# Final Report

Abstract

Many IoT devices have the tradeoff between scalability and low latency. However, with the maturity of AWS cloud platform, it offers us some possible solutions to deploy our own IoT devices and provide reliability and reasonable latency. Our IoT system is able to collect data from the sensors and show the data to the users via web application in a real time fashion. Users can also send command back to the sensors to remotely control lights. The primary focus of this paper is to illustrate and justify how we design and implement an IoT ecosystem that has considerable low latency and can withstand relatively high throughput.

Introduction

There is widespread that IoT is going to lead the next generation of industry evolution. With the cloud services gaining popularity and becoming more viable to the developers, leveraging the messaging service and data processing provided by AWS enables us to develop and implement our own IoT system much easier. Before we dive into the explanation of our IoT implementation, we need to give a brief introduction of the various services provided by AWS that our design relies on.

* 1. AWS EC2

Amazon Elastic Compute Cloud is a web service that provides resizable compute capacity. It provides some power function that makes developers easier to deploy and develop their own applications. Moreover, EC2 instance has the advantage of being protected by a firewall. To allow outside applications to access certain ports, users have to manually configure the firewall and make sure the security groups have full control of the traffic on certain ports. Another advantage of EC2 is that it allows users to specify how many computing power they need; the more computation intensive the application is, the more cores the users can require.

* 1. AWS SNS

AWS SNS stands for Simple Notification Service. It adopts the traditional publish/subscribe messaging paradigm. The SNS acts an information aggregator (broker) as the publishers send messages to it and SNS redistributes the messages to subscribers based on their interested topics. Moreover, SNS has an advantage of being able to support multiple languages (programming languages independent). In our implementation, the publisher application is written in C++ while the subscriber application is in Javascript. Furthermore, SNS also provides text messaging and email services, which means applications can send text messages and emails to the users via SNS. As for latency performance, we will have an in-depth discussion in the later section.

* 1. AWS IoT

AWS IoT is a cloud platform that enables devices to be connected easily and securely. Moreover, it enables developers to leverage on the various other services it provides to build a scalable and configurable ecosystem that supports the transmission of millions of messages concurrently. Using AWS IoT, applications can keep track of and communicate with other devices and retrieve real time data.

IoT System Overview

The IoT system we design is able to collect data from the sensor and users can access these data in a real-time fashion via web application. Moreover, users can remotely control lights and other devices through the application and send email alert to their email addresses. All these features have relatively low latency, which satisfies the real-time requirement.

Design

The IoT system we designed have several components, which include sensors, raspberry pi, intermediate server, database, and user application (frontend and backend).

2.1 Sensors and Raspberry Pi

In our current implementation, we use temperature, light, and moisture sensors. All these sensors are connected the Raspberry Pi 3 and some data retrieving applications are constantly listening data sent from the sensors. When the applications receive sensor data, they publish the data to the AWS MQTTs under the topic “sensorData”. Another application, which is also deployed on AWS, subscribes messages from the AWS MQTTs. It is responsible for receiving commands sent from the user application and retransmitting the command to the sensors, in our case a led light.

2.2 Intermediate Server and Database

Both the intermediate server and the database are deployed on the AWS EC2 instance. The intermediate server receives messages, which are sent by the Raspberry Pi, through the EC2 SNS service. After it receives the messages, it processes the message and store them in the database. Moreover, the intermediate server can also receive HTTP request sent by the user application and publish the information (commands) to the Raspberry Pi. The database we used is MongoDB, which is a flexible and lightweight database.

2.3 Application Backend and Frontend

The framework we used to implement the application is Meteor. It has the advantage of easy deployment and can provide some features of the native mobile apps (IOS & Android). Yet, due to the tight schedule of the development, we didn’t migrate the application to the mobile end. Another key advantage of Meteor is that it can update the frontend dynamically. For instance, when the intermediate server receives a new message and store the data into the database, the frontend application will update itself automatically without the user refreshing the page. One thing needs to notice is that while this feature is tempting and greatly enhances user experience, it will consume large amount of CPU resources. In our implementation, as we deploy intermediate server, database and application backend on one single core EC2 instance, some moderate overhead will be added to enable this automatic update.

Moreover, when users click on the “send alert” button on the webpage, the backend server will send a command to the intermediate server, and the intermediate server will send the message to the Raspberry Pi through the AWS MQTTs.

# Latency Performance

Latency of Raspberry Pi to Frontend Application

The overall traffic from Raspberry Pi to frontend application has three separate parts: Raspberry Pi sending messages to intermediate server, backend server polling data from the database, backend server sending data to the frontend application.

Figure 2 shows the latency decomposition of this one-way traffic. The rightmost column is the total latency of the traffic (Raspberry Pi – Frontend App). Each message is 1KB and the sending rate is 1 msg/s. Figure 3 and Figure 4 show the change of latency when we increase the sending rate to 10 msg/s and 100 msg/s. As we can see from the graphs, both the total latency and the decomposed latency remain stable. And a message sending rate of 100KB/s is actually relatively high. So, the experiment result shows that the system we design can provide relatively low latency (approximately 1 second) and support large volume of traffic.

One thing needs to notice is that the biggest contributor to the overall latency is actually the time it takes to transmit messages from the backend server to the frontend application. The reason for this behavior is because to enable the real-time automatic update of the frontend application, the backend server takes up a large chunk of the CPU resources. As the intermediate server, database, and the backend server are all deployed on a one core EC2 instance, the CPU contention causes the additional overhead (latency penalty).

Latency of Frontend Application to Raspberry Pi

The overall traffic from frontend application to raspberry pi has two parts: frontend application sending messages to the backend server, and backend server sending messages to the Raspberry Pi via AWS MQTTs. The reason why there are less components in this part than the previous one is because we want to reduce the latency as much as possible so that if the command is really urgent, it will be received by the sensor on time.

Figure 5 shows the latency decomposition of this one-way traffic. We can see that the total latency of this traffic is really low, approximately 0.2s, which matches the real-time requirement.

Latency of AWS Email Service vs NodeJS Email API

When the users send an alert message to the sensor, the backend server also sends out emails to the users’ email addresses. There are currently two feasible ways to implement that feature. The one we adopt is using AWS SNS Email Service. While it has a higher latency than the native NodeJS Email API, AWS SNS Email Service is actually linked with the MQTTs service; when the message sent by the server contains certain key word, the AWS SNS Email Service will be triggered and will send out an email alert to the designated email addresses. Figure 5 shows the latency performance of both AWS Email Service and NodeJS Email API. We timestamp the email service by subtracting the time the server sent the email with the time the user receives the email. From the graph, we can see that AWS Email service has a latency around 5s while NodeJS Email API has a latency of 2.5s. However, due to the lack of scalability and flexibility of NodeJS Email API, we still choose AWS SNS Email Service, which provides a fair amount of features with moderate latency penalty.

Conclusion

Our work represents a fair and reasonable IoT design by leveraging on the services provided by AWS. The frontend user application can show and plot the sensor data collected by the Raspberry PI in a real time fashion. Moreover, to ensure the user command reaches the Raspberry PI as soon as possible, we solely use the AWS MQTTs service and get rid of the AWS SNS service, which adds extra overhead. Empirical results showed that the overall latency for the sensor data to be updated on the frontend application is around 1s, and most of the latency is contributed by the CPU contention as intermediate server, database and backend server are all deployed on a single core EC2 instance. However, the latency for the commands to be received by the Raspberry PI is really low, around 0.2s.



