuration**
Server	AWS IoT Cloud	Thinkpad Laptop
Memory	Unlimited	8GB
Processor	Unlimited speed and scalability	2.5 GHz
Mosquitto broker	MQTT broker 1.4.4	MQTT broker 1.4.4
MQTT protocol Version	3.1	3.1
Client	IoT application in C	IoT application in C
Operating system	Linux	Linux

Table 2: Expriement specification with message

Specification	**SP/SC**	**SP/MC**	**MP/SC**	**MP/MC**
Client Message Rate*(Message/Second/per client)*	1000	1000	1000	1000
Client Message size*(bytes)*	100	100	100	100
No of publisher clients	1-100	1-100	1-100	1-100
No of subscriber clients	1-100	1-100	1-100	1-100
No of publisher clients	1	600	3	600
No of subscriber clients	1	600	3	600
No of paths	1	1	3	3

a b c

[a]This wil get executed for both Local server and AWS IoT server
[b]SP/SC-Single Path Single client, SP/MC-Single Path, Multiple client
[c]MP/SC-Multiple Path Single client, MP/MC-Multiple Path, Multiple client

the local system. the applications can be developed in either C, Java script for rapid prototyping. The following are the components that are present in the AWS IoT we describe briefly about these services below.

4.1 AWS IoT Design

AWS IoT enables Internet-connected things to connect to the AWS cloud and lets applications in the cloud interact with Internet-connected things. Common IoT applications either collect and process telemetry from devices or enable users to control a device remotely

Things report their state by sending messages, in JSON format, to MQTT topics. Each MQTT topic has a hierarchical name, such as ”myhouse/livingroom/temperature.” The message broker sends each message received by a topic to all the clients subscribed to the topic.

You can create rules that define one or more actions to perform based on the data in a message. For example, you

can insert, update, or query a DynamoDB table or invoke a Lambda function. Rules use expressions to filter messages. When a rule matches a message, the rules engine invokes the action using the selected properties.You can use all JSON properties in a message or only the properties you need. Rules also contain an IAM role that grants AWS IoT permission to the AWS resources used to perform the action.

Each thing has a thing shadow that stores and retrieves state information. Each item in the state information has two entries: the state last reported by the thing and the desired state requested by an application. An application can request the current state information for a thing. The shadow responds to the request by providing a JSON document with the state information (both reported and desired), meta data, and a version number. An application can control a thing by requesting a change in its state.The shadow accepts the state change request, updates its state information, and sends a message to indicate the state information has been updated. The thing receives the message, changes its state, and then reports its new state.

4.1.1 Message broker as server

The MQTT broker is the server to which the clients publish messages using a topic name and also subscribs to a topic the clients are interested. Message broker provides a secure mechanism for things and IoT applications to publish and receive messages from each other.MQTT protocol can be used to publish and subscribe. The SDK is provided by the AWS IoT to develop applications to the clients.

Message broker or MQTT Broker [1] is the server with which the things(IoT devices) in the internet can publish and subscribe to the topics. The MQTT broker in AWS is responsible for receiving the published messages, filtering them and identify who is interested and sending the messages to the interested subscribers. The MQTT broker is also responsible for holding the sessions of all the connected clients. One more responsibility of the broker is authentication and authorization of the clients.

There are many versions of the Message brokers available in the market as prototypes but we use in this project the genuine Mosquitto Mqtt broker.

4.1.2 Thing Registry

Thing Registry is also referred as Device registry which organizes the resources associated with each Internet thing registered with AWS IoT. We register all devices as things and associate up to three custom attributes(certificates) with each thing. We can also associate certificates as mentioned earlier and MQTT client IDs with each thing to improve the ability to manage and troubleshoot the things.

4.1.3 Rules engine

Rules Engine provides message processing and integration with other AWS services. The rules engine helps to curtail the access by roles in the AWS IoT. we can use a SQL-based language to select data from message payloads, process the data, and send the data to other services, such as Amazon S3, Amazon DynamoDB, and AWS Lambda. The following is a sample SQL command used in this project to send data from AWS IoT to dynamo DB and Lambda.

```
SELECT * FROM 'topic/lightravi'
```

```
JSON data:
```

```
{
  "all": "1449782096631",
  "deviceID": "lightravi",
  "payload": {
    "sensor_Key": "Sensor Data"
  }
}
```

We can also use the message broker to republish messages to other subscribers.

4.1.4 Thing Shadows service

Things shadows service provides persistent representations of Internet of things in the AWS cloud. It also allows us to publish, subscribe, updated state information to a thing shadow, and the thing can synchronize its state when it connects.The IoT devices can also programed to publish their current state to a thing shadow for use by applications or devices.

Thing Shadows.
Sometimes referred to as a device shadow. A JSON document used to store and retrieve current state information for a thing.

Device Gateway.
Enables devices to securely and efficiently communicate with AWS IoT.

Security and identity service.
Provides shared responsibility for security in the AWS cloud.Your things must keep their credentials safe in order to send data securely to the message broker. The message broker and rules engine use AWS security features to send data securely to devices or other AWS services.

AWS Command Line Interface (AWS CLI).
Run commands for AWS IoT on Windows, Mac, and Linux. To get started, see the AWS Command Line Interface User Guide. For more information about the commands for AWS IoT, see iot in the AWS Command Line Interface Reference.

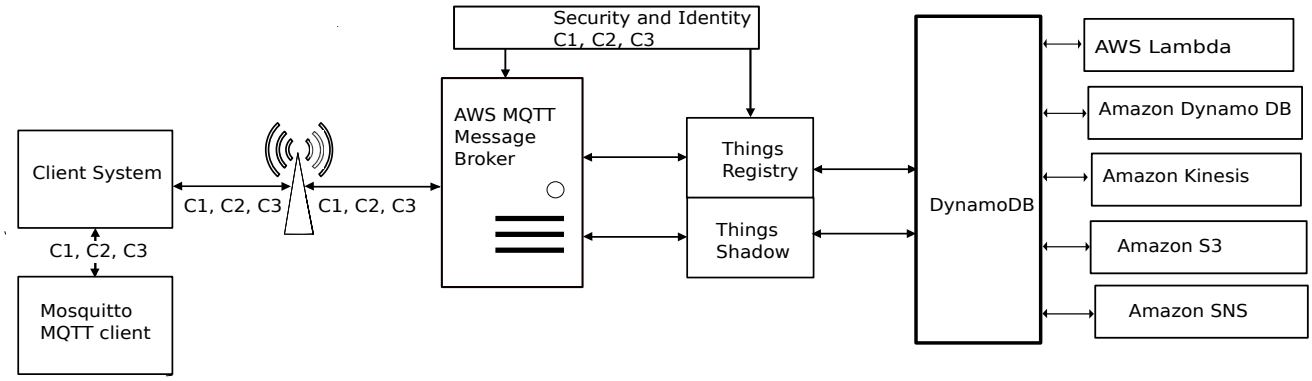
AWS SDKs.
Build the IoT applications using language-specific APIs.

AWS IoT Thing SDK for C.
Build IoT applications for resource-constrained things, such as micro controllers.

4.2 Setting up Mosquitto MQTT broker as server in local system

We set up Mosquitto MQTT broker 1.4.4 version running in Ubuntu operating system as our server. MQTT broker is configured to hold default number of queued messages as 100 which means the MQTT broker will not be able to hold the messages in buffer Queue more than 100 numbers. If the number of flooding messages make the queue full then the MQTT broker drops the messages. We also configure the maximum size limit for broker as default which means the MQTT broker will accept all the valid messages up to maximum payload size of 268435455 bytes. The security is

Figure 2: AWS*(Amazon Web Services)* IoT setup



a. C1, C2, C3 are the X.509 certificate, device certificate and device private key in .pem format b. Lightravi is created things registry c. Mosquitto MQTT client passes the certificates to AWS MQTT broker to successfully get authenticated and publish/subscribe to topics. d. Using wireshark the attacker extracts the end destination address

disabled as we assume that the certificates (C1, C2, C3 or user-name and password) are already available for the attacker. The listener port is set to default which is 1883 as prescribed by the Mosquitto.org. Apart from these configuration settings all other options are set to default.

5. PROJECT EXECUTION

We setup a local MQTT broker which acts as our target to demonstrate all the difference of performance betwen AWS IoT suite and locally set MQTT broker. The design of this system is discussed in detail in the section in this section. AWS IoT is set up using the instructions provided in the reference section [1].

To create a thing in the things registry, to create roles and policies AWS IoT command line interface should be installed. AWS IoT CLI provides variety of commands like create-policy, create-thing, create-thing, create-topic-rule etc. All these commands will be useful to create the the IoT things and execute instructions over them successfully in doing this project. To develop applications which can be embedded into the computers or low power embedded device AWS IoT SDK is used in this project. AWS IoT provides AWS IoT account is created and a thing *lightravi* is registered in the AWS IoT thing registry, when a thing is created in the AWS IoT suite it is just an isolated node which don't have any connections to the Internet and no actions can be performed over that thing. Now for granting access to lightravi a role has to be created. The role can be created using the following command.

```
1.  aws iot create-thing --thing-name "lightravi"
2.     {"thingARN": "arn:aws:iot:us-east-1:
aws-account-id:thing/lightravi",
    "thingName": "lightravi"
   }
```

5.1 Creating Certificate and keys for Security

AWS IoT provide both user generated certificate as well as auto generated certificates by the AWS IoT for authentication of the devices connecting to the AWS IoT. The following command is used to create the certificate and key required for the authentication while creating the device.

```
aws iot create-keys-and-certificate
--set-as-active --output text
```

The certificates create above are common for all the device which means any device that has MQTT client installed can connect to the AWS IoT with these certificates. This has been tested. To see the certificates that have been generated by the AWs IoT suite we can give the following command.

```
aws iot describe-certificate --certificate-id id
 --output text --query
 certificateDescription.certificatePem >cert.pem
```

The certificate we created should be now be given a access policy. The policy will specify the access allowed for the device who are holding this certificate. The policy shold be in JSON format and the same is described in the following commands.

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action":["iot:*"],
        "Resource": ["*"]
    }]
}
```

The above mentioned policy allows all the iot actions to be performed by the devices which hold the generated certificate.

Now that the certificates are created and assigned a policy which provide access to the iot tasks present in the AWS IoT suite, but the certificates now should be attached to the devices or things we created in the previous step. To do so we have saved the certificate in the previous step in the cert.pem file. now we read the cert.pem file back and attache to the device *lightravi* we created in the previous steps.

```
aws iot attach-thing-principal --thing-name
lightravi
 --principal
"certificate-arn"
```

5.2 Connecting to AWS IoT using IoT application

AWS IoT is set up now with a topic *lightravi* and certificates are generated to perform the IoT tasks in AWS IoT, now we create an application and install the MQTT broker in the local device which we treat as an IoT node.

As to keep simple we use a linux based operating system to connect to the AWS IoT MQTT broker. After installing the eclipse paho client [3] for MQTT protocol [7] in linux operating system.

We also download the AWS IoT SDK kit to connect to the AWS IoT.Before we develop the application we require the end point address of the AWS IoT the following is the CLI command used to get the end point address.

```
aws iot describe-endpoint
```

The above command provides the end destination address that can be used to publish and subscribe the messages to the AWS IoT broker. For developing the MQTT application for AWS IoT we specify the specifications of the system as in Table **??**

5.2.1 *setting up Dynamo DB*

We link dynamoDB with AWS IoT by creating a rule and policy as shown in picture 3dynamo DB collects the data from AWS IoT and the Lambda is invoked to perform actions whenever there is new update for a particular device or topic.

5.2.2 *Developing Applicaion*

First, we read the certificates we generated in the previous subsections. The certificates are read using the following message.

```
char rootCA[PATH_MAX + 1];
char clientCRT[PATH_MAX + 1];
char clientKey[PATH_MAX + 1];
char CurrentWD[PATH_MAX + 1];
char cafileName[] = AWS_IOT_ROOT_CA_FILENAME;
char clientCRTName[] = AWS_IOT_CERTIFICATE_FILENAME;
char clientKeyName[] = AWS_IOT_PRIVATE_KEY_FILENAME;
```

The following are the parameters that are specified for connecting to the AWS IoT MQTT broker.

```
connectParams.KeepAliveInterval_sec = 10;
connectParams.isCleansession = true;
connectParams.MQTTVersion = MQTT_3_1_1;
connectParams.pClientID = "CSDK-test-device";
connectParams.pHostURL = HostAddress;
connectParams.port = port;
connectParams.isWillMsgPresent = false;
connectParams.pRootCALocation = rootCA;
connectParams.pDeviceCertLocation = clientCRT;
connectParams.pDevicePrivateKeyLocation
= clientKey;
connectParams.mqttCommandTimeout_ms = 2000;
connectParams.tlsHandshakeTimeout_ms = 5000;
connectParams.isSSLHostnameVerify = true;
connectParams.disconnectHandler
= disconnectCallbackHandler;
```

The below is the code to read the configured destination address and creating the conection which we call as remote connection *rc*.

```
TLSParams.DestinationPort = pParams->port;
TLSParams.pDestinationURL = pParams->pHostURL;
TLSParams.pDeviceCertLocation =
pParams->pDeviceCertLocation;
TLSParams.pDevicePrivateKeyLocation
 = pParams->pDevicePrivateKeyLocation;
TLSParams.pRootCALocation
= pParams->pRootCALocation;
TLSParams.timeout_ms
= pParams->tlsHandshakeTimeout_ms;
TLSParams.ServerVerificationFlag
= pParams->isSSLHostnameVerify;
rc = iot_tls_connect(&n, TLSParams);
```

The application is developed in C language and can be ported any device like the Raspberry pi, Audrino etc. by changing the configuration, which is mentioned in the above code snippets.

6. PERFORMANCE EVALUATION OF AWS IOT MQTT BROKER AND LOCAL MQTT BROKER

Using the IoT application we developed in the previous section we compare the performance of locally setup MQTT Broker and the AWS IoT MQTT broker. Later we use the DynamoDB service to show how we can use the advantage of AWS IoT

The rest of this section will be as follows. First we will use the local MQTT broker and publish maximum number of the messages and check the delay and the latency. Later we will use the same application and publish to the Amazon AWS and see the latency of the request. once we do this we will compare the latency and see which is better. Later we will demonstrate how the AWS services such as DynamoDB and Lambda services can be effectively used to solve the notification problems.

We will explain the results of tests conducted and see the advantages of using the AWS cloud services rather than locally setup server. An we will conclude this paper about this project.

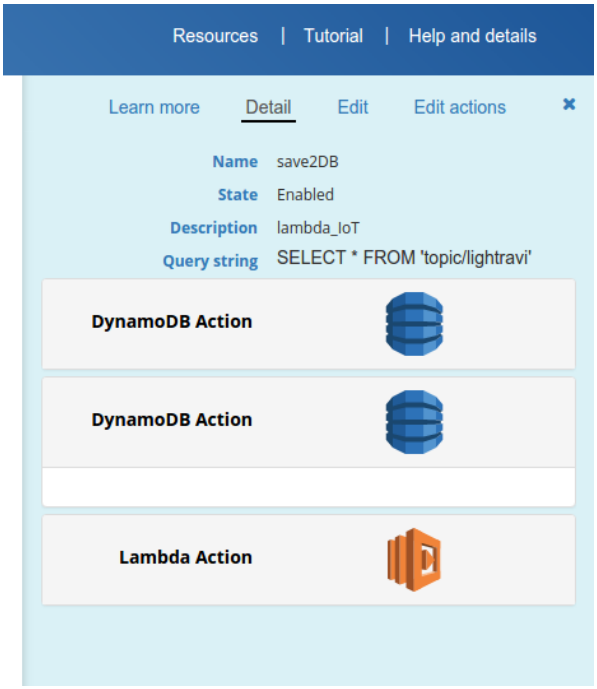
6.1 AWS IoT MQTT broker vs Local MQTT broker

We use the previously developed application in the earlier sections to send the publish and subscribe requests to the local MQTT broker and the same application to publish and subscribe the requests to local MQTT broker. Now we send the messages to AWS IoT using the application developed, we assume that the sender is temperature sensor and sending temperature data to the AWS IoT suite in the cloud, MQTT broker in the cloud receives published data from temperature application we developed.

The screen shots in figure 6 shows the data publishing to AWS IoT sending temperature data. The picture in figure 7 shows the subscriber which is receiving the data published to the topic figure *lightravi*, the picture in 8 shows the received data in the AWS IoT forwarded to the dynamoDB of AWS IoT.

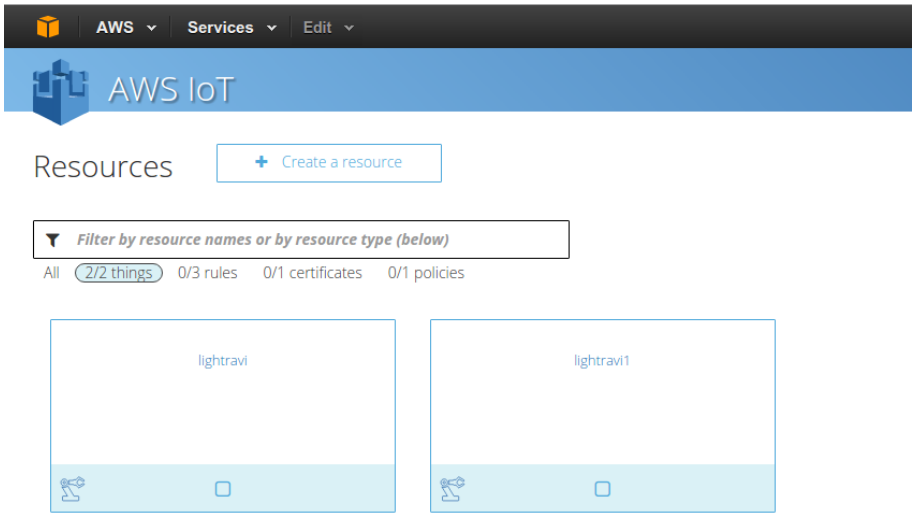
We check the performance of local MQTT broker using four methods as we mentioned earlier in this document. Single Path Single Client, Single Path Multiple Client, Multiple Path Single Client and Multiple Path Multiple Client.

Figure 3: DynamoDB and Lambda linked to the AWS IoT



The picture shows the dynamoDb and lambda services created and linked to the AWS IoT

Figure 4: Internet things registered in the AWS IoT



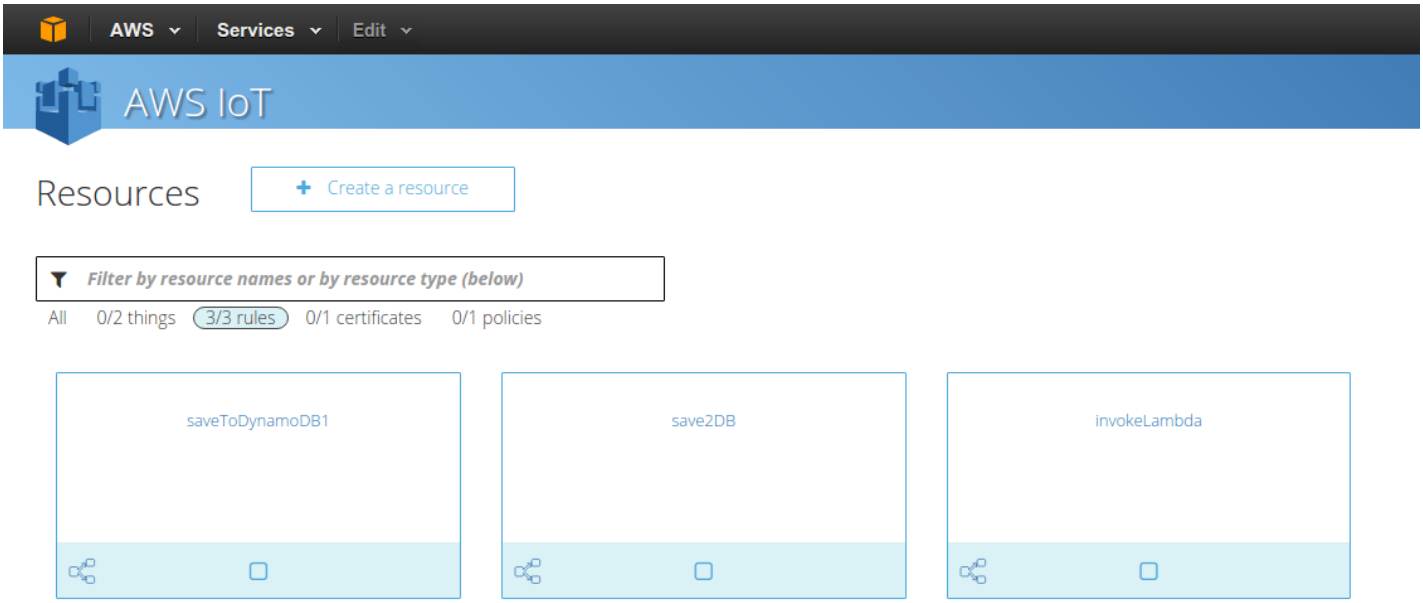
The picture shows the two things registered in the AWS IoT cloud, the IoT application we develop will connect to the server to publish and subscribe to the topics

6.1.1 Single Client Single Path

In Single Path Single Client method we use one instance of the application and connect to the local MQTT broker and AWS IoT broker. This provides less load but when the number of clients increased then the scalability problem comes for local MQTT broker but for AWS IoT suite there wont be any limitation.

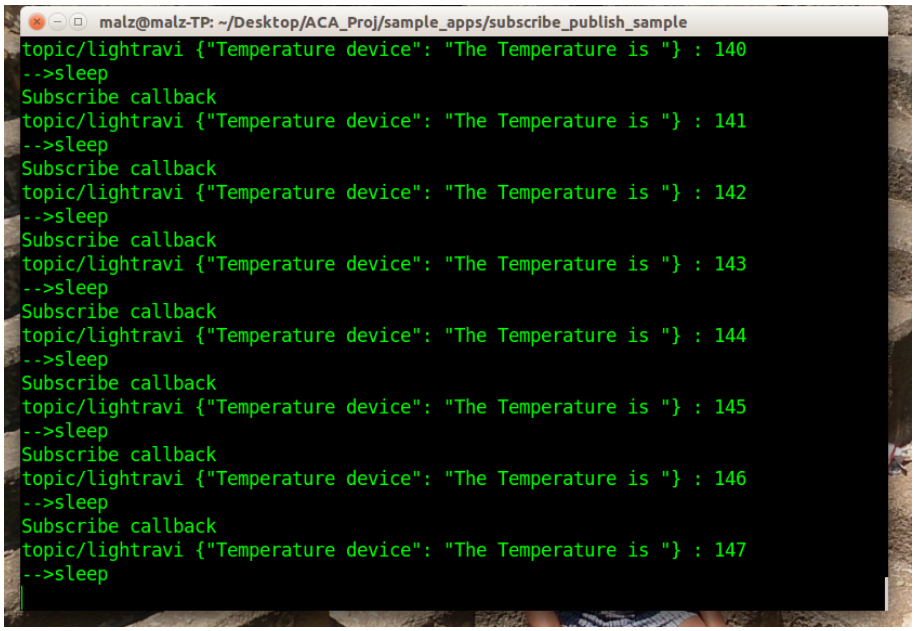
```
---------------
Aggregate Stats (Total # clients: 1):
-----------------------------------------------
Total Messages transmitted = 13456
Computed publish rate (msg/sec) = 79.0
```

Figure 5: Dynamo DB showing the data received from AWS IoT suite



The picture shows the socket error when number of client connections increased for local MQTT broker, where as in case of AWs IoT we did not face any problem

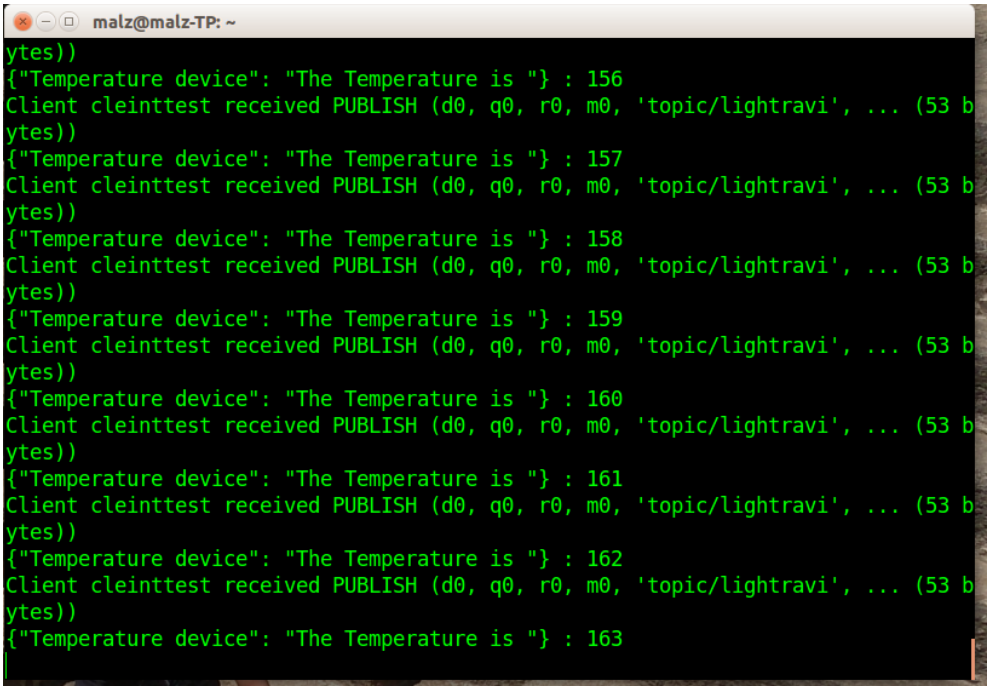
Figure 6: Temperature data updating to the AWS IoT



The picture shows the IoT temperature application publishing the temperature to the AWS IoT

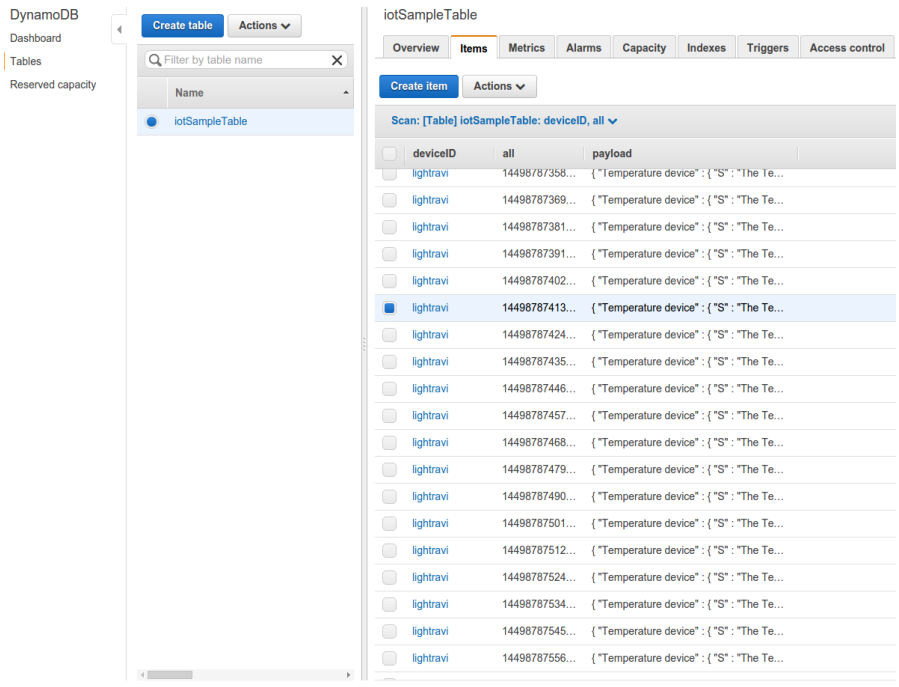
```
------------------------------------------------        (msg/sec across all subscribers) = 126
Total Messages received across all subscribers = 21784
Messages received with discard indication = 0           CPU usage = 0%
Computed subscriber rate
```

Figure 7: Subscriber to the AWS IoT thing with topic



The picture above shows the subscriber which is in another system receiving the messages published to a topic lightravi

Figure 8: Dynamo DB showing the data received from AWS IoT suite



The picture above shows the recorded values of the temperature sensor from AWS IoT server holding MQTT broker

```
Latency Info:
Individual latency bucket size = 1.0 us
Latency warmup                 = 30.0s
Latency Messages received      = 18694
Latency msg rate (msg/sec)     = 131
```

```
Latency Stats:
Minimum latency for subs       = 5688
Average latency for subs       = 84698221
50th percentile latency        = 0
95th percentile latency        = N/A
99th percentile latency        = N/A
99.9th percentile latency      = N/A
Maximum latency for subs       = 169485453
Standard Deviation             = 84697197

==> Garbage collection information
====> Collection count 96
====> Collection time 265 ms.
```

6.1.2 Single Path Multiple Client

In this method we create multiple clients in single ip address and test the performance of local MQTT broker and AWS IoT MQTT broker. This provides the load which is very high and local MQTT broker can not handle that much of load as the memory and processing speed are limited where as in the AWS IoT MQTT cloud there are not limits of memory and the scalability provides best usage of the resources for MQTT broker. Hence in this case AWS IoT MQTT broker is best and hence we are using the cloud.

```
---------------
Aggregate Stats (Total # clients: 200):
------------------------------------------------
Total Messages transmitted = 9701
Computed publish rate (msg/sec) = 54.0
------------------------------------------------
Total Messages received across all subscribers = 258851
Messages received with discard indication = 0
Computed subscriber rate
(msg/sec across all subscribers) = 1287

CPU usage = 0%

Latency Info:
Individual latency bucket size = 1.0 us
Latency warmup                 = 30.0s
Latency Messages received      = 222775
Latency msg rate (msg/sec)     = 1301

Latency Stats:
Minimum latency for subs       = 7046631
Average latency for subs       = 106253007
50th percentile latency        = 0
95th percentile latency        = N/A
99th percentile latency        = N/A
99.9th percentile latency      = N/A
Maximum latency for subs       = 195987088
Standard Deviation             = 106251983

==> Garbage collection information
====> Collection count 994
====> Collection time 29696 ms.
```

6.1.3 Multiple Path Single Client

In this case we consider the situation when the clients are distributed across globe which means the access time by the IoT client for local MQTT client is very slow and some time requires frequent maintenance, whereas in the case of AWS IoT cloud suite there are no limits of connectivity as the MQTT broker is shared across the globe and any moment of the IoT device can be tackled in the AWS IoT cloud by just changing the location of the MQTT in the AWS.

Table 3: Results

Method of execution	AWS IoT MQTT broker	Local MQTT broker
SP/SC	No Errors	No Errors
SP/MC	No Error	Socket Errors
MP/SC	No Error	No Errors
MP/MC	No Error	Socket/Data drop errors

6.1.4 Multiple Path Multiple Client

In this case multiple clients are installed in multiple locations which is much more scenario and the MQTT broker will receive heavy load. In this case the clients can range from 100 to millions and the local MQTT broker can handle maximum of 1017 clients at a time with very less message payload. Hence we choose the AWS IoT MQTT broker and we could not check its performance as we have very less number of clients and limited number of systems the maximum number of simultaneous connections we could test are 4 and our hardware resources limited us to instantiate more clients.

With this experiment it is shown that there can be unlimited number of client connections to the AWS IoT hub without any limitation, while the local systems has always a limitation in scaling as we need to buy more processors and keep them in the series. If we are using the AWS IoT, it provides performance, security and scalability without any additional. When we increased the number of client connections to more than 100, local MQTT broker throws socker error as shown in the picture.

The following snipped shows the client performance in local MQTT broker.

While it is very difficult to organize the data received in the MQTT broker and we need to implement a new API which copies the data from MQTT broker to the storage device, AWS IoT provides well structured database dynamo DB which receives the data from the

6.2 Results and conclusion

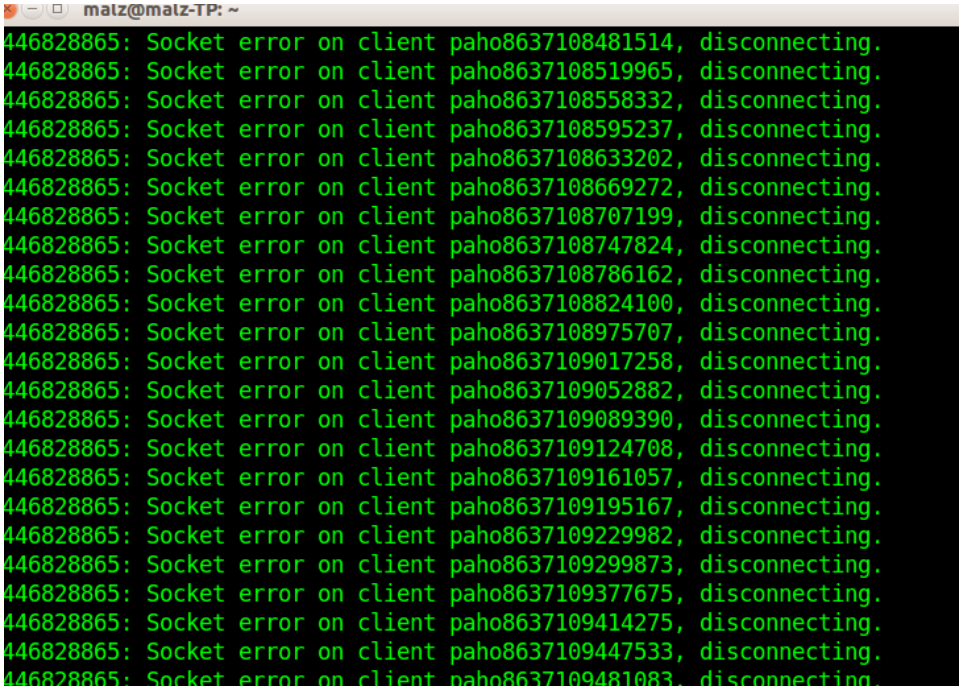
From the above table 3 it is very clear in all the scenarios the AWS IoT cloud performs better and also provides best scalability, changeability and memory management. AWS IoT also provides efficient memory management and process organization. As in the case of IoT there can be many devices which require continuous connection with the server as events need to be monitored and notified at the right time. So we conclude that the advantages of cloud services makes AWS IoT MQTT broker better choice when compared to the local MQTT broker. AWS IoT dynamo DB and Lambda services also makes it

7. REFERENCES

[1] Amazon. https://aws.amazon.com/documentation/iot/, 2015.

[2] A. Aris, S. F. Oktug, and S. B. O. Yalcin. Internet-of-things security: Denial of service attacks.

Figure 9: Dynamo DB showing the data received from AWS IoT suite



The picture shows the socket error when number of client connections increased for local MQTT broker, where as in case of AWs IoT we did not face any problem

In *Signal Processing and Communications Applications Conference (SIU), 2015 23th*, pages 903–906. IEEE, 2015.

[3] eclipse. http://www.eclipse.org/paho/, 2015.

[4] S. Misra, P. V. Krishna, H. Agarwal, A. Saxena, and M. S. Obaidat. A learning automata based solution for preventing distributed denial of service in internet of things. In *Internet of Things (iThings/CPSCom), 2011 International Conference on and 4th International Conference on Cyber, Physical and Social Computing*, pages 114–122. IEEE, 2011.

[5] Mosquitto. http://mosquitto.org/, 2015.

[6] D. Moustis and P. Kotzanikolaou. Evaluating security controls against http-based ddos attacks. In *Information, Intelligence, Systems and Applications (IISA), 2013 Fourth International Conference on*, pages 1–6. IEEE, 2013.

[7] MQTT. http://mqtt.org/documentation, 2015.

[8] P. Pongle and G. Chavan. A survey: Attacks on rpl and 6lowpan in iot. In *Pervasive Computing (ICPC), 2015 International Conference on*, pages 1–6. IEEE, 2015.

[9] S. Ranjan, R. Swaminathan, M. Uysal, A. Nucci, and E. Knightly. Ddos-shield: Ddos-resilient scheduling to counter application layer attacks. *IEEE/ACM Transactions on Networking (TON)*, 17(1):26–39, 2009.

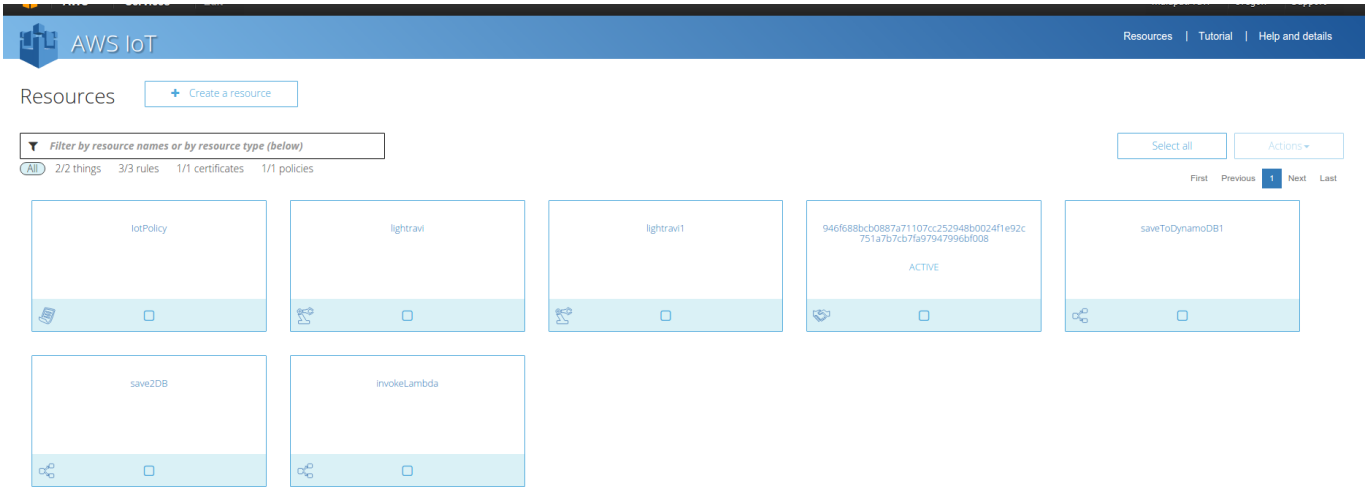
[10] remakeelectric. https://github.com/remakeelectric/mqtt-malaria, 2015.

[11] solacesystems. http://dev.solacesystems.com/downloads, 2015.

[12] X. Yang, T. Ma, and Y. Shi. Typical dos/ddos threats under ipv6. In *Computing in the Global Information Technology, 2007. ICCGI 2007. International Multi-Conference on*, pages 55–55. IEEE, 2007.

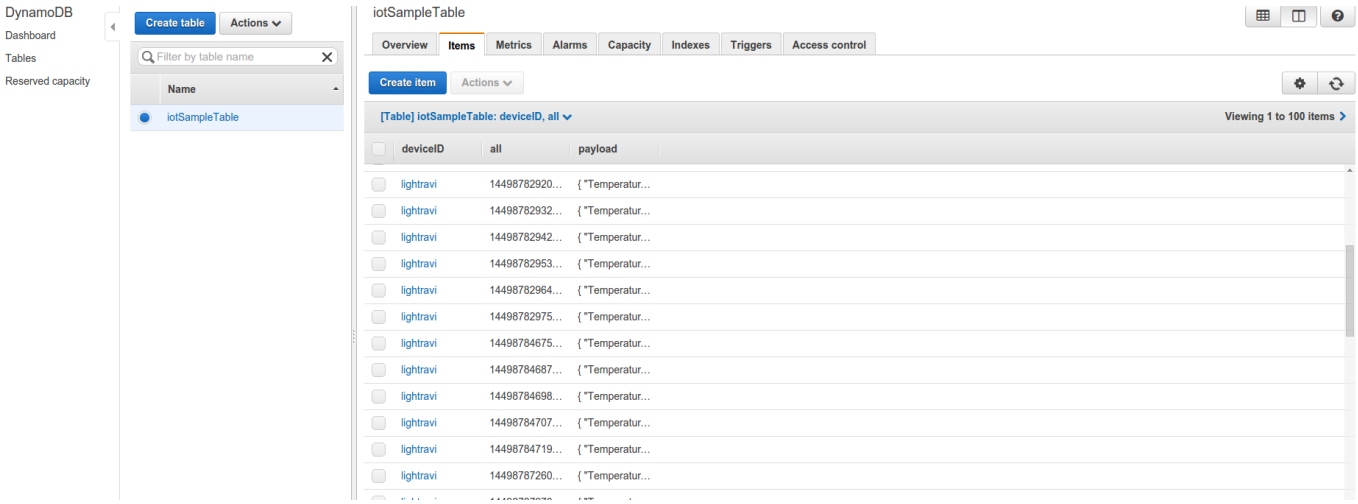
APPENDIX

Figure 10: AWS IoT suite setup in AWS



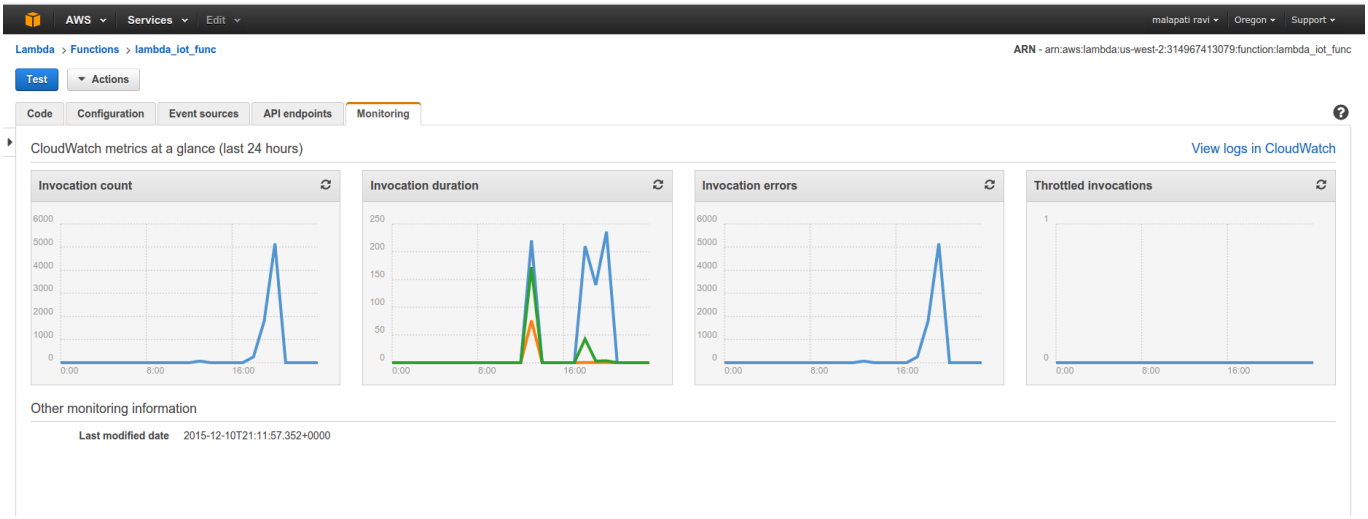
The picture shows the creation of AWS ioT things in the AWS IoT server with the MQTT broker and also shows the dynamo DB set up and lambda services setup

Figure 11: Dynamo DB showing the data received from AWS IoT suite



The picture shows the dynamoDb table storing the data received from the AWS IoT

Figure 12: Lambda showing the statistics of invocation from AWS IoT



The picture shows that the creation of Lambda services for AWS IoT events