Project Report
Summer Internship 2020

Smart City Control and Management

Interns

Chetan R Athni	PES1201800468
Sakshi Vattikuti	PES2201800157
Hemant Sathish	PES2201800045
A Anantha Krishna	PES1201800506

Mentor(s)

Vishwas R	PES1201700704
Pranav Bhatt	PES1201800764

PES Innovation Lab
PES University
100 Feet Ring Road,
BSK III Stage,
Bangalore-560085

Abstract

In the present-day world, the tag of "Smart City" proves to be a statement of eminence. But due to the vast amount of sensors around the city, it becomes difficult to manage the entire system. We propose a solution to this issue, which offers Sensor as a service. Different sensors across the city (edge nodes) send data to an intermediate layer. The data is filtered as per the users' request in this layer and sent forward to be displayed on a dashboard. As this approach doesn't use any proprietary software, it avoids any hindrance in the development. This can be considered as the major advantage and difference in comparison with the current models.

Contents

Chapter 1

Introduction

In today's IT age, data is the main commodity, and possessing more data typically generates more value in data-driven businesses. EMC estimates that there will be around 30 billion connected devices by 2020 [1]. These connected devices constitute the Internet of Things (IoT) and generate a massive amount of data. The challenge with respect to the current system is that everyone needs to deploy their own sensors and installation, monitoring and maintenance of those sensors can be difficult and expensive.
In current implementations of cloud-based applications, most data that needs storage, analysis, and decision making is sent to the data centres in the cloud. As the data velocity and volume increases, moving the big data from IoT devices to the cloud might not be efficient, or might not be feasible in some cases due to bandwidth constraints. On the other hand, as time-sensitive and location-aware applications emerge, the distant cloud would not be able to satisfy the ultra-low latency requirements of many real-time applications, provide location-aware services, or scale to the magnitude of the data that these applications produce. Moreover, in some applications, sending the data to the cloud may not be a feasible solution due to privacy concerns and also the cost of storage/ processing of big data.[2]
To address the issues of high-bandwidth, geographically dispersed, ultra-low latency, and privacy-sensitive applications, there is a quintessential need for a computing paradigm that takes place closer to connected devices. A new platform is needed to meet these requirements; a platform called Fog Computing or, briefly, Fog, simply because the fog is a cloud close to the ground. Function-as-a-Service (FaaS) is a promising programming model for fog computing to enhance flexibility.[3]

The proposed model provides a platform to collect and process digital data from the sensors without having to compromise on factors such as latency and cost, using Fog computing. It employs FaaS to allow users to define their functions to be executed using the collected data. It uses an intermediate layer which is a very meagre line away from becoming a fog layer, and processes the data requested by the user and sends only the required data to the Backend. This filtering layer reduces the volume of data to be sent to the Backend, thereby reducing the latency and providing real-time data analysis and visualization.

1.1 Problem Statement

There is a need for an easy to use system that allows for management, logging and control of edge nodes on a massive scale. Existing solutions make use of proprietary designs and software, hindering development.
We aim to provide smooth control and management of sensors and devices spread out in a large vicinity to make all types of information readily available to anyone who requires it, with the help of a Dashboard. The model is capable of implementing Function as a Service (Faas), Sensor as a Service and Dataset as a Service .
This report further talks about how this problem statement is assessed, how the architecture is built which can be seen in the literature survey and how it has been implemented for a small scale in the main body.

1.1.1 Use Case

Consider an air purifier company as the client for the system. The company will want the data on different factors like ppm levels, temperature, light intensity, humidity etc. at various locations. With these factors, the company can calculate its factor of comfort according to which they can manufacture necessary products. To get this factor, the company can insert a function with the mathematical formula for calculating the comfort factor (which might be confidential). So, the sensors (edge nodes) send in all the data to the intermediate layer and in this layer, the required data is extracted and the values are substituted in the formula.
Also, the system stores all the data from the edge nodes in the intermediate

layer and sends the filtered data to the Backend layer. The data incoming in the Backend layer is used by the dashboard to plot graphs dynamically. The data incoming in the Backend layer is used by the dashboard to plot graphs dynamically. Whenever required, the clients can download the dataset for their usage, which portrays the Dataset as a Service function. With all these services, it can implement Sensor as a Service.

Chapter 2

Literature Survey/Related Work

The current models across the world follow a cloud-based system into which data is ingested from the edge nodes but the task of analyzing data and responding real-time in dashboards is seen to be a difficulty due to the distance between the edge nodes and the cloud server. To beat this drawback Singapore has come to terms with using Fog computing to decrease latency as cited in [4].
The world generates more data every two days than from the dawn of early civilization through the year 2003 combined and data rates are still growing at approximately 40% per year. It was concluded that in 2016, cloud computing dominated many components within the Information and Communication Technologies (ICT) market, pushing cloud revenue growth above 25% year-on-year. A cloud-enabled business model survey found that 62% of Chief Information Officers and Chief Data Officers consider cloud computing as the top priority for ICT [4].
The different types of available cloud infrastructure support different types of connectivity across the city such as connectivity from IoT to city services, government agencies etc. and connectivity between applications, platforms and storage. As all of these connectivities are managed by cloud computing infrastructure they provide Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS). This also leads to both vertical and horizontal scalability. With respect to cloud infrastructure, backup servers are allowed which helps in the scenario of disasters. Recovery can also be done via scheduling. Cloud infrastructure also provides a high level

of security for all the stored data. This proves the demand and role of cloud computing to help in the development of smart cities, as cited in [5].
We take in a few inputs from the above citations to propose a system of our own which has many more functionalities and handles other drawbacks which these systems face.

2.1 Apache Openwhisk

Apache OpenWhisk is a serverless platform that performs functions in response to events. The platform uses a function as a service (FaaS) model to manage infrastructure and servers for cloud-based applications and servers using Docker containers. It uses the FaaS model to serve tasks known as functions. These functions are made up of triggers and actions.
Openwhisk here, is used to send sensor data and metrics to the other layers by invoking a wsk action. The invoked action sends data through a Kafka message queue to the next layer.[6]

2.2 Apache Kafka

Apache Kafka is typically used for building real-time data pipelines and streaming applications. It powers many production workloads that need reliable, high-velocity data ingestion. Kafka provides a durable message store, similar to a log, run in a server cluster, that stores streams of records in categories called topics. It can support a large number of consumers and retain large amounts of data with very little overhead. The documentation does a good job of discussing popular use cases like Website Activity Tracking, Metrics, Log Aggregation, Stream Processing, Event Sourcing and Commit logs. It employs a dumb broker - smart consumer model, where the broker does not try to track which messages are read by consumers and only keeps unread messages. Kafka keeps all messages for a set period and has a very high throughput [7].
The proposed model uses Kafka as a data streaming unit to send Metrics and sensor logs to different layers for further processing.

2.3 Docker

Docker is a tool designed to make it easier to create, deploy, and run applications by using containers. Containers allow a developer to package up an application with all of the parts it needs, such as libraries and other dependencies, and deploy it as one package.
In this project, we use docker to run Kafka broker, zookeeper, Kafka producers and consumers, OpenWhisk and dashboard.

2.4 Dash

Dash is a Python framework used for building analytical web applications. It is a powerful library that simplifies the development of data-driven applications. Built on top of Plotly.js, React, and Flask, Dash ties modern UI elements like dropdowns, sliders and graphs directly to any analytical python code.
The project uses dash, to create the dashboard where the user can interact with the data. The user can choose an area, type of graph and many more features offered by the dashboard, to view the collected data however needed.
Most of the already existing models work only on a single type of sensor. This project aims to be flexible enough, to be able to work with multiple sensors of different types, to provide necessary data as and when the client requests for data about any available sensor and also allow users to define their functions to be implemented with the sensor data collected.

2.5 Docker API Client

The Docker SDK for is a Python library for the Docker Engine API which lets you do anything the docker command does, but from within Python apps – run containers, manage containers, manage Swarms, etc [8]. It is an API used to execute a custom function in the form of a docker container, from within a running container.

Chapter 3

System Architecture and Design

3.1 Approach

First, each part of the problem was analysed and the quantitative, as well as qualitative data, were taken into account to decide on how to approach this with an efficient solution. It had been decided to go about an approach that highlighted the ease of use of our system. The Edge nodes need to be powerful enough to continuously transmit data to the intermediate layer and perform custom functions if needed by the user, this could be done by including a FaaS framework.
Apache Openwhisk had been decided to perform serverless execution of functions and had concluded that the same could be used to build the FaaS Platform.
The Intermediate Layer had to be efficient as it had to filter all the data coming from the Edge nodes while simultaneously pushing all the logs into a database. It also had to listen to custom functions and compute them parallelly with the original task.
The Backend had to be meticulous to cover all the bases that the UI laid out for us. To achieve this Dash, a Python framework for building web applications had been used.

Finally, basic knowledge of the dependencies required for the project was acquired.

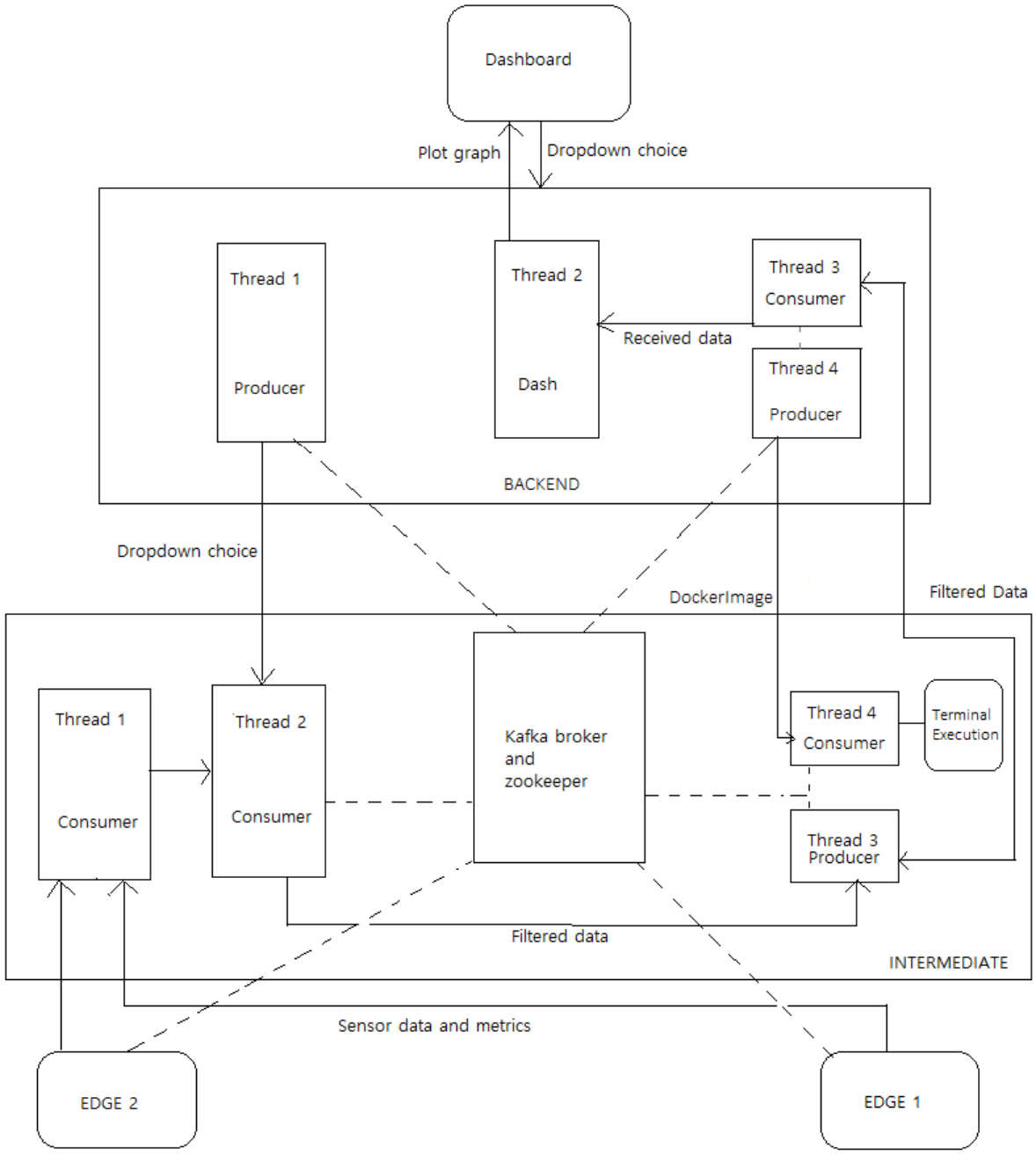

Figure 3.1: The Architecture

As it can be seen in diagram 3.1 the Edge layer consists of two edge nodes, namely EDGE1 and EDGE2. On each of these nodes, Openwhisk is used to create an action. This action consists of a python script that generates the system metrics and raw sensor values. When an action is invoked, sensor data is generated every 2 seconds (Time delay can be changed). A Kafka producer is set up in each of the nodes to transmit the data to the intermediate layer to a topic called 'sample'.
For this prototype, only PPM values are generated at these nodes.

The Intermediate Layer is responsible for all the processing. It receives data from the edge nodes at all times. The Kafka-broker and zookeeper are located in this layer, to which all the other producers and consumers are connected. These connections are represented as dotted lines in the figure.
It works on 3 threads:

1. Thread 1: Consumer thread
   This thread receives all the data sent by the edge nodes.

2. Thread 2: Consumer thread
   This thread receives the user's choice from the backend and sends the choice to the producer thread for filtering.

3. Thread 3: Producer thread
   This thread filters out the necessary data and sends the required data to the Backend to a topic called 'filtered'.

4. Thread 4 : Consumer thread
   This thread receives the DockerImage name from the backend at the topic 'OptionName'. An API called Docker Client API is used to pull this image from DockerHub and is then run, from within the running Intermediate container.
   If the function has any values to be printed, they are printed on the Intermediate terminal.
   Essentially, the returned values are to be redirected back to the dashboard, to be displayed to the user.

The Backend Layer acts as an interface between the system and the user. It has the dashboard web page running in a docker container. The dash app works on multi-threading, with 3 threads:

1. Thread 1: Producer thread
   Once the user chooses an option from the drop-down list, this value is sent to the intermediate layer through a Kafka-producer. It sends the requested option to a topic named 'LocationReq'.

2. Thread 2: Consumer thread
   This thread consumes all the filtered data sent from the intermediate layer. It uses this data and plots graphs and updates the gauge meters and any other necessary data.

3. Thread 3: Dash thread
   This is the main thread which updates graphs dynamically. It uses callback functions to plot graphs at an interval of 5 seconds and updates the graph on a real-time basis. By default, the drop-down value is set to 'HSR'. Every time the user chooses any other option, the graph is refreshed and the plotting starts from zero again.
   The user can also choose from a list of types of graphs such as "Bubble Chart", "Scatter plots" etc.. and view the same data in different graphs.
   There's also a section for aggregate values such as Minimum, Maximum and Average of all the data received from the edge node.
   Apart from the user, the admin has a separate section where they can view the system usage such as the CPU, RAM and Disk utilization of the edge nodes the data is received from.

4. Thread 4: Custom function
   This functionality of the dashboard allows the user to upload any function of their choice, which would be executed with the sensor data collected.
   The function to be uploaded has to be converted into a Docker Image, which would then be pulled from the docker registry to be executed in the intermediate layer.

3.2 Hardware and Software Requirements

Software and computer

Software Name	Version
Apache Kafka	0.9.0
Apache OpenWhisk	2.5.0
Dash	1.12.1
Docker	19.03
Kafka-python	2.4.0
OpenWhisk CLI	1.0.0
OpenWhisk Run-time Python	1.14.0
Docker API Client	1.0.0

Table 3.1: Software used

VM Instance	OS	RAM	Storage
Edge 1	Ubuntu 20.04	7.5 GB	100 GB
Edge 2	Ubuntu 20.04	5 GB	100 GB
Intermediate	Ubuntu 20.04	7.5 GB	100 GB
Backend	Ubuntu 20.04	3.75 GB	100 GB

Table 3.2: Computer statistics

Chapter 4

Results and Discussions

Based on the preliminary milestone of 8 weeks, our prototype has been developed successfully. While the edge nodes continually obtain data to be perceived as sensor values, they are automatically forwarded to the intermediate layer through Kafka. The intermediate layer receives the area request made by the client from the frontend, and thus forwards the requested data, thereby acting as a filter.

The dashboard plays a vital role in communicating between the user and the nodes. It visualizes the received data as per the user's choice. As a proof-of-concept, the CPU and RAM statistics data of the particular edge node, which are useful for the system admins to monitor the nodes to ensure there are no breakdowns or outages are also displayed.

The custom function option is made available to the user, for him/her to execute his/her functions on the collected sensor data. The current prototype executes and displays the function output on the intermediate terminal itself, and has not yet been redirected back to the dashboard. This can be visualized in flowchart 4.1

4.0.1 Challenges faced

The main roadblock faced during the project was the setup of the OpenWhisk platform. Since the installation guides cover the process at a broader level, many errors required a more detailed understanding of what we're causing them. The primary attempt towards installation was using the Ansible

Playbooks. Due to an error caused by CouchDB, other methods were to be looked into.
The James Thomas blog [9] was of great help for this setup which showed the incubator method of installation and seemed to be a better fit. Moving forward with the installation, another hurdle faced was with the configuration of the wsk command-line interface, which produced an API error. The solution was to go through the OpenWhisk directories to obtain the proper authentication keys and hostname. The final step towards configuring the wsk CLI was to bypass security authentication since wsk would expect an HTTPS connection but received an HTTP connection instead. Suffixing the wsk commands with '-i' solved this issue.
However, towards the end of 8 weeks, a new update in the Openwhisk installation seemed to have broken the setup process. There have been discussions on considering an alternative platform to continue with the project.

At the backend, a design challenge faced during the construction of the dashboard was the choropleth map. Due to the unavailability of required geoJSON files, it was not possible to include a choropleth map on the dashboard, which is believed to elevate and enhance the user experience to another level.

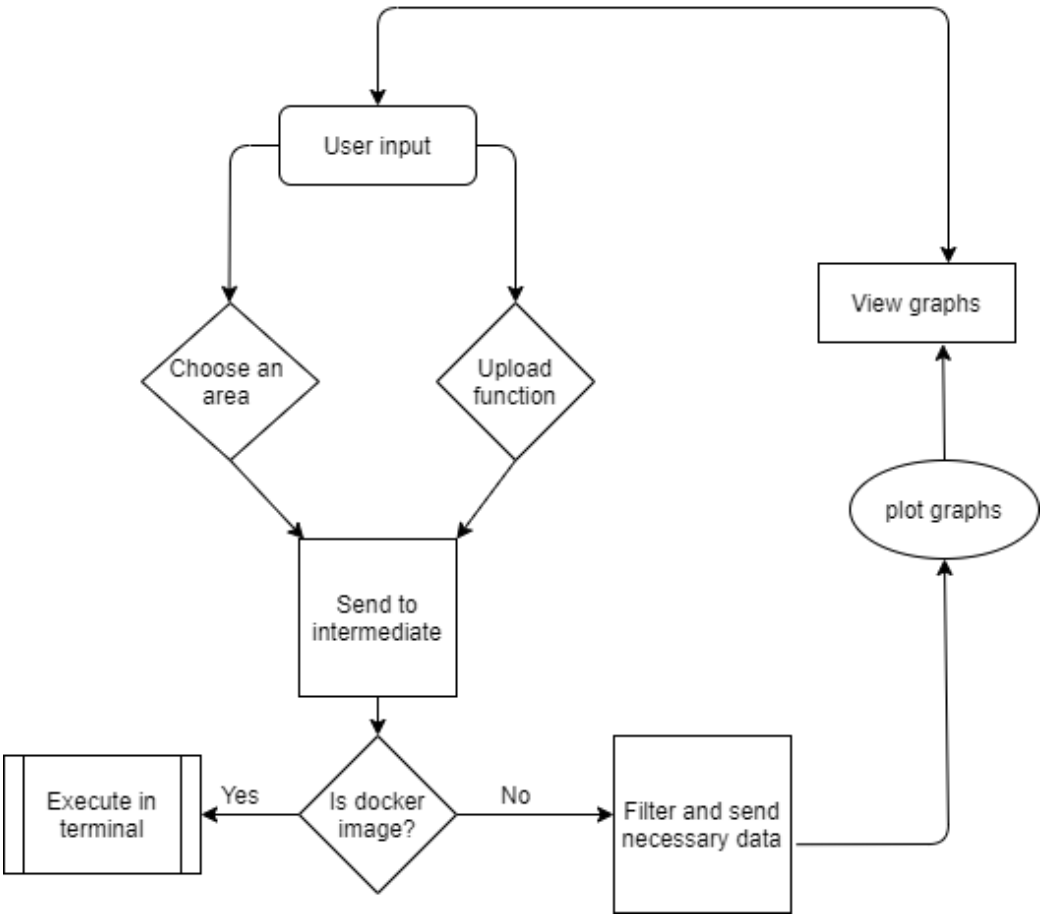

Figure 4.1: The Logic Diagram

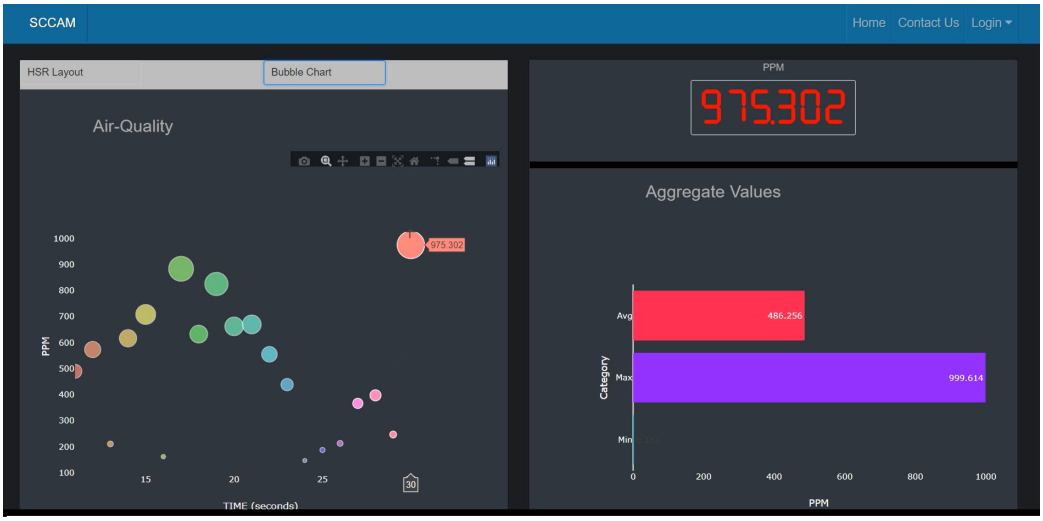

Figure 4.2: User view : Dashboard with "Bubble Chart" chosen

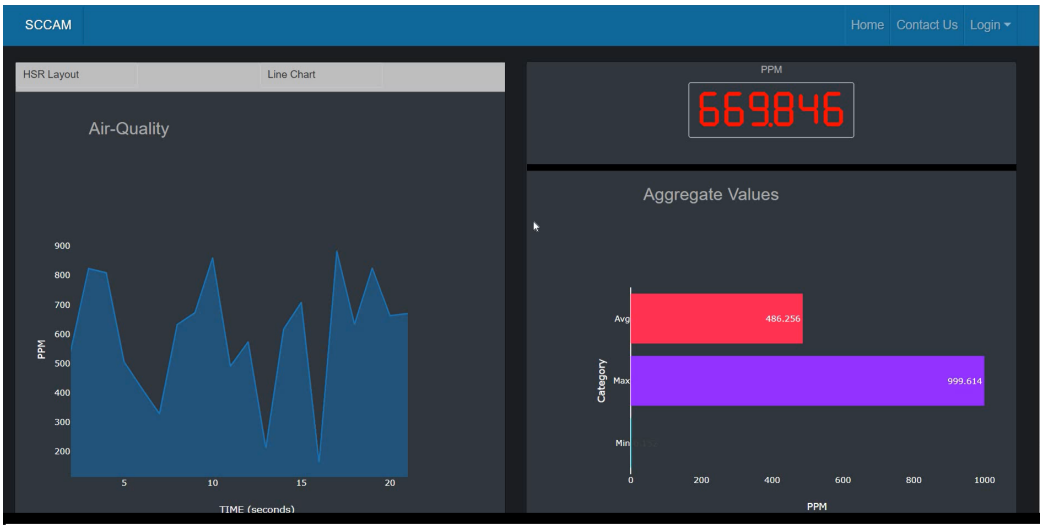

Figure 4.3: User view : Dashboard with "Line Chart" chosen

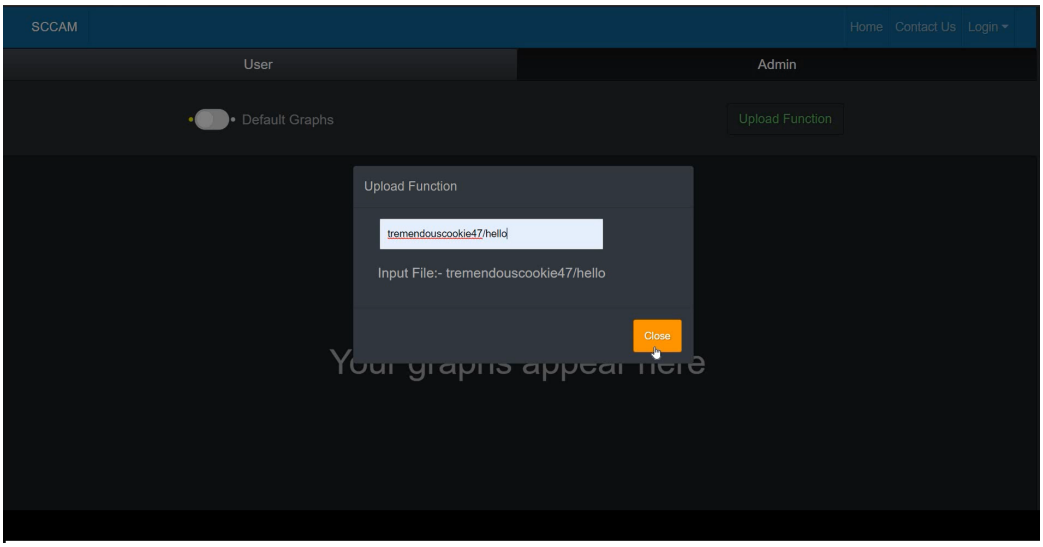

Figure 4.4: User view : Dashboard upload function

Figure 4.5: Admin view

Chapter 5

Conclusions and Future Work

Considering the rate at which all the cities are developing, it can be confidently said that down the line smart cities will be prevalent. But a system to handle the issues that come along with the tag "Smart City" will be necessary. Hence, it can be seen that the SCCAM system is a strong approach to overcome the issue of handling many sensors in a large vicinity which is what a smart city would primarily require.

Since the system replicates a Sensor as a Service, the clients do not need to bother about the infrastructure as they would only need to register themselves and use it entirely from the web. They will be charged solely on the basis of the function activations done.

In the near future, the system will also have supplemented functionalities for downloading the dataset and viewing the incoming data stream in forms of choropleth, scatter-map etc. It will also have a setup for user and admin login authorization with which the dashboard will be appreciably more organized. The whole system will then be tested on campus to verify the scalability of the product. Eventually, the system will be upgraded to be of use at a larger scale.

Bibliography

[1] E. D. U. with Research Analysis by IDC 4 2014.

[2] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, MCC '12, (New York, NY, USA), p. 13–16, Association for Computing Machinery, 2012.

[3] A. Yousefpour, C. Fung, T. Nguyen, K. P. Kadiyala, F. Jalali, A. Niakanlahiji, J. Kong, and J. P. Jue, "All one needs to know about fog computing and related edge computing paradigms: A complete survey," *ArXiv*, vol. abs/1808.05283, 2019.

[4] R. Ng, "Cloud computing in singapore: Key drivers and recommendations for a smart nation," *Politics and Governance*, vol. 6, p. 39, 11 2018.

[5] J. Wu, "Cloud computing powers smart cities security, scalability, and elasticity are the benefits of cloud computing," 10 2019.

[6] J. Thomas, "Python packages in openwhisk," 4 2017.

[7] A. Siddiqi, "Getting started with apache kafka in python," 6 2018.

[8] "Docker api."

[9] J. Thomas, "Starting openwhisk in sixty seconds," 1 2018.