Miniature Implementation of an IoT-based Smart City

Jonathan Camenzuli

Supervisor: Dr Inġ. Trevor Spiteri

June, 2023

Submitted in partial fulfilment of the requirements for the degree of B.Sc. (Hons.) in Computer Engineering.

L-Università ta' Malta
Faculty of Information & Communication Technology

Acknowledgements

The completion of this project would not have been possible without the support and assistance of various people.

First and foremost, I would like to thank my supervisor **Dr Inġ. Trevor Spiteri**. His expert advice, insightful feedback and continuous support has been significant in the shaping the direction of this project. I am grateful for his time, patience, support and guidance.

I would also like to provide my sincere thanks to **Dr Inġ. Mario Cordina** on behalf of **Epic Communications Limited** for providing technical support and access to Epic's NB-IoT infrastructure.

I am eternally grateful to **my family** for their unwavering support, love and understanding throughout my academic journey. Their constant belief in my abilities and aspirations through their sacrifices and encouragement have been the driving force behind my achievements. I am incredibly fortunate to have them by my side, and I attribute my success to their unending support.

I would also like to thank **my course mates** with whom I have shared these past three years. Despite the various challenges, in particular the COVID-19 pandemic, we have still managed to form great bonds of friendship and camaraderie. Their friendship and shared experiences have made this academic journey more meaningful. I would like to wish them all the best in their respective careers.

To anyone who has not been mentioned here, but has provided their valuable help and insights during this project, I am deeply grateful. Their contributions, though perhaps not prominent, have made a significant role in the making of this project. Whether it was sharing their technical expertise, offering suggestions, or providing a word of encouragement, their input has greatly influenced the outcome of this project. I appreciate their willingness to lend a hand and their presence in my academic journey.

Abstract

Ever since the idea of a Smart City was introduced, the Internet of Things (IoT) has been a key pillar of the technological aspect of Smart City development.

Cities should fully grasp the advantages and opportunities of the IoT for Smart Cities since there is so much promise and opportunity in a wide variety of fields, including traffic management, urban mobility, security and healthcare.

This project presents a scalable miniature implementation of an IoT-based smart city model, using three physical nodes that perform air quality monitoring, parking sensing, and fire detection.

The nodes communicate with a centralised server using NB-IoT and CoAP protocols. The data ingested from said nodes is stored and visualised using InfluxDB and Grafana.

The project evaluates the system from different perspectives, such as integration, load, and battery consumption. Integration testing indicated that all system components integrate properly with each other. Load testing demonstrated that the system can handle a reasonable number of requests without performance degradation. Battery consumption tests indicate that the physical nodes' battery life lasts for several days.

Contents

List of Figures

List of Tables

List of Abbreviations

1 Introduction

This project aims to develop an implementation for a Smart City model, with a focus on the data collection aspect. This will be achieved by making use of a small number of heterogeneous physical nodes, that would not be constrained to that number, in the case of implementation on a much larger scale.

The implementation involves three physical nodes which will be performing the following tasks individually:

- Air Quality Monitoring (AQM)
- Parking Sensor
- Fire Detection System

Any data collection performed from these nodes will only be made for testing purposes and will be performed in a laboratory environment. Following data collection, the data will be stored and visualised on a server setup to handle such data accordingly.

1.1 Motivation

The project's motivation is the world's growing urbanisation trend and the strain it places on urban resources such as infrastructure and energy. By utilising emerging technologies to raise citizen quality of life and increase service efficiency, smart city concepts seek to address these issues. Nevertheless, putting such concepts into practice necessitates overcoming a number of technological obstacles, including choosing the appropriate communication protocols, reducing power consumption, and assuring efficient data storage and presentation. Due to these difficulties, this is a non-trivial topic that takes extensive thought and preparation in order to be implemented successfully.

1.2 Aims and Objectives

Other than the main objective of setting up the network which will model a smart city, other objectives have been set for this project:

1. Minimise power consumption by the physical nodes
2. Identify the best possible Application Layer Protocol to be used in this project

3. Identify the best possible Low-Power Wide-Area Network technology to be used in this project
4. Devise a data collection pipeline which makes use of already existing software platforms in order to collect, store and visualise data appropriately

1.3 Report Structure

This document is structured as follows:

Chapter 2 discusses the research conducted to understand the basis behind this project which includes Smart Cities, Previous Works and any hardware and software used.

Chapter 3 introduces a justified top-level outlook of the system architecture, communication protocols, and software and hardware configurations.

Chapter 4 describes the hardware and software methodologies behind the Smart City implementation.

Chapter 5 goes over the methodology used to evaluate the system from different perspectives such as integration between each system component and battery consumption

Chapter 6 concludes the report by giving a brief overview of the system and what was achieved. Possible improvements to the system are also mentioned.

2 Background & Literature Overview

2.1 Smart Cities

Smart cities use digital technologies to reduce resource input, improve people's quality of life, and increase sustainable economic competitiveness within the local area. It involves the utilization of intelligent solutions for infrastructure, energy, housing, mobility, services, and security that are based on integrated sensor technologies, connectivity and data analytics.

Production processes are the initial step toward such intelligence or the representation of the actual world in ones and zeros. The goods, processes, and services of a city acquire a digital representation as they become intelligent, autonomous, networked, and integrated to support ecological and social improvements.

As such, secure transactions and identities within cities can be enabled by distributed ledger technologies. To perform pattern recognition and autonomous system management, Artificial Intelligence (AI) and Machine Learning (ML) are applied. IoT which is something this project is focused on, acts as an interface between the real world and this digital representation of the city. [1]

2.2 Previous Works

2.2.1 ParkS

ParkS [2] is a parking system that monitors parking space availability and uses sensors at entrance and exit points to feed data to the software. The program then determines if the parking lot is full or if there are still places available and generates a physical signal showing this. At certain intervals, transactions are logged, and the data is presented via a web interface. The system aims to use Internet technologies to connect computers to real-life practical applications.

The project makes various technologies, such as Microsoft Visual Studio for development and testing and an ASP server for the web interface. Communication between hardware and software is enabled by making use of a parallel interface. The system assumes that a straightforward switch would be sufficient to cause an event that the application could read in order to determine when a car had entered or departed the parking lot.

The program was divided into two applications: the control panel application, which updates the parking availability in real-time, and the stealth application, which records parking activities and transmits this data to the database every 30 minutes. ASP.NET is the foundation of the web interface. The system utilises databases to store data, and the web interface can access such databases for reports and analysis. [2]

2.2.2 City-Scale Air Quality Monitoring

This article [3] describes a pilot study that investigated the Air Quality of a large city in the UK using low-cost Particulate Matter (PM) sensors. The study deployed six Air Quality IoT devices, each equipped with four different low-cost PM sensors, at two locations within the city. These devices used Long Range Wide Area Networking (LoRaWAN) wireless network transceivers to test city-scale Low-Power Wide-Area Network (LPWAN) coverage.

The Air Quality IoT device is built around a Raspberry Pi 3 Model B with a Power Over Ethernet (PoE) HAT stacked on top, providing both power and network connectivity. A LoRaWAN HAT is also stacked on top of the Raspberry Pi and PoE HAT, providing long-range wireless communication. The device is equipped with four different low-cost PM sensors, which are housed within an enclosure and connected to the Raspberry Pi via USB-serial converters.

The results of the pilot study showed that:

- the physical Air Quality IoT device developed can operate at a city scale.
- some low-cost PM sensors are viable for monitoring Air Quality and for detecting PM trends
- LoRaWAN is suitable for city-scale sensor coverage where connectivity is an issue.

Based on these findings, a larger LoRaWAN-enabled Air Quality sensor network is being deployed across the city of Southampton in the UK. [3]

2.2.3 The MARVEL Project

The MARVEL project [4] is a European Union-funded research initiative that focuses on developing a framework for integrating IoT and AI technologies into smart city applications. The project aims to provide solutions to real-world urban challenges such as traffic congestion, parking management, and public safety.

The project makes use of IoT and AI to enable applications for smart cities. The MARVEL project processes data from IoT devices and sensors put all around a city using AI algorithms. These devices and sensors gather information on a range of topics, including crowd monitoring, parking availability, and traffic flow. The data is then utilized to generate insights that allow improved decision-making and management of city resources.

Many use cases, such as traffic control, public safety, and parking management, are part of the MARVEL project. For instance, the project makes use of AI and IoT technologies to forecast traffic jams and provide drivers with real-time traffic information. The initiative also makes use of ML algorithms to track down and anticipate criminal activities in public areas and notify authorities of potential security threats. Using real-time information gathered from IoT sensors, the parking management technology helps vehicles locate open parking spaces.

By increasing the efficiency and effectiveness of city services, decreasing traffic congestion, strengthening public safety, and decreasing the time and effort needed to find parking, these use cases help to establish smart cities. The MARVEL project offers solutions that let cities manage their resources more effectively, reduce spending, and raise citizen quality of life through the use of IoT and AI. [4]

2.3 Sensors

2.3.1 HC-SR04 Ultrasonic Sensor

The HC-SR04 Ultrasonic Sensor is a popular and widely used sensor for measuring distances in a variety of applications. The sensor has four pins:

1. Power Supply, V_{cc}
2. The Trigger Input Pin which is labelled as `TRIG` which is used to initiate a distance measurement
3. The Echo Output Pin which is labelled as `ECHO` is used to receive the signal back from the object
4. Ground Pin, `GND`

It works by emitting high-frequency sound waves and measuring the time it takes for the waves to bounce back from an object, allowing it to calculate the distance to the object as follows:

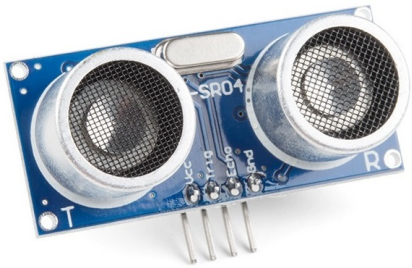

Figure 2.1 The HC-SR04 Sensor
[5]

$$s = \frac{vt}{2} \tag{2.1}$$

where:

$s =$ distance between the sensor and the object by which the sound wave collides with
$v =$ the velocity of the sound wave (340 ms^{-1}) [5]
$t =$ the round trip time of the sound wave

It has a detection range of up to 4 meters, and a resolution of 3mm, providing accurate and reliable distance measurements. [5]

2.3.2 DHT11 Humidity and Temperature Sensor

The DHT11 is a sensor that measures temperature and humidity. It consists of a resistive-type humidity sensor and a thermistor. With an accuracy of $\pm$2°C and $\pm$5% relative humidity, it offers a measurement range of 0–50°C for temperature and 20–90% for relative humidity. It can be powered with 3–5.5 V DC. The sensor has four pins:

1. Power Supply, V_{DD}
2. The `DATA` pin, which is used to send data to some form of processor
3. The `NC` pin - serves no purpose and is not connected to any functionality.
4. Ground Pin, `GND`

Since the sensor only makes use of the `DATA` pin, it makes use of a single-wire communication protocol that consists of a start signal, a response signal and 40 bits of data. The data bits are divided into five bytes: the first two bytes are the integer part of the humidity value, the third and fourth bytes are the integer part of the temperature value, and the fifth byte is a checksum to verify the data integrity. [6]

2.3.3 Gas Detection Modules

In this project, two particular sensors are used — **MQ-135** and **MQ-4**.

The MQ-135 sensor is a sensor used to detect a wide range of gases, including benzene alcohol, Carbon Dioxide (CO_2), Carbon Monoxide (CO) and ammonia, among others [7]. For the purposes of this project, it is being employed to detect CO_2 and CO as part of the AQM System. The MQ-4 sensor is designed to detect various flammable gases in the air, such as Liquid Petroleum Gas, Hydrogen, CO and Smoke among others [8]. In this project, it is utilised to detect CO and smoke for the Fire Detection System.

Both sensors operate based on the principle of gas conductivity, utilising a tin dioxide layer to sense changes in gas concentrations in the surrounding environment. The sensors act as variable resistors that react to the gas concentration, causing changes in resistance. Higher gas concentrations lead to decreased resistance, while lower concentrations result in increased resistance. The gas concentrations can be calculated based on the sensitivity characteristic curve shown in Figure 2.3.

The sensor modules used in this project feature a general gas sensor circuit as depicted in Figure 2.2. In this circuit, R_L represents the load resistor set to 100Ω.

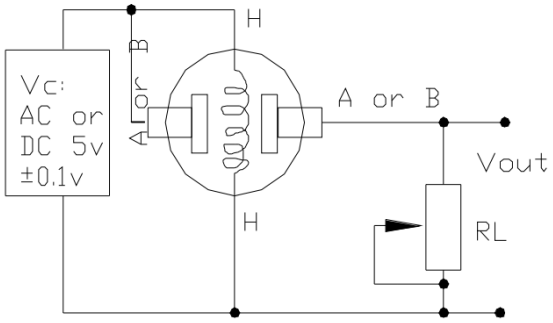

Figure 2.2 General Gas Sensor Circuit

The sensitivity characteristic curves shown in Figure 2.3 illustrate the sensitivity ratio of the sensors. The vertical axis represents the resistance ratio, denoted as:

$$\text{Sensitivity Ratio} = \frac{R_s}{R_o} \tag{2.2}$$

where:

$R_s =$ the resistance of the sensor
$R_o =$ the resistance of the sensor at a known concentration without the presence of other gases

From the curves, it can be observed that the sensitivity ratio of the MQ-135 sensor ($\frac{R_s}{R_o}$) is 3.6ppm for fresh air, while for the MQ-4 sensor, it is 4.4ppm.

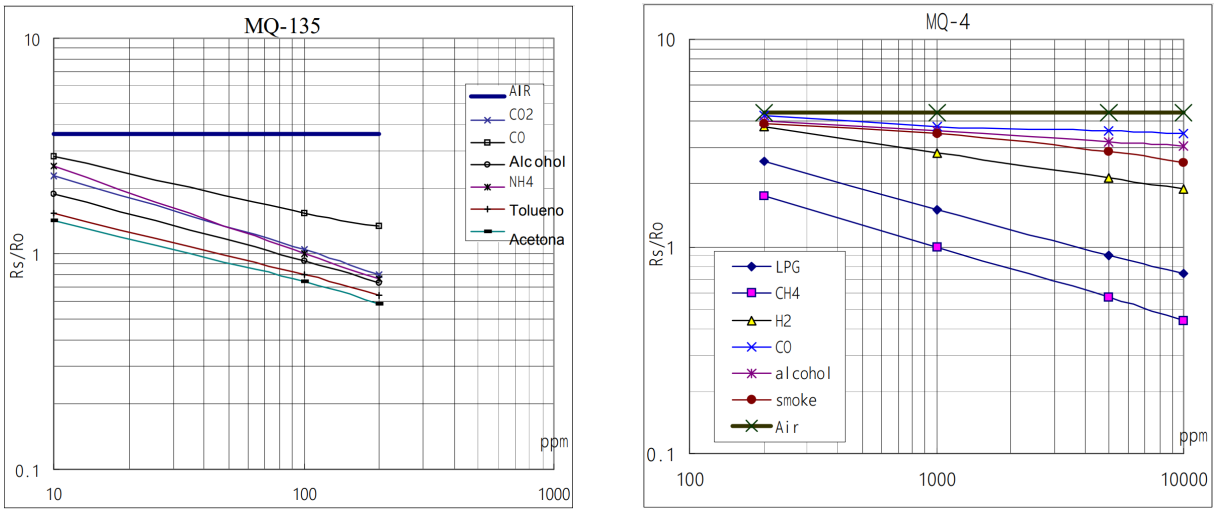

Figure 2.3 Gas Sensor Sensitivity Characteristic Curves.

One can make use of the characteristic curves in Figure 2.3 to create a relation between the resistance ratio and gas concentration. In order to deduce a relationship between the resistance ratio and gas concentration, there will be three assumptions made:

- The relationship between resistance ratio and gas concentration is assumed to be linear
- The relationship will be based on the data provided in both plots. The data for gas concentrations ranges from 10ppm to 200ppm for the MQ-135 and from 200ppm to 10,000ppm for the MQ-4
- As indicated in Figure 2.3, the scale of both plots is logarithmic on both axes

With those assumptions, we come up with the following equations:

$$a = \frac{\log_{10}(\frac{y_0}{y_1})}{\log_{10}(\frac{x_0}{x_1})} \tag{2.3}$$

$$b = \log_{10}(y_2) - a\log_{10}(x_2) \tag{2.4}$$

$$x = 10^{\frac{[\log_{10}(y)-b]}{a}} \tag{2.5}$$

where:

a = slope (gradient) of the line
b = the y-intercept
y_0, y_1, y_2 = points on the y-axis (Resistance Ratio)
x_0, x_1, x_2 = points on the x-axis (ppm)
y = The Resistance Ratio obtained from the sensor
x = Gas Concentration (ppm)

With Equation 2.3 and Equation 2.4, one would be able to obtain a and b. In turn with Equation 2.5, one would be able to determine the gas concentration of a particular gas.

2.4 Arduino MKR NB 1500 Board

The Arduino MKR NB 1500 board is an open-source development board that is designed for IoT applications. It is equipped with the u-blox SARA-R410M-02B module, which is a low-power chipset that supports narrowband communication over the Long Term Evolution for Machines (LTE-M) and Narrowband Internet of Things (NB-IoT) cellular networks. This enables the board to connect to the internet and exchange data with cloud services and other devices. [9]

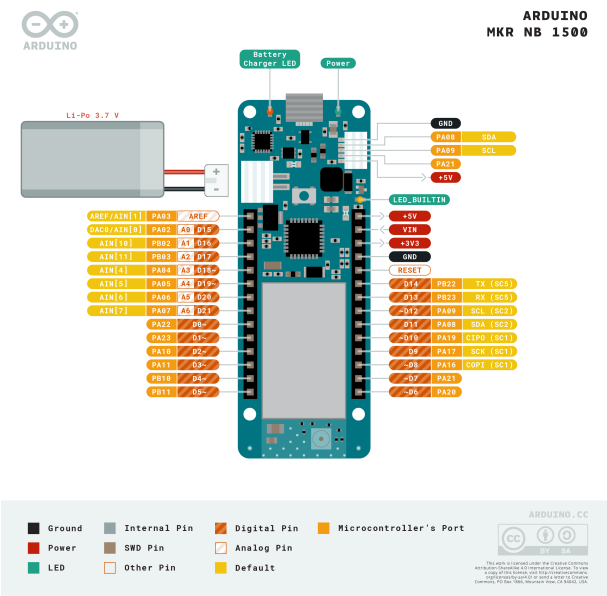

Figure 2.4 The Arduino MKR NB 1500 board [9]

2.5 LPWAN Technologies

LPWAN is a wireless technology for connecting low-bandwidth, battery-powered devices over long distances. It is suitable for IoT and Machine to Machine (M2M)

networks that need low cost and power efficiency. An LPWAN can be private or public, using licensed or unlicensed frequencies and various standards and protocols.

2.5.1 NB-IoT

NB-IoT is an LPWAN which makes use of existing 4G infrastructure to provide low-cost, low-power and reliable communication for a wide range of IoT devices.

NB-IoT operates on the licensed spectrum and is a subset of the 4G standard, providing improved network coverage, longer battery life, and better indoor penetration compared to other wireless communication technologies. It operates in the single narrowband of 200kHz [10], hence the name being Narrowband Internet of Things (NB-IoT).

Advantages of NB-IoT include its low power consumption, which can allow devices to operate for years on a single battery, making it ideal for use in remote or hard-to-reach locations. It also has excellent penetration and coverage, making it suitable for use in indoor environments, underground or inside buildings, and in rural or remote areas. Additionally, it has a higher level of security and reliability compared to other LPWANs, as it benefits from all the security and privacy features of mobile networks. [11]

The main caveat of NB-IoT is that it operates on a licensed spectrum, which can be expensive and may limit access to the technology for smaller companies or startups looking to incorporate IoT into their operations.

2.5.2 LoRaWAN

LoRaWAN is an LPWAN technology that uses Long Range (LoRa) modulation to provide long-range low-bandwidth communication for IoT applications. It operates in the unlicensed spectrum, which makes it more accessible and cost-effective for companies and developers.

It uses a star network topology, where end devices communicate with a gateway, which then sends the data to a network server [12].

The LoRaWAN specification defines three different device classes [13]:

- **Class A**: All LoRaWAN devices must implement Class A. Class A communication is always initiated by the end-device and supports bi-directional communication (uplink and downlink).
- **Class B**: In addition to the Class A initiated receive windows, Class B devices open scheduled receive windows for receiving downlink messages from the

network server. Class B devices have lower latency than Class A end devices but shorter battery life.

- **Class C**: Extend Class A by keeping the receive windows open unless they are transmitting. This allows for low-latency communication but is much more energy consuming than Class A devices.

One of the key advantages of LoRaWAN is its long-range communication capabilities, which can reach up to 15 kilometres in rural areas and up to 5 kilometres in urban areas [14]. This makes it ideal for applications where devices need to be placed in remote or hard-to-reach locations. Additionally, LoRaWAN has low power consumption, which allows devices to operate for long periods of time on a single battery, reducing maintenance costs.

One of the main issues that come with LoRaWAN is that since it operates in the unlicensed spectrum, it can be subject to interference from other devices that operate in the same frequency range. To mitigate this, a duty cycle limit is put in place that has to be adhered to in order to not cause interference with other devices operating in the same frequency band. In Europe, the duty cycle limit is typically 0.1% and 1.0% per day, depending on the channel [15].

2.6 Application Layer Protocols

An application layer protocol is the topmost layer of the OSI seven-layer model[1]. It provides an abstraction of lower-level protocols which define how different applications communicate over the internet, irrespective of their physical medium.

2.6.1 CoAP

CoAP is a lightweight, low-power communication protocol designed for constrained devices and networks. It is designed to be used in a client-server model.

The main advantage of CoAP is its low overhead, which is achieved by using a simple binary format for messages and limiting the number of methods and options available. CoAP uses a simple request-response model, similar to Hypertext Transfer Protocol (HTTP), with four basic methods: `GET`, `PUT`, `POST`, and `DELETE`. In addition, it provides support for observing resources, which allows a client to be notified when a resource changes, without the need for continuous polling.

[1] The OSI layer model is a conceptual framework that defines the functions and interactions of different network protocols, dividing them into seven distinct layers for efficient communication between computer systems.

Another advantage of CoAP is its support for security mechanisms, such as Datagram Transport Layer Security (DTLS). This provides end-to-end security for communications over the network, which is important for many IoT applications.

CoAP also includes support for proxies and gateways, which allows it to interoperate with existing HTTP infrastructure. This makes it possible to use CoAP with existing web services, without the need for modification.

CoAP makes use of User Datagram Protocol (UDP), due to it being more faster and efficient than Transmission Control Protocol (TCP). As messages sent through UDP may arrive out of order, appear duplicated, or go missing without notice [16] [17], CoAP implements a lightweight reliability mechanism, without having to implement the full feature set that a much more reliable transport-layer protocol such as TCP has. [18]

2.6.2 MQTT

MQ Telemetry Transport (MQTT) is a lightweight, publish-subscribe messaging protocol designed for use in IoT and M2M applications. It is based on the client-server model and operates over TCP, allowing it to be used over a variety of networks, including wireless and cellular networks.

MQTT provides a simple and efficient way for devices to communicate with each other, using a publish-subscribe model. In this model, devices can publish messages to a specific topic, which are then delivered to all subscribed devices that are listening to that topic. This allows devices to send and receive messages in a highly efficient and scalable manner.

One of the advantages of MQTT is its low overhead, which is achieved through the use of a simple binary protocol. This makes it well-suited for use in low-power and constrained environments, where resources are limited.

MQTT also includes support for features such as last will and testament, which allows devices to be notified when a device becomes unavailable, and retained messages, which allows devices to receive the most recent message published to a topic, even if they were not subscribed at the time the message was sent. [19]

2.7 Software

2.7.1 InfluxDB

InfluxDB is a Time-Series Database (TSDB) designed for handling large amounts of time-series data. It is frequently used in IoT applications because it can effectively store and handle huge amounts of data produced by these sensors. Its efficient indexing and compression algorithms also reduce storage requirements and lower costs for long-term data retention. Together with this, InfluxDB provides a number of integrations with additional instruments frequently used in IoT applications, like Telegraf for data gathering and Grafana for data visualisation. It makes use of *Flux*, a functional data scripting language for querying and manipulating data. [20]

2.7.2 Grafana

Grafana is an open-source data visualisation and monitoring tool. It offers a customizable and adaptable framework for building dashboards and visualisation that may assist users in learning more about their data. Grafana is the perfect tool for combining data from several sources since it supports a broad variety of data sources, including databases, Application Programming Interfaces (APIs), and IoT devices. Moreover, it provides a selection of data visualisation choices, such as graphs, gauges, and tables, all of which may be adjusted to match the needs of individual users. [21]

2.7.3 Docker

Docker is a software platform that allows for the building, running and sharing of applications using containers. Containers are isolated environments that package up the code, dependencies and configuration of an application, making it easy to deploy and run on any system. Docker uses a client-server architecture, where the Docker daemon runs on a host machine and communicates with the Docker client, which is a command-line tool or a graphical user interface. [22]

Docker also provides a registry service called Docker Hub, where you can distribute and make use of other images. Some of the advantages of using Docker are:

- **Ease of Deployment**: One can create a consistent and portable environment for an application, regardless of the underlying infrastructure. The deployment process can also be automated using tools such as Docker Compose or Kubernetes.

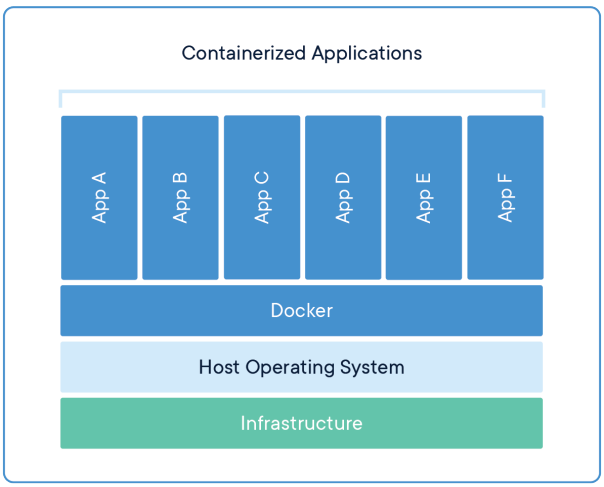

Figure 2.5 A Container-Based Architecture [22]

- **Encapsulation and Modularity**: An application can be broken into smaller and independent components, each with its own container.
- **Resource Efficiency**: Multiple containers can be run on a single host machine, sharing the same operating kernel and resources.

3 Specification and Design

This chapter introduces a top-level outlook of the system architecture, communication protocol, data collection pipeline, and physical nodes devised for the model.

3.1 System Architecture and Physical Nodes

The approach consists of a centralised server that uses a star topology to receive data from different physical nodes. Such architecture is shown in Figure 3.1. Using NB-IoT and CoAP, each physical node sends data to the server, which is handled accordingly.

The Car Park Sensor is a proof of concept based on a single parking spot. It detects the presence of a vehicle and sends data to the server.

The AQM System sends periodic data that includes temperature (°C), humidity level, CO levels, and CO_2 levels. This data is sent to the server, allowing for the monitoring of air quality.

The Fire Detection System aims to routinely send data which may be able to correlate to a possible fire such as temperature, humidity, presence of IR and CO and CO_2 levels.

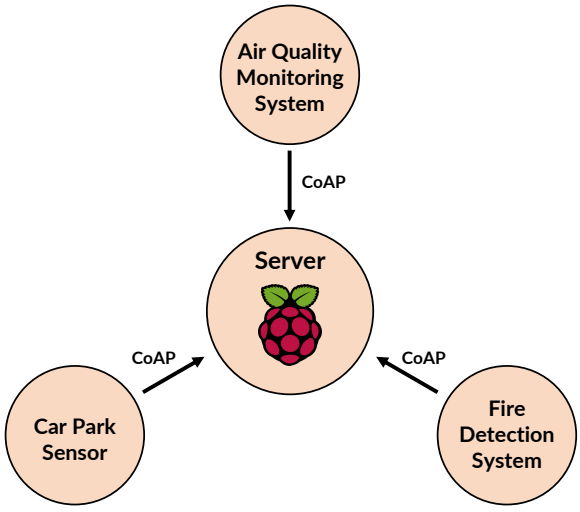

Figure 3.1 System Architecture

3.2 LPWANs and Application Layer Protocols

The wireless technology and communication protocol used to implement the Smart City model are essential for assuring the system's dependability and effectiveness. The

communication protocol used was the CoAP protocol, and the wireless communication channel chosen was NB-IoT.

The following factors led to the decision to choose CoAP as the communication protocol. Firstly, due to the usage of UDP and its connectionless design, CoAP is considered to be more efficient than MQTT. As a result, CoAP is more power-efficient, which is crucial for IoT devices because their battery life may be constrained. The requirement for lossless and orderly delivery is not there in CoAP, which lowers overhead. When compared to MQTT, which needs TCP/IP level connection and session settings, CoAP does not. Finally, CoAP is more data-efficient than MQTT as CoAP requires less bandwidth [23].

Initially, LoRaWAN was chosen over NB-IoT due to its lower power consumption, which aligned with the project's objectives. LoRaWAN's advantage lies in its ability to operate on low power, enabling long battery life for IoT devices. However, despite this benefit, the extensive hardware and infrastructure requirements associated with LoRaWAN became a significant drawback. Deploying LoRaWAN necessitates specialised infrastructure development and ongoing maintenance, which went beyond the project's scope.

On the other hand, NB-IoT offered a compelling alternative. Its selection as the wireless communication standard was primarily driven by the accessibility of infrastructure and hardware. Leveraging the existing 4G infrastructure, NB-IoT eliminated the need for costly infrastructure modifications. This advantage simplified the deployment process and reduced the overall implementation costs. Consequently, NB-IoT emerged as the preferred choice, allowing the project to leverage the readily available infrastructure without compromising on essential requirements.

3.3 Data Collection Pipeline

A data collection pipeline was created utilising a Raspberry Pi 4B as a server to provide efficient data gathering, storage, and visualisation from data gathered by the physical nodes. Such data collection pipeline is shown in Figure 3.2. A CoAP server, InfluxDB, and Grafana are all operating in separate Docker containers as part of the pipeline. The utilisation of Docker containers offers numerous advantages, primarily centred around ease of deployment. Such containers allow for the encapsulation of the server software within portable containers, which in turn create a consistent and reproducible environment across different platforms.

Data received from the physical nodes is received by the CoAP server, which is

developed in Python. It sends the information to the InfluxDB container, where it is stored in a time-series database. This database instance provides an API for querying and accessing the data collected from the physical nodes.

Data visualisation is handled by the Grafana container, which uses an InfluxDB instance as a data source. Users are able to track the progress of the Smart City model implementation in real-time thanks to the dashboard that displays the aggregated data.

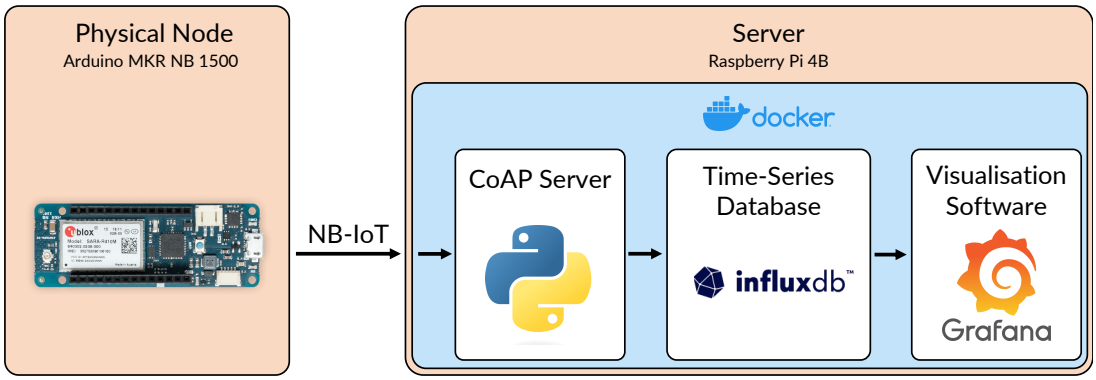

Figure 3.2 Data Collection Pipeline

The data collection pipeline ensures that the data generated by the physical nodes is effectively stored and viewed. The pipeline can be simply deployed and maintained using Docker containers, and each component may be upgraded or replaced separately without compromising the pipeline's overall functionality.

3.4 Physical Nodes

The Physical Nodes were developed using the C and C++ programming languages. The MKRNB library [24], which provides ready-built functions used for Narrowband communication, was made available via the Arduino platform. To help with the development process, other libraries [25–27] were also used.

After gathering the data, the modem is turned on to send a JavaScript Object Notation (JSON) document to the server. The Physical Node enters sleep mode once the data has been transmitted for a set duration, which varies depending on the type of node. All Physical Nodes transmit the specific data together with their Node Type and ID.

The Car Park Sensor is quasi-event-driven, which means it sends data when an event takes place. It uses the HC-SR04 Ultrasonic Sensor to detect whether a vehicle is parked in the sensor's range. It then sends data on whether a vehicle is on the sensor itself. Its sleep duration will be for a minute.

The AQM System is based solely on periodic data collection. It measures CO and CO_2 levels using a MQ135 Gas Detection Module, and temperature and humidity levels using a DHT11 Humidity and Temperature Sensor. It sends data concerning the current temperature, humidity level, and CO and CO_2 levels. Its sleep duration will be for 30 minutes.

The Fire Detection System make use of periodic data collection. It uses a Flame Sensor Module, MQ4 Gas Detection Module for CO and smoke detection, and a DHT11 Humidity and Temperature Sensor for Temperature and Humidity measurement. It sends data such as the temperature, if infrared (IR) is detected, and whether smoke is detected. Its sleep duration will be for a minute and a half.

4 Implementation

4.1 Hardware

4.1.1 Car Park Sensor

The sensor used for this particular physical node is the HC-SR04 Ultrasonic Sensor.

As indicated in Figure 4.1, a voltage divider was used with the `ECHO` pin due to the fact that the I/O voltage for the MKR NB 1500 was 3.3V [9] and the HC-SR04 ultrasonic sensor operated at a 5V logic level. This meant that if directly connected to the 3.3V I/O pin of the MKR NB 1500, it would exceed the safe voltage level and potentially damage the board. The following equation demonstrates the conversion in voltage levels:

$$V_{out} = V_{in} \times \frac{R_2}{R_1 + R_2} \tag{4.1}$$

where:

V_{out} = the output voltage (3.3V)
V_{in} = the input voltage (5V)
R_1, R_2 = Resistances (1kΩ, 2kΩ respectively)

The `ECHO` and `TRIG` pins are connected to the `D7` and `D6` on the MKR NB 1500 board.

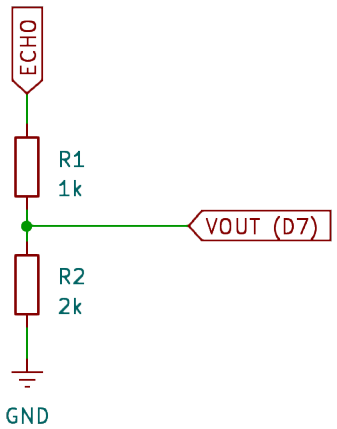

Figure 4.1 HC-SR04 Circuitry

4.1.2 Air Quality Monitoring System

The fundamental components for this particular physical node are the DHT11 Humidity and Temperature Sensor, and the MQ-135 Gas Detection Module.

According to the DHT11 datasheet [6], a 5kΩ pull-up resistor is recommended, if the connection is shorter than 20 metres. This was done by making use of two 10kΩ resistors in a parallel configuration. On the other hand, the MQ-135 does not require any special wiring requirements.

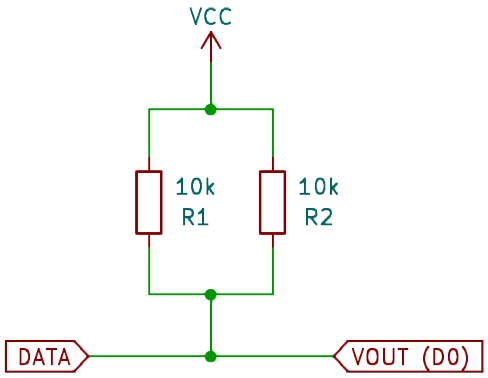

Figure 4.2 DHT11 Circuitry

The `DATA` pin of the DHT11 sensor and Analog output of the MQ-135 are connected to `D0` and `A1` on the MKR NB 1500 board.

4.1.3 Fire Detection System

The fundamental components for this particular physical node are the DHT11 Humidity and Temperature Sensor, the MQ-4 Gas Detection Module, and the Flame Sensor.

Similar to the AQM System's hardware configuration, A 5kΩ pull-up resistor is used by putting 10kΩ resistors in a parallel configuration. The MQ-4 and Flame Sensor Module do not require any special wiring requirements.

The `DATA` pin of the DHT11 sensor, Analog output of the MQ-135 and Digital output of the Flame sensor are connected to `D0`, `A1` and `D6` on the MKR NB 1500 board.

4.2 Software

4.2.1 Averaging Sensor Readings

Due to fluctuating outputs from sensors, an averaging routine was implemented in order to obtain a representative value from a small set of data obtained within a given time span rather than obtaining a single value. As shown in Listing 4.1, the function takes in the array containing the values and the number of elements in said array. This routine is employed in all physical nodes and used on all sensors.

```
float averageArray(int *array, int elems){
    float sum = 0L;
    for (int i = 0; i < elems; i++)
        sum += array[i];
    return sum/elems;
}
```

Listing 4.1 Averaging Function

4.2.2 Communication with Server

When it comes to communication, the physical nodes project makes use of various libraries in order to enable Narrowband communication [24, 28], make use of the CoAP protocol [25], obtaining the CoAP server's Internet Protocol (IP) address by using the HTTP protocol [26] and to serialise JSON documents [27].

The following is the communication routine employed on the physical nodes:

1. Serialising the JSON document
2. Preparing the Modem for NB-IoT connectivity
3. Connecting to the NB-IoT Service Provider
4. Obtaining the IP address of the CoAP server
5. Sending the JSON document through CoAP by making a `PUT` request to the server
6. Disconnecting from the Service Provider by turning off the Modem
7. Wait for the JSON document to be sent to the server
8. Putting the physical node to sleep

4.2.3 Car Park Sensor

As highlighted in subsection 2.3.1, there is a particular sequence in order to measure a distance:

1. Supply a pulse which lasts for 10 µs to the `TRIG` pin
2. Start timing and wait for a pulse to arrive from the `ECHO` pin
3. Stop timing and store time it took for pulse to arrive to the `ECHO` pin
4. Calculate the distance using Equation 2.1

The function in Listing 4.2, `getUltrasonicReading()` demonstrates this particular sequence:

```
int getUltrasonicReading(){
    long duration;
    int distance;

    digitalWrite(TRIG_PIN, LOW);
    delayMicroseconds(2);
    digitalWrite(TRIG_PIN, HIGH);
    delayMicroseconds(10);
    digitalWrite(TRIG_PIN, LOW);

    // Distance Calculation
    duration = pulseIn(ECHO_PIN, HIGH);
    distance = duration * 0.034 / 2;
    return distance;
}
```

Listing 4.2 Obtaining a Distance Reading from the Ultrasonic Sensor

After taking multiple readings using `getUltrasonicReading()` shown in Listing 4.2, the averaging function in Listing 4.1 was used in order to get a value in which one could decide whether a vehicle is actually parked on the sensor itself.

The average is compared to a set threshold. If the average distance is lower than the set threshold then it can be determined that currently, a vehicle is parked on the sensor itself. Else, it is determined that no vehicle is currently parked on the sensor.

Given that this sensor is event-based, the sensor sends data depending on the change of state. If there is a change of state, the indication of a change of state is sent to the CoAP server in the form of the JSON document shown in Listing 4.3. If no change in state is detected, the physical node automatically goes to sleep.

```
{
```

```
    "nodetype": "CPS",
    "id": "cps0001",
    "data": {
        "isCarParked": true
    }
}
```

Listing 4.3 Car Park Sensor JSON document

As specified in section 3.4, other than specifying whether a vehicle is present on the sensor, the Node Type and its ID are also provided in order to identify the specific sensor, in case of a larger scale implementation.

4.2.4 Air Quality Monitoring System

In order to simplify development, libraries were used in order to get sensor readings from the DHT11 [29] and the MQ-135 [30] sensors.

The functionality of the DHT library [29] abstracts the overall single-wire communication routine with the DHT11 sensor, which is highlighted in subsection 2.3.2. On the other hand, the unified library for MQ sensors [30] abstracts the whole process of calculating Gas Concentrations. Nonetheless, one would need to input a and b in order for the equation to make use of Equation 2.5 properly.

By making use of Equation 2.3 and Equation 2.4, which were highlighted in subsection 2.3.3, a and b were obtained. The coefficients are shown in Table 4.1.

Table 4.1 Coefficients used for MQ-135 Sensor

	a	b
CO	− 0.220061597	0.653581876
CO_2	− 0.366725791	0.763607977

Using the coefficients in Table 4.1, one would be able to implement a method which is able to read the gas concentrations, as shown in Listing 4.4. The library has functionality which reads the Analog data from the sensor itself and calculates the gas concentration of a gas based on the coefficients provided, using Equation 2.5.

```
float getCO(MQUnifiedsensor &mq135){
  mq135.setA(-0.220061597);
  mq135.setB(0.653581876);
  float co_ppm = mq135.readSensor();
  return co_ppm;
}

```

```
float getCO2(MQUnifiedsensor &mq135){
  mq135.setA(-0.366725791);
  mq135.setB(0.763607977);
  float co2_ppm = mq135.readSensor();
  co2_ppm += 400; // CO2 present in the atmosphere is around 400 PPM.
  return co2_ppm;
}
```

Listing 4.4 Obtaining Gas Concentration Readings from the MQ-135 sensor

After taking multiple temperature, humidity and gas concentration readings, the averaging function in Listing 4.1 was used in order to get a representative value for a set amount of time.

Given that this physical node sends data periodically, data is sent in regular intervals to the CoAP server in the form of the JSON document shown in Listing 4.5.

```
{
    "nodetype": "AQMS",
    "id": "aqms0001",
    "data": {
        "temperature_c": 1,
        "humidity_percent": 1,
        "co_level_ppm": 1,
        "co2_level_ppm": 1
    }
 }
```

Listing 4.5 AQM System JSON document

4.2.5 Fire Detection System

Similar to the AQM System, libraries were used in order to get sensor readings from the DHT11 [29] and the MQ-4 [30] sensors.

The coefficients used in the MQ library are shown in Table 4.2.

Table 4.2 Coefficients used for MQ-4 Sensor

	a	b
CO	− 0.05849699	0.75427267
Smoke	− 0.036579755	0.6076452

Using the coefficients in table Table 4.2, one would be able to implement a method which is able to read the gas concentrations for CO and Smoke, as shown in Listing 4.6. The library has functionality which reads the Analog data from the sensor itself and

calculates the gas concentration of a gas based on the coefficients provided, using Equation 2.5.

```
float getCO(MQUnifiedsensor &mq4){
  mq4.setA(-0.05849699);
  mq4.setB(0.75427267);
  float co_ppm = mq4.readSensor();
  return co_ppm;
}

float getSmokePPM(MQUnifiedsensor &mq4){
  mq4.setA(-0.036579755);
  mq4.setB(0.6076452);
  float smoke_ppm = mq4.readSensor();
  return smoke_ppm;
}
```

Listing 4.6 Obtaining Gas Concentration Readings from the MQ-4 sensor

After taking multiple temperature, humidity and gas concentration readings, the averaging function in Listing 4.1 was used in order to get a representative value for a set amount of time.

Given that this physical node sends data periodically, data is sent in regular intervals to the CoAP server in the form of a JSON document shown in Listing 4.7.

```
{
  "nodetype": "FDS",
  "id": "fds0001",
  "data": {
    "temperature_c": 1,
    "humidity_percent": 1,
    "co_level_ppm": 1,
    "smoke_level_ppm": 1,
    "isIRDetected": true
  }
}
```

Listing 4.7 Fire Detection System JSON document

4.2.6 CoAP Server

It is based on the *aiocoap* Python package [31] and a *InfluxDB* Python Client [32] being used in order to interface with the TSDB. The *aiocoap* package helps in abstracting CoAP specifications [18] and hence aiding in the development of the server. The CoAP server consists of four resources, three for each physical node and one used for testing purposes.

All four resources are built on top of a basic resource, which is shown in Listing 4.8. It provides a foundational implementation of a CoAP resource with observation capabilities. It should be mentioned that in their current form, the physical nodes, which can be referred to as clients, only make uplink communication (i.e send data to the server). Furthermore the physical nodes do not make use of the current observation capabilities that a client would be able to make use of as part of a `GET` request to the CoAP server.

```
class BasicResource(resource.ObservableResource):

    def __init__(self):
        super().__init__()

        self.has_observers = False
        self.notify_observers = False
        self.influx_client = Influx(influxdb_bucket, influxdb_url,
    influxdb_token, influxdb_org)

    def notify_observers_check(self):
        while True:
            if self.has_observers and self.notify_observers:
                print('notifying observers')
                self.updated_state()
                self.notify_observers = False

    def update_observation_count(self, count):
        if count:
            self.has_observers = True
        else:
            self.has_observers = False
```

Listing 4.8 Basic CoAP Resource

In Listing 4.9, is an implementation of one of the four resources. In this case, it is being used for the Car Park Sensor. It also acts as a child class for the basic resource in Listing 4.8. It also holds member variables for the Node ID, whether a Vehicle is parked on the sensor and an *InfluxDB* `Sensor` object.

```
class CPS_Resource(BasicResource):

    def __init__(self):
        super().__init__()

        self.node_id = ""
        self.status_isCarParked = 0
```

```
8         self.influx_sensor = Sensor("cps", self.influx_client)
```

Listing 4.9 Car Park Sensor CoAP Resource

Listing 4.10 shows the implementation of how the server would respond to a `GET` request to `CPS_Resource`. It grabs the data held in member variables and serialises it as part of the JSON document, which in this case was specified in Listing 4.3. It then returns the serialised JSON document as a payload.

```
1  async def render_get(self, request):
2
3      # Serialise JSON Document with Data from Member Variables
4      json_obj = {
5          "nodetype": "CPS",
6          "id": {self.node_id},
7          "data":
8          {
9              "isCarParked": bool({self.status_isCarParked})
10         }
11     }
12     payload = json.dumps(json_obj)
13     payload = payload.encode('utf8')
14
15     return aiocoap.Message(payload=payload)
```

Listing 4.10 Car Park Sensor Resource `GET` request handler

Listing 4.11 shows the implementation of how the server would respond to a `GET` request to `CPS_Resource`. It parses the received payload as a JSON document and stores the data specified in the received document in the member variables of the `CPS_Resource` class. The data is sent to the TSDB which the server is configured to.

```
1  async def render_put(self, request):
2      payload = request.payload.decode('utf8')
3      print(payload)
4      payload_json = json.loads(payload)
5      self.node_id = payload_json['id']
6      self.status_isCarParked = int(payload_json['data']['isCarParked'] ==
       True)
7
8      self.influx_sensor.add_value("node_id", self.node_id)
9      self.influx_sensor.add_value("isCarParked", self.status_isCarParked)
10     self.influx_sensor.write()
11
12     return aiocoap.Message(code=aiocoap.CHANGED, payload=payload.encode('
       utf8'))
```

Listing 4.11 Car Park Sensor Resource `PUT` request handler

Each resource has its own endpoint, meaning that on that the physical nodes or a CoAP client should be configured to send JSON documents to addresses with the format specified in Listing 4.12

```
coap://<Domain>:<Port>/<Endpoint>
```

Listing 4.12 CoAP server address format with endpoints specified

The CoAP server is packaged as a Docker container as specified in subsection 4.2.8.

4.2.7 IP Lookup Server

At the time of writing, the CoAP library being used [25] does not support inputting hostnames. To counteract this issue, an HTTP server was created as a workaround to provide the server's Public IP Address.

The IP Lookup server is based on the Flask framework [33]. It looks up its own Public IP Address through a Public IP Lookup API [34]. The public lookup server returns the address obtained from the Public IP Lookup API.

The IP Lookup server is packaged as a Docker container as specified in subsection 4.2.8.

4.2.8 Docker

The configuration for starting all containers is stored in a Docker Compose file. In the case of containers which need to be built from scratch, extra configuration is made in their own respective Dockerfiles. All containers are set to automatically restart in case of any interruptions, ensuring their availability.

Network Configuration

The network configuration is instrumental in enabling communication between the containers. Three networks with a bridge configuration have been defined:

- `fyp_backend`: Facilitates communication between the CoAP server and the TSDB
- `fyp_frontend`: Facilitates communication between the TSDB and Grafana
- `fyp_ip_lookup`: Specific to the IP Lookup Server

This network isolation ensures secure and controlled communication between the relevant services.

Volume Configuration

Docker volumes have been employed for containers which require persistent storage:

- `influxdb_volume`: Ensures that databases and configuration within the TSDB remain intact
- `grafana_volume`: Used to preserve Grafana's configuration and dashboards

CoAP Server

The CoAP server is based on the `python:3.8-slim` image and is built using a dedicated Dockerfile. As stated earlier, the server makes use of the `fyp_backend` network, enabling communication with the TSDB.

IP Lookup Server

The IP Lookup server is based on the `python:3.7-slim` image and is built using a dedicated Dockerfile. As stated earlier, the server makes use of the `fyp_ip_lookup` network, which prevents any form of interoperability with the other containers.

InfluxDB

The InfluxDB container is based on the official `influxdb:2.6.1` image. The container makes use of the `fyp_backend` and `fyp_frontend`, ensuring interoperability between data being received from the CoAP server and data being requested by the Grafana container. To ensure persistent storage, the `influxdb_volume` volume is mounted to the InfluxDB container. The container makes use of pre-configured environmental variables, which define settings such as database schemas, authentication options, and storage and retention policies. This is done in order to get out-of-the-box support for communication with the CoAP server upon building the image, without additional configuration, post-build.

Grafana

The Grafana container is based on the official `grafana/grafana:9.3.6` image. The container makes use of the `fyp_frontend`, enabling communication with the TSDB. To ensure persistent storage, the `grafana_volume` volume is mounted to the Grafana container. The container makes use of pre-configured environmental variables, which define settings related to sign-in procedures. Additional setup is required post-build in order to enable communication with the TSDB and configure the dashboard.

5 Evaluation and Results

This chapter will go over the testing process, which plays a vital role in ensuring the functionality, reliability and efficiency of the developed system. Based on the data gathered from the testing process, the system's alignment with the original aims and objectives will also be discussed.

5.1 Integration Testing

Throughout the development stage, each component within the project underwent continuous testing to verify that each component works as it was intended in the project's specification. In line with this testing methodology, integration testing takes precedence as it focuses on verifying the interaction and interoperability among the individual components.

This mode of testing will consist of sending a basic JSON document consisting of a pseudo-random number, as shown in Listing 5.1.

```
{
  "nodetype": "TEST",
  "id": "test0001",
  "data": {
    "testValue": 1.1
  }
}
```

Listing 5.1 Test JSON document

Based on this mode of testing, the following can be evaluated:

- The test node manages to communicate with the IP Lookup server to get the CoAP server's IP address
- The CoAP server receives the JSON document
- The data parsed from the JSON document is being forwarded to the database and the visualisation system

Figure 5.1 shows the Serial Output from the test node, which indicates that it has managed to make contact with the server by getting the server's address from the IP Lookup Server and sent the JSON document to the CoAP server.

Figure 5.2 shows that the CoAP server has received the payload containing the JSON document from the physical node.

```
Waiting for modem to get ready...done.
Setting Radio Access Technology to NB-IoT...done.
Applying changes and saving configuration...done.
Modem ready, turn radio on in order to configure it...done.
Check attachment until CSQ RSSI indicator is less than 99...done.
Connecting to ISP...done.
Sending packet to CoAP Server on [REDACTED IP ADDRESS]
{"nodetype":"TEST","id":"test0001","data":{"testValue":24}}
done.
```

Figure 5.1 Serial Output from Test Node

```
(04/07/2023 14:16:40 +0200 CEST) Incoming message <aiocoap.Message at
    0x7f984a9fa0: CON PUT (MID 17866, empty token) remote
    <UDP6EndpointAddress [REDACTED IP ADDRESS] (locally 172.22.0.2
    eth0)>, 2 option(s), 59 byte(s) payload>
(04/07/2023 14:16:40 +0200 CEST) New unique message received
(04/07/2023 14:16:40 +0200 CEST) Payload Received is as follows:
    b'{"nodetype":"TEST","id":"test0001","data":{"testValue":24}}'
(04/07/2023 14:16:40 +0200 CEST) Payload from test0001: TEST PACKET
    RECEIVED
```

Figure 5.2 Output from CoAP Server Logs

Figure 5.3 shows that the visualisation system has a new data point on the time-series database. This means that the CoAP server has parsed the JSON document and forwarded the data to the database which was ultimately forwarded to the visualisation system.

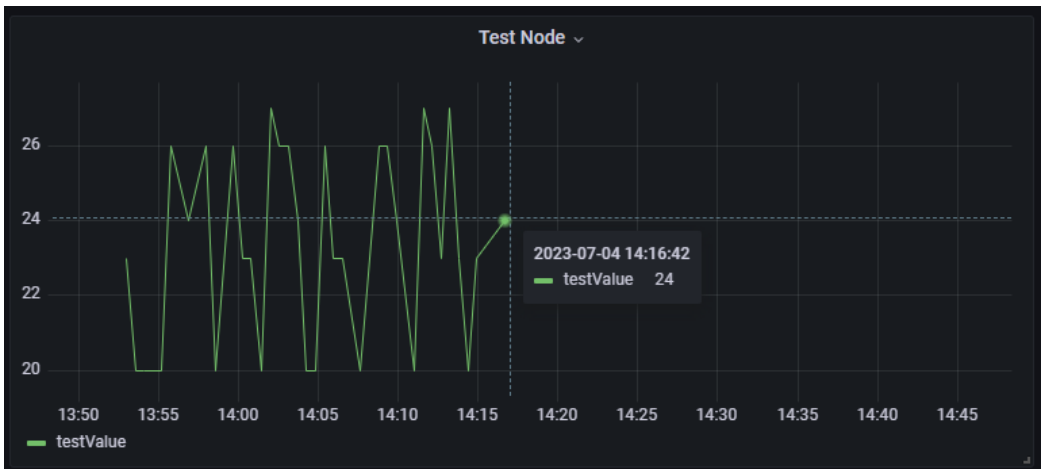

Figure 5.3 Visualisation System Outputting Data

5.2 Load Testing

This section will address the evaluation of the server's performance given all three physical nodes sending data to the server. This will help us understand how the system

handles increased workloads and whether it can sustain optimal performance levels under such conditions.

This was done through an automated load-testing process of simulating three physical nodes. This is done by sending HTTP `GET` requests to the IP Lookup server three times. Then, three CoAP `PUT` requests (one for each physical node) which consist of predefined JSON documents are sent to the server. While all of this is done, another script is executed on the server which monitors the load on system resources such as CPU, memory and disk usage.

Figure 5.4 shows the usage of system resources over the timespan of 20 minutes. It also indicates when the HTTP requests and CoAP messages were received on the server. Table 5.1 provides some descriptive statistics on the data which is presented in Figure 5.4.

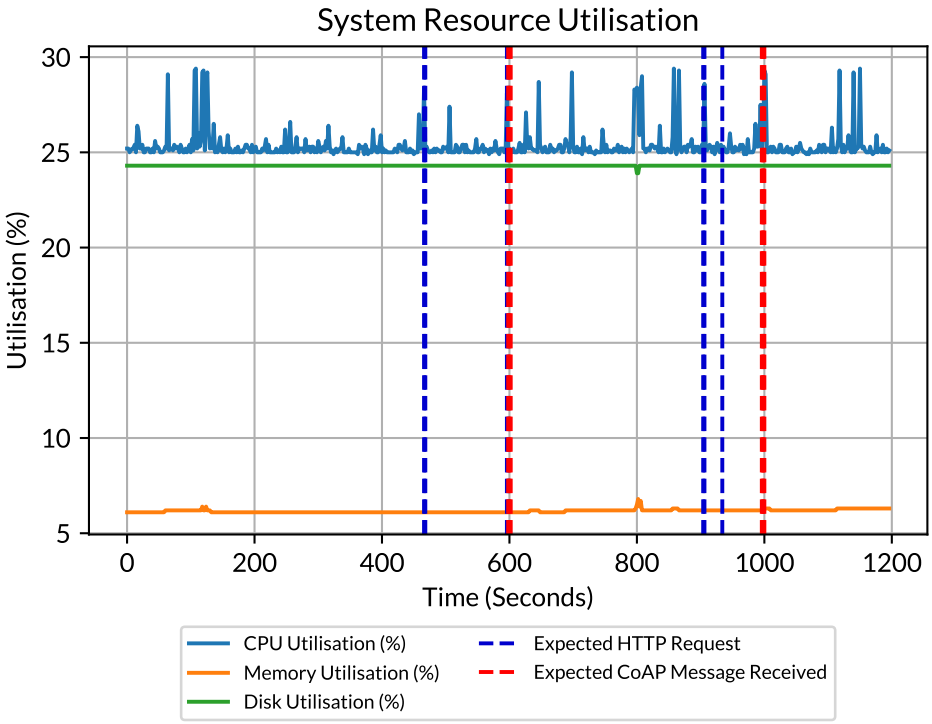

Figure 5.4 System Resource Utilisation Plot

	CPU (%)	Memory (%)	Disk (%)
Mean	25.35	6.16	24.30
Standard Deviation	0.81	0.08	0.02
Minimum	24.90	6.10	23.90
Maximum	29.40	6.80	24.30

Table 5.1 Descriptive Statistics for Resource Utilisation

5.3 Battery Consumption

A critical aspect of evaluating an IoT-based system lies in assessing the battery consumption of any components which are deployed in the field. By quantifying the power consumption of the physical nodes, one can assess their efficiency and look at where one can optimise their performance to ensure longer battery life.

This was done by measuring current for three transmission-sleep cycles by using the INA219 DC Current Sensor Breakout Board and creating a data set from said measurements. These measurements will then be analysed for further evaluation of battery consumption. It should be noted that the physical nodes are currently tested on 10,000mAh power banks.

5.3.1 Car Park Sensor

Given that the Car Park Sensor is event-based, measurements were taken on two types of cycles. One of which, consists of a constant change in state and the other being no detection of a state change.

Figure 5.5 shows the variation in current across three transmission-sleep cycles. In this case, the average current load was found to be **32 mA**.

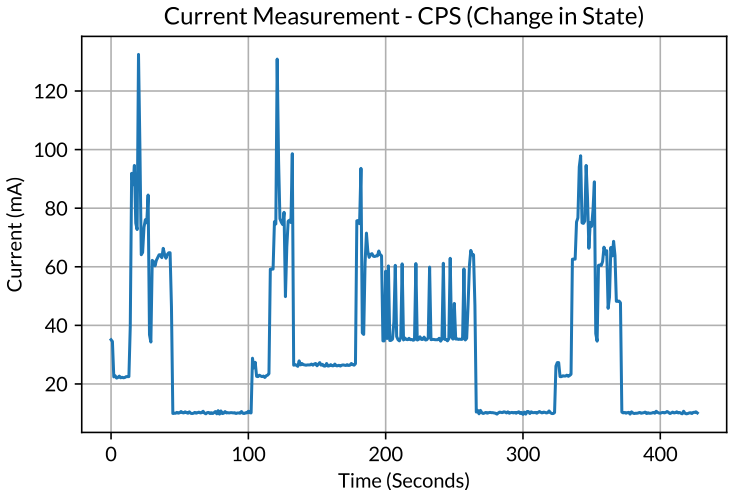

Figure 5.5 Car Park Sensor Battery Consumption Plot (Change in State)

Figure 5.6 shows the variation in current across three cycles. The average current load for a constant state was found to be **12 mA**.

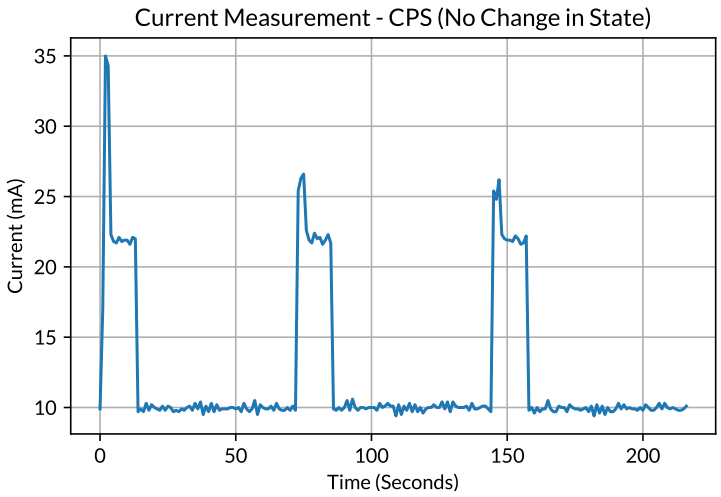

Figure 5.6 Car Park Sensor Battery Consumption Plot (Idle State)

5.3.2 Air Quality Monitoring System

Figure 5.7 shows the variation in current across three transmission-sleep cycles. The average current load was found to be **41 mA**. With a 10,000 mAh battery, the system has a battery life of around **242 hours** or **10 days**.

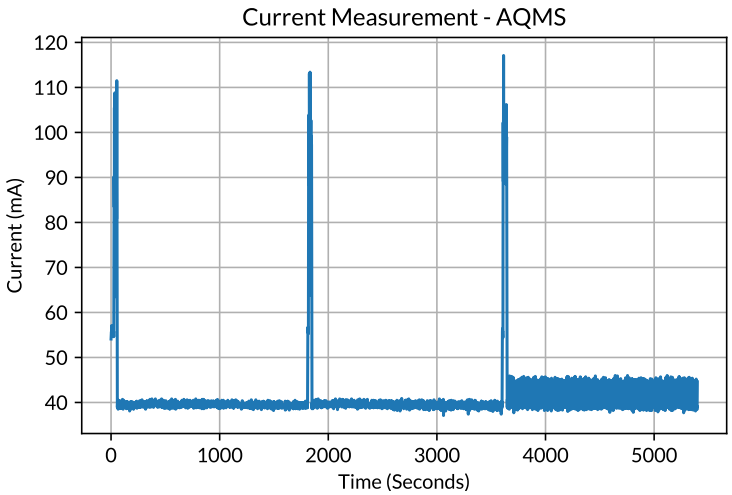

Figure 5.7 AQM System Battery Consumption Plot

5.3.3 Fire Detection System

Figure 5.8 shows the variation in current across three transmission-sleep cycles. The average current load was found to be **36 mA**. With a 10,000 mAh battery, the system has a battery life of around **280 hours** or **12 days**.

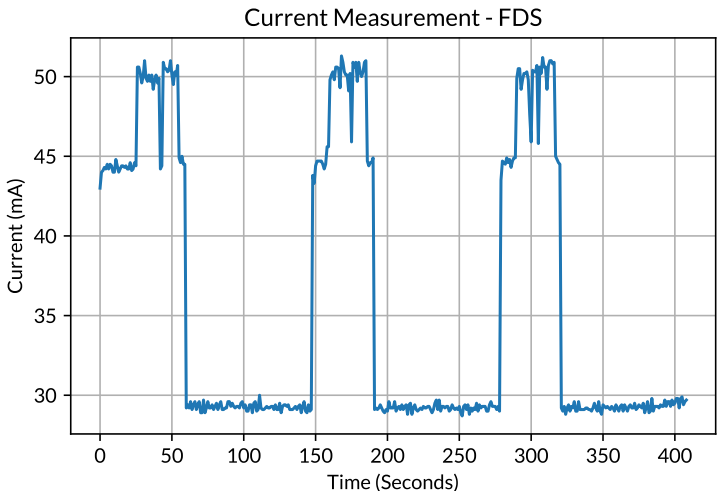

Figure 5.8 Fire Detection System Battery Consumption Plot

5.4 Discussion of Results

Integration testing indicates that not only does each individual component within the system function properly in isolation, but they also seamlessly integrate with each other.

Load testing revealed that the server was able to handle the load efficiently and reliably, without experiencing any performance issues or bottlenecks. The resource utilisation was relatively low and stable, with an average CPU usage of 25.35%, an average memory usage of 6.16%, and an average disk usage of 24.30%. The data also shows that the resource utilisation did not vary significantly with the load, as the standard deviation values were low for memory and disk, indicating a small dispersion from the mean. However, the standard deviation for the CPU usage was higher than the other resources indicating that the CPU usage was more variable and less predictable than the memory and disk usage. Despite this, the CPU usage remained within a reasonable range, with a maximum of 29.40% and a minimum of 24.90%.

The battery consumption analysis shows that the battery consumption of each physical node differs due to the different operating conditions such as sleep time and event-based operation. The Car Park Sensor had the lowest battery consumption, with an average current of 32.10 mA when there was a change in state, and an average current of 12.31 mA when there was no change in state. The Fire Detection System has moderate consumption, with an average current of 36 mA, and would last for about 12 days on a single battery charge. The AQM System has the highest battery consumption out of all three physical nodes, with an average current of 41 mA, which translates to a battery life of about 10 days on a single battery charge.

6 Conclusion

This project successfully implemented a miniature model of a smart city, using three physical nodes that collect data from different domains, such as air quality, parking, and fire detection. The project achieved the original aims and objectives, which were to develop a low-power, reliable, and scalable system that can demonstrate the potential of IoT and smart city applications. The project also identified the best possible wireless communication technology and application layer protocol for this system, which were NB-IoT and CoAP, respectively. The project also devised a data collection pipeline which makes use of existing software platforms to store and visualise the data collected from the physical nodes. The project evaluated the system from different perspectives, such as integration testing, load testing, and battery consumption testing. The results showed that:

- All system components integrate well with each other
- The system was able to handle the data collection and transmission efficiently and reliably, without experiencing any performance issues or bottlenecks
- The physical nodes had different power efficiency and longevity, depending on their data transmission frequency, sleep duration, mode and sensors.

Ultimately, the project demonstrated the feasibility and potential of using IoT technologies for smart city applications.

6.1 Future Work

The project could look into embedding geolocation into the JSON document for identifying the location of a physical node. Additionally, there is potential for expanding the number of physical nodes and their capabilities to monitor and collect data on additional aspects of a smart city, such as traffic flow, waste management, and energy consumption. Exploring the integration of additional technologies, such as AI and ML, could improve data analysis and decision-making. For the physical nodes, moving beyond the development board and creating more integrated solutions by using bespoke designed Printed Circuit Boards (PCBs) could also be considered. The server infrastructure, which is currently hosted on a Raspberry Pi can be upgraded to a more suitable solution for a system implemented on a larger scale. This improvement aims to enhance the robustness and security of the system. Further research should be conducted on improving the power efficiency of the physical nodes as well as exploring alternative communication protocols and wireless technologies could also be beneficial.

References

[1] O. Gassmann, J. Böhm, and M. Palmié, *Smart Cities: Introducing Digital Innovation to Cities*, First edition. Bingley, UK: Emerald Publishing, 2019, ISBN: 1-78769-614-6.

[2] R. M. Flask, "ParkS : Monitoring parking space availability in a multilevel car park," M.S. thesis, University of Malta, 2008. [Online]. Available: https://www.um.edu.mt/library/oar/handle/123456789/74401.

[3] S. J. Johnston *et al.*, "City Scale Particulate Matter Monitoring Using LoRaWAN Based Air Quality IoT Devices," English, *Sensors*, vol. 19, no. 1, Jan. 2019. DOI: 10.3390/s19010209. [Online]. Available: https://www.proquest.com/docview/2232209737/abstract/1FAA7910B5B74840PQ/1.

[4] D. Bajovic *et al.*, "MARVEL: Multimodal Extreme Scale Data Analytics for Smart Cities Environments," eng, Zenodo, Novi Sad, Serbia, Tech. Rep., Oct. 2021. DOI: 10.5281/zenodo.5547216. [Online]. Available: https://zenodo.org/record/5547217.

[5] SparkFun Electronics, *Ultrasonic Ranging Module HC-SR04*. [Online]. Available: https://cdn.sparkfun.com/assets/b/3/0/b/a/DGCH-RED_datasheet.pdf.

[6] Mouser Electronics, *DHT11 Humidity & Temperature Sensor*. [Online]. Available: https://eu.mouser.com/datasheet/2/758/DHT11-Technical-Data-Sheet-Translated-Version-1143054.pdf.

[7] Hanwei Electronics, *MQ-135 Gas Sensor*. [Online]. Available: https://www.electronicoscaldas.com/datasheet/MQ-135_Hanwei.pdf.

[8] Hanwei Electronics, *MQ-4 Gas Sensor*. [Online]. Available: https://www.sparkfun.com/datasheets/Sensors/Biometric/MQ-4.pdf.

[9] Arduino, *MKR NB 1500*. [Online]. Available: https://docs.arduino.cc/hardware/mkr-nb-1500.

[10] J. Ryu, *NB-IoT Handbook*. [Online]. Available: http://www.sharetechnote.com/html/Handbook_LTE_NB_LTE.html.

[11] GSMA, *Narrowband - Internet of Things (NB-IoT)*. [Online]. Available: https://www.gsma.com/iot/narrow-band-internet-of-things-nb-iot/.

[12] *What is LoRaWAN® Specification*. [Online]. Available: https://hz1.37b.myftpupload.com/about-lorawan/.

[13] *Device Classes*. [Online]. Available: https://www.thethingsnetwork.org/docs/lorawan/classes/.

[14] *LoRa and LoRaWAN: Technical overview*. [Online]. Available: https://lora-developers.semtech.com/documentation/tech-papers-and-guides/lora-and-lorawan/.

[15] *Duty Cycle*. [Online]. Available: https://www.thethingsnetwork.org/docs/lorawan/duty-cycle/.

[16] J. Postel, "User Datagram Protocol," Internet Engineering Task Force, Request for Comments RFC 768, Aug. 1980, Num Pages: 3. DOI: 10.17487/RFC0768. [Online]. Available: https://datatracker.ietf.org/doc/rfc768.

[17] J. Postel, "Transmission Control Protocol," Internet Engineering Task Force, Request for Comments RFC 793, Sep. 1981, Num Pages: 91. DOI: 10.17487/RFC0793. [Online]. Available: https://datatracker.ietf.org/doc/rfc793.

[18] Z. Shelby, K. Hartke, and C. Bormann, "The Constrained Application Protocol (CoAP)," Internet Engineering Task Force, Request for Comments RFC 7252, Jun. 2014, Num Pages: 112. DOI: 10.17487/RFC7252. [Online]. Available: https://datatracker.ietf.org/doc/rfc7252.

[19] R. Coppen, E. Briggs, K. Borgendale, and R. Gupta, *MQTT Version 5.0*. [Online]. Available: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html.

[20] InfluxData, *InfluxDB Time Series Platform*, Jan. 2023. [Online]. Available: https://www.influxdata.com/products/influxdb/.

[21] Grafana Labs, *Grafana*. [Online]. Available: https://grafana.com/grafana/.

[22] Docker Inc., *What is a Container?* Nov. 2021. [Online]. Available: https://www.docker.com/resources/what-container/.

[23] S. Hamdani and H. Sbeyti, "A Comparative study of COAP and MQTT communication protocols," in *2019 7th International Symposium on Digital Forensics and Security (ISDFS)*, Jun. 2019. DOI: 10.1109/ISDFS.2019.8757486.

[24] Arduino Libraries, *MKRNB Library for Arduino*, C++, Mar. 2023. [Online]. Available: https://github.com/arduino-libraries/MKRNB.

[25] H. Niisato, *CoAP client, server library for Arduino*. C++, May 2023. [Online]. Available: https://github.com/hirotakaster/CoAP-simple-library.

[26] Arduino Libraries, *ArduinoHttpClient*, C++, May 2023. [Online]. Available: https://github.com/arduino-libraries/ArduinoHttpClient.

[27] B. Blanchon, *ArduinoJson*, C++, May 2023. [Online]. Available: https://github.com/bblanchon/ArduinoJson.

[28] u-blox, “SARA-R4 series AT Commands Manual,” Sep. 2022. [Online]. Available: https://content.u-blox.com/sites/default/files/SARA-R4_ATCommands_UBX-17003787.pdf.

[29] Adafruit Industries, *DHT sensor library*, C++, May 2023. [Online]. Available: https://github.com/adafruit/DHT-sensor-library.

[30] M. A. Califa Urquiza, G. Contreras Contreras, and Y. R. Carrillo Amado, *MQSensorsLib*, C++, Sep. 2019. DOI: 10.5281/zenodo.3384301. [Online]. Available: https://doi.org/10.5281/zenodo.3384301.

[31] C. Amsüss, *Aiocoap – The Python CoAP library*, Python, May 2023. [Online]. Available: https://github.com/chrysn/aiocoap.

[32] InfluxData, *influxdb-client-python*, Python, May 2023. [Online]. Available: https://github.com/influxdata/influxdb-client-python.

[33] Pallets Projects, *Flask*, Python, May 2023. [Online]. Available: https://github.com/pallets/flask.

[34] R. Degges, *python-ipify*, Python, Jun. 2023. [Online]. Available: https://github.com/rdegges/python-ipify.