DEPARTMENT OF INFORMATION AND ELECTRONIC ENGINEERING,

MSC IN WEB INTELLIGENCE

Development of an IoT application for natural disaster rescue scenarios

MASTER'S THESIS

OF

DIMITRIOS KAMPAS

Supervisor: Periklis Chatzimisios
Professor, International Hellenic University

Thessaloniki, July 2021

INTERNATIONAL HELLENIC UNIVERSITY

Department of Information and Electronic Engineering, I.H.U.
MSC IN WEB INTELLIGENCE

An IoT-based application for natural disaster rescue scenarios

MASTER'S THESIS

of

DIMITRIOS KAMPAS

Supervisor: Periklis Chatzimisios
Professor, International Hellenic University

Approved by the three-member selection board at 2/7/2021.

(Signature)	*(Signature)*	*(Signature)*
.....	.....	.....
Periklis Chatzimisios Professor, I.H.U.	Ilioudis Christos Professor, I.H.U.	Ougiaroglou Stefanos Special Teaching Staff, I.H.U.

Thessaloniki, July 2021

(Signature)

.....

Dimitrios Kampas

MSc Advanced Microelectronic Systems Engineering, University of Bristol
BSc Computer Science and Telecommunications Engineering, Technological Educational Institute of Thessaly

Abstract

The Internet of Things refers to an ecosystem of autonomous interconnected devices that are capable of interacting with each other as well as their environment based on different stimulae. Their connectivity capabilities provide us with the power to automate and optimize processes within various sectors, such as healthcare, agriculture and industry. One of the fields that may benefit vastly from these technologies refers to the rescue of individuals following natural disasters, which is the main focus of the current thesis. Its goal is the investigation of the challenges within the field, the analysis of relevant solutions and ultimately the development of an application aimed towards aiding the rescue of individuals after such events. The application was implemented with reliability, high-availability and low-cost constraints and was based upon four architectural tiers. The first tier involves the use of low-power ESP32 microcontrollers via custom firmware for the detection of 802.11 devices and the subsequent transmission of data via LoRa communication. The second tier involves data processing and the use of MQTT protocol to transfer data to the Cloud. The third tier involves storing and processing the information via Cloud-based serverless technologies. The final level refers to the use of RESTful APIs to trigger data processing and collect the results for use by an Internet-facing web application, developed for expedited and reliable locating of trapped individuals.

Keywords:

Internet of Things, Microcontrollers, ESP32, LoRa, MQTT, Publish/subscribe, NOSQL, RESTful API, JavaScript, React

Περίληψη

Το Διαδίκτυο των Πραγμάτων αναφέρεται σε ένα οικοσύστημα αυτόνομων διασυνδεδεμένων συσκευών, που είναι σε θέση να αλληλεπιδρούν μεταξύ τους αλλά με και με το περιβάλλον τους βάσει ενός εύρους ερεθισμάτων. Η διασύνδεση τους με το Διαδίκτυο μας παρέχει τη δυνατότητα να αυτοματοποιούμε και να βελτιστοποιούμε διεργασίες σε τομείς όπως η υγεία, η γεωργία και η βιομηχανία. Ένας από τους τομείς που μπορούν να επωφεληθούν από τις τεχνολογίες αυτές είναι η διάσωση σε σενάρια φυσικών καταστροφών, στον οποίο και επικεντρώνεται η παρούσα διπλωματική εργασία. Κεντρικός στόχος αυτής είναι η διερεύνηση των προκλήσεων στον τομέα, η ανάλυση σχετικών υλοποιήσεων, και τελικά η ανάπτυξη μιας εφαρμογής για τον εντοπισμό αποκλεισμένων ατόμων έπειτα από τέτοια περιστατικά. Η υλοποίηση της έγινε με γνώμονα την δημιουργία ενός συστήματος με απρόσκοπτη λειτουργία ανεξάρτητα από τις περιρρέουσες συνθήκες και στηρίχτηκε σε τέσσερα επίπεδα. Στο πρώτο επίπεδο επιτελέσθηκε ο προγραμματισμός μικροελεγκτών χαμηλής κατανάλωσης τύπου ESP32, για την αναζήτηση συσκευών 802.11 και την μετάδοση των μετρήσεων τους μέσω πρωτοκόλλου LoRa. Στο δεύτερο επίπεδο τα δεδομένα γίνονται προϊόν επεξεργασίας και αποστέλλονται μέσω πρωτοκόλλου MQTT στο υπολογιστικό νέφος. Στο τρίτο επίπεδο η πληροφορία αποθηκεύεται και γίνεται προϊόν επεξεργασίας με τη Serverless τεχνολογιών. Στο τέταρτο επίπεδο, RESTful APIs μεσολαβούν στην προώθηση επεξεργασμένων δεδομένων σε διαδικτυακή πλατφόρμα, η οποία αναπτύχθηκε με γνώμονα την αξιόπιστη και ταχύτερη εύρεση εγκλωβισμένων ατόμων.

Λέξεις Κλειδιά:

Διαδίκτυο των Πραγμάτων, Μικροελεγκτές, ESP32, LoRa, MQTT, Εκδότης/συνδρομητής NoSQL, RESTful API, JavaScript, React

Acknowledgements

I would like to extend my sincere gratitude to my supervisor, Dr. Chatzimisios, for his invaluable tutelage and support throughout the duration of this degree. His supervision and advice was critical towards the completion of this work. My gratitude also extends to Dr. Diamantaras for his inspiration and for the opportunity he and the Faculty members gave me to take part in this endeavor.

Big thanks to my family and friends for their continuous encouragement and understanding during all this time. Very special thanks to my friend and colleague Aris for his guidance and support during our collaboration in Cloud-related projects. His wealth of knowledge and attention to detail have been most impactful to me.

Last but not least, my most heartfelt gratitude goes to my wife, Despoina, for her enduring patience and support during all those difficult moments. This wouldn't have been possible without you.

Table of contents

List of figures

1

Introduction

1.1 IoT in disaster rescue scenarios

Internet of Things (IoT) refers to the latest technological trend that has emerged as a result of the advances in communication technologies and electronics miniaturization. Among others, it allows for a vast range of devices to communicate independently with each other, as well as with a vast range of services using Internet-based technologies. IoT technology is being increasingly applied to a wide range of applications which, as stated in [1], includes healthcare monitoring, disaster management, and vehicular management. The IoT term first appeared in 1999 in a presentation called "That "Internet of Things" Thing" by Kevin Ashton [2]. The field extends in many sectors and allows for interoperability between all and any of its application areas. It encapsulates a vast array of sensor-enabled devices interconnected through Internet-based technologies, which in consequence caters for unprecedented automation capabilities. Indicative of the magnitude of the change that IoT is bringing forth, is the fact that IoT is often referred to as the force behind the fourth industrial revolution. As expected, this opens up vast research opportunities. As further stated in [3], for an IoT system to be deemed successful it should be able to provide unlimited node connectivity, regardless of environment or platform, using low-power, secure and fast communication means to accommodate for reliable and close to real-time communication.

One of the areas that can benefit greatly from the use of IoT-related technologies is their use towards human rescue following natural or manmade disaster events, such as earthquakes, floods, landslides, gas explosions etc. As stated in [4], climate-related disasters have resulted in the loss of 2.5 million lives and damage accounting for 4 trillion dollars over a 30-year period from 1970 to 2000. The fact that there is the capability of deploying a large number of autonomous devices in large areas is particularly beneficial in these cases, since it offers the capability of continuous surveillance and early warning, with the goal of pre-emptive action and effective intervention. Additionally, devices like these can be used in areas that are deemed hazardous or unreachable following some phenomena, which could lead to the avoidance of pointless resource expenditure in irrelevant locations.

Relative to the specific area of disaster recovery, there are numerous fields of active research where the applicability of these technologies is under research. The most prominent being [5]:

- Service-oriented disaster management systems
- Volcanic disaster management
- Flood disaster management
- Forest fire management
- Landslide disaster management
- Earthquake disaster management
- Industrial disaster management
- Urban disaster management
- Terrorist attack management
- Victim localization

Generally speaking, disaster management can be divided into pre-disaster and post-disaster management with regard to the approach followed [1]. The former relates to forecasting and exposure reduction, while the latter relates to orchestrating recovery plans to remedy the disaster's after-effects.

1.2 Thesis motivations and contribution

Given the mentioned facts, it is clear that there is currently no definitive means of reliably predicting an oncoming natural disaster. That said, further approaches should be taken to enhance post-disaster recovery processes so that rescue response times are kept to a minimum since they directly refer to human lives. It is a known fact that the first 72 hours following a catastrophic event are crucial towards saving a person [6], which means that any additional overhead in the rescue process may well be translated to casualties.

This paper proposes a means of autonomously and continuously surveying large, sparsely populated areas with the use of low-power autonomous microcontrollers, long-range communication and cloud-based technologies. The final product of the implementation is an Internet-facing web application targeted towards first responders, which will allow close-to-real time monitoring of trapped individuals and will provide additional metrics in their aid.

Contributions made via this implementation include the use of Wi-Fi-based human detection via unsupervised autonomous sensors, the feasibility of using LoRa for long-range and low-power measurement transmission and the benefits of using Cloud-based services as opposed to on-premise ones.

1.3 Thesis structure

The rest of this thesis is organized in the following manner:

Chapter 2 provides a short analysis of topics relevant to IoT-based disaster recovery, while Chapter 3 provides a brief overview of the major technologies used in the developed platform. In Chapter 4 an analysis is performed on the use case that the platform is intended for along with an architectural overview. Chapter 5 contains an in-depth code analysis of all the parts that the implementation is comprised of. Chapter 6 describes platform testing and results extraction based on a test deployment. Finally, Chapter 7 provides an evaluation of the platform's results and a review on whether the initial objectives were fulfilled or not. Additionally, implementation suggestions are made in the last chapter, to aid in upscaling the application in future iterations of the platform.

2

Related work

In this chapter the latest relevant research is presented, which contributes towards the platform analyzed and implemented in the current thesis.

Oliveira et al. [7], [8] developed the Sherlock platform, which allows for passive human detection via smartphone probe request capturing and 802.11 frequency hopping. An ESP8266 microcontroller is used to store data and relay time-accurate metrics via the additional use of a Real Time Clock (RTC) module. An aggregation between known and unknown devices is ultimately performed to discern particular people of interest along with an estimation of the total number of people present within the monitored space.

In a similar manner, research performed by Rosyidi et al. [6] suggested a system which utilizes Wi-Fi probe requests and GPS positioning to estimate the area in which a trapped individual may be present. Retrieved GPS measurements are converted to the Cartesian coordinate system and distance calculation is performed based on the received signal strength (RSSI) value measured for every smartphone detected in the area. Distance estimation and three different means of pinpointing the center of the resulting circular area are used, with the final goal of displaying it on a map. The process followed on the proposed architecture is depicted in Figure 2.1 below.

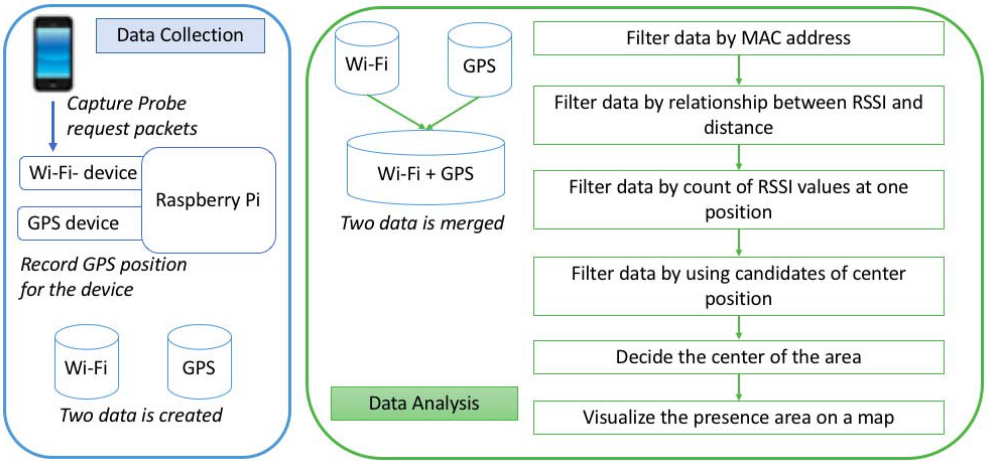

Figure 2.1: Victim localization architecture [6]

Research performed in [9] also suggests tracking 802.11 probe for occupancy monitoring in indoor environments. This is achieved using strategically placed sensors in indoor locations, storing measurements locally and relaying them to a database server at predefined sparse intervals. Subsequently, data processing techniques such as linear least squares and weighted k-nearest-neighbor (WKNN) algorithms are applied to increase location estimation precision. This solution's architectural overview is depicted in Figure 2.2.

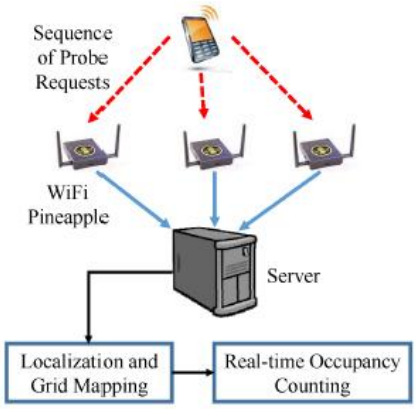

Figure 2.2: Occupancy tracking high-level architecture [9]

Karakaya, Şengül and Gökçay [10] proposed the development of an application for locating victims after the event of an earthquake-driven building collapse. The proposal is aimed towards fulfilling the need of rescuers to prioritize searches in buildings that hold a larger number of individuals compared to others that do not. The architecture as depicted in Figure 2.3 is based on the installation of an application on the smart device that every person within the building possesses. The application then allows for continuous scans of each device's surrounding area, to locate the nearest Wi-Fi Access Point at any given moment. Relevant AP-related information is stored in a database and a battery-operated mini-PC equipped with motion sensors acts as a web server and Wi-Fi hotspot for the devices to connect to in the event of an earthquake.

Babu and Rajan [11] proposed a system to monitor high-risk areas for earthquakes and floods and aid in the rescue process in case those occur. The system utilizes a range of sensors such as vibration and float sensors to trigger alerts towards authorities via GSM communication in the event of emergency. It also utilizes RF communication to trigger light and auditory signals for the affected buildings residents to be notified of imminent danger. An ESP32 module acts as a gateway for relaying information to a cloud-based platform for operational management.

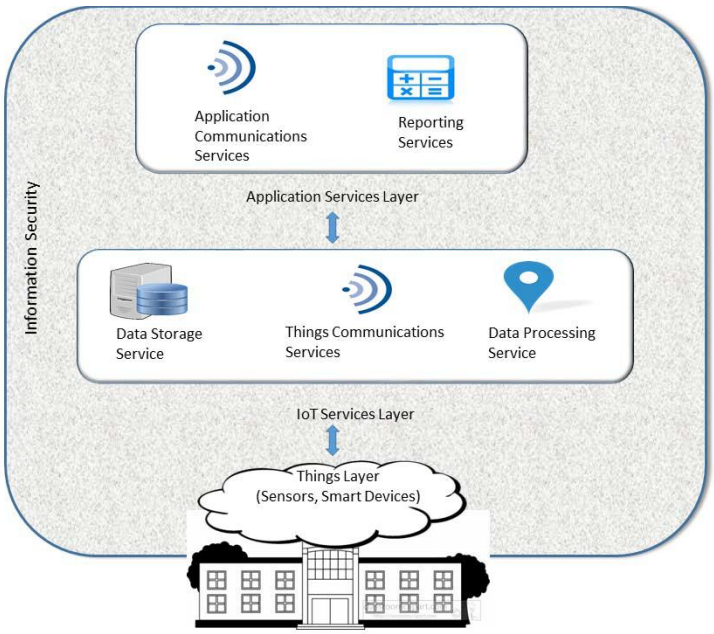

Figure 2.3: Earthquake victim detection platform architecture [10]

Boccadoro, Montaruli and Grieco [12] developed the QuakeSense platform, an open-source monitoring system for earthquakes and severe weather natural phenomena. The platform is based on the use of autonomous ARM-based microcontrollers which are powered via energy harvesting and use a communication paradigm based on LoRa and MQTT to ultimately allow real-time event monitoring via a web-based application. The architectural layout is described in Figure 2.4 below.

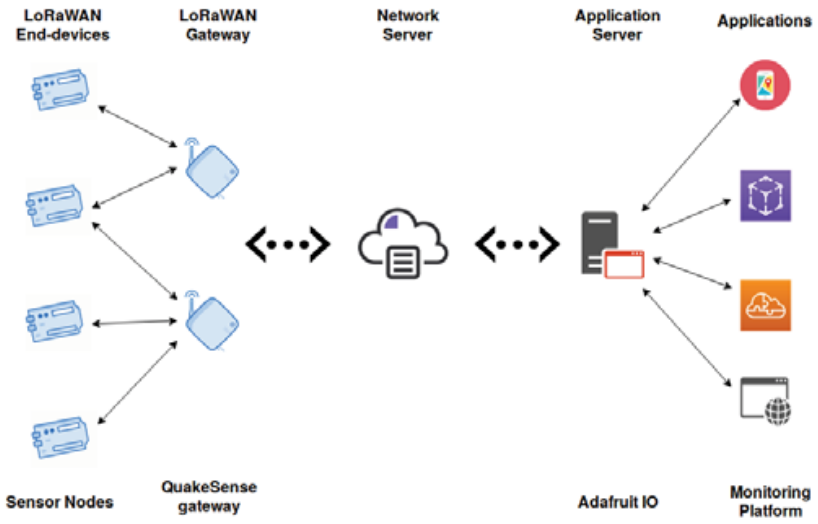

Figure 2.4: QuakeSense platform architecture [12]

An IoT-based security system was developed by Kodali et al. [13], based on the use of ESP8266 microcontrollers' promiscuous mode, which as will be analyzed on a following

chapter, allows for Wi-Fi packet sniffing within the device's range. PIR sensors are connected to the microcontrollers so that each scan is triggered with the mentioned method in case movement is detected within the monitored space. Information is subsequently transmitted via MQTT and stored on a database and any detected MAC addresses not defined as belonging to the house owners, will trigger a notification to the latter's smartphone.

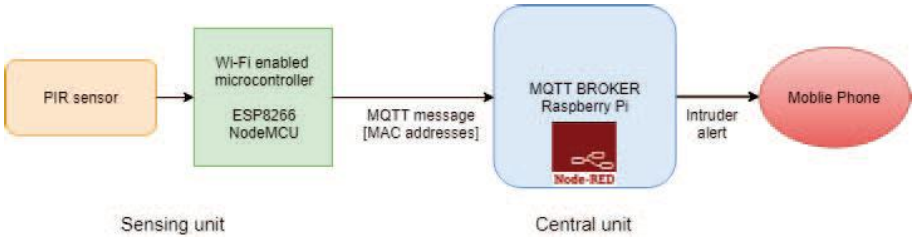

Figure 2.5: IoT-based security system architectural overview [13]

In [14], the use of ESP32 microcontrollers is investigated towards enabling LoRa-based smartphone long range communication in cases where existing communication infrastructure may be unreachable or offline, such as in rural areas or in areas where disaster has struck. A chat application was developed on top of the proposed architecture, to allow fast and simple communication catering for the special circumstances these situations impose. In addition to these, a disruption-tolerant networking paradigm was applied to further enhance the system's robustness. The resulting implementation is stated to provide a "low-cost, low-energy, and infrastructure-less" [14] means of communication.

Amr et al. [15] proposed the use of RSSI values measured between beacons and individual devices, towards estimating the distance between them. A distance correction formula is also performed to improve distance calculation accuracy. The solution is applied on a Bluetooth Low Energy (BLE) beacon-based architecture and results showed that it achieved significant accuracy improvements by up to 80.97%.

Similar to the aforementioned research, Sciullo, Trotta and Di Felice [16] developed the LOCATE platform, which allows multi-hop LoRa-based emergency message transmission, in situations when 3G/4G networks cease to operate following disaster-driven catastrophic failure. The implementation allows for emergency messages sent by disaster victims to be re-broadcasted by any number of peers until they reach first-responder teams. The information transmitted includes geo-location information and the implemented platform provides a means of surpassing communication failures and providing first-responders with fast and accurate data that point to trapped victims.

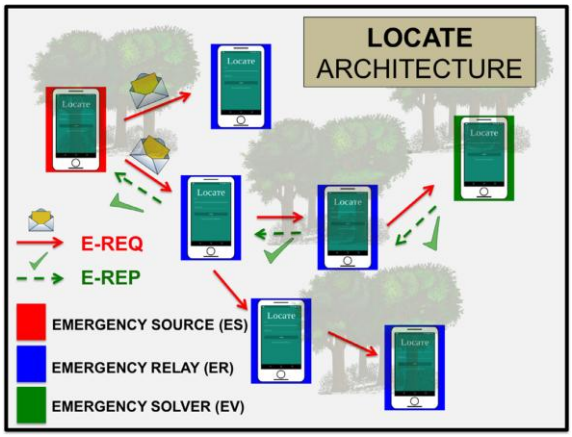

Figure 2.6: LOCATE platform high-level architecture [16]

Kodali and Yerroju [17] proposed the use of ESP32 microcontrollers that encompass flame, smoke and gas sensors as well as a GPS module, for alerting rescue services directly and notifying them of the location that the event has occurred. MQTT is used for reliable message communication towards the targeted crews, as can be seen in Figure 2.7.

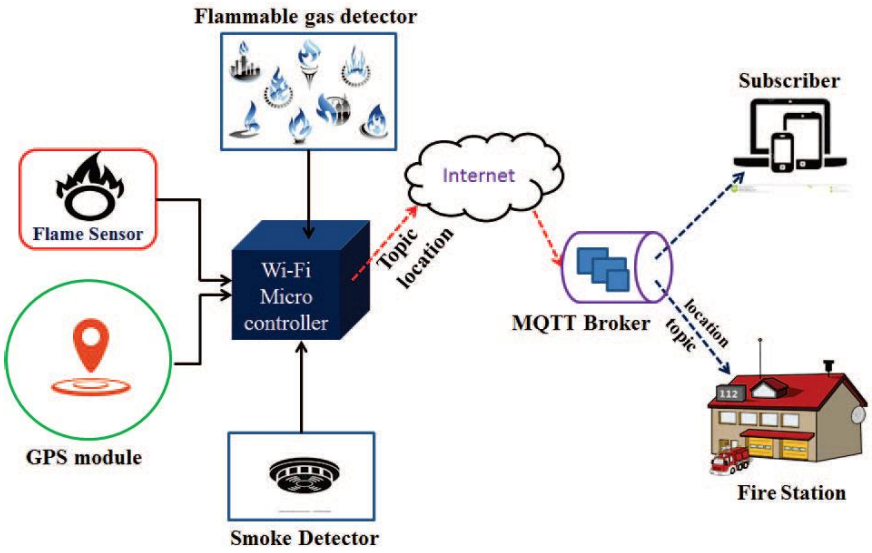

Figure 2.7: Fire hazard emergency response system [17]

3

Technologies

In the current chapter a brief overview is made against the most significant technologies used throughout the platform.

3.1 ESP32 microcontrollers

The emerging trend of low-cost, low-power and feature-rich microcontrollers has led to the development of devices that can handle a vast range of use cases. ESP32 is the successor of the ESP8266 microcontroller by Espressif and encompasses integrated Wi-Fi and Bluetooth/BLE modules. Specific vendor-specific variants also include LoRa, Sigfox and LTE/LTE-M connectivity, as well as a range of onboard peripherals such as OLED screens, and micro SD slots. These microcontrollers have extremely low power consumption specifications, at 3.3V and 0.80mA [17][18][19] and offer the capability of operating in 4 different power-states for maximizing power efficiency. Proper handling of these power states may allow for an extension of battery-based operation from 8.74 to 34.43 days on the same 3000mAh battery without intermittent charging[18]. An ESP32 microcontroller may additionally operate under otherwise hazardous conditions, such as temperatures of -40 to +120 C [17], which makes it ideal for use in extreme scenarios, such as in proximity to an ongoing wildfire.

Performance-wise, an ESP32 microcontroller employs a dual-core Tensilica Xtensa 32-bit 240MHz, as well as an Ultra low power (ULP) co-processor which among others allows ADC conversions[19][18] , a feature that may prove useful for upscaling the platform in future implementations. Commonly marketed ESP32-based modules such as the ones used in this implementation, come at a cost of less than 15 Euros each and offer capabilities for 802.11 b/g/n, Bluetooth v.4.2 Br/EDR and BLE connectivity [20][19], as well as LoRa communication capability which is crucial to the current use case. Firmware development is performed in a form of C++ via the Arduino IDE, which is backed by numerous libraries to support the wide range of ESP32-based variants currently on the market.

A typical physical form and Input/Output (I/O) pin layout of an ESP32-based development board is depicted in Figure 3.1.

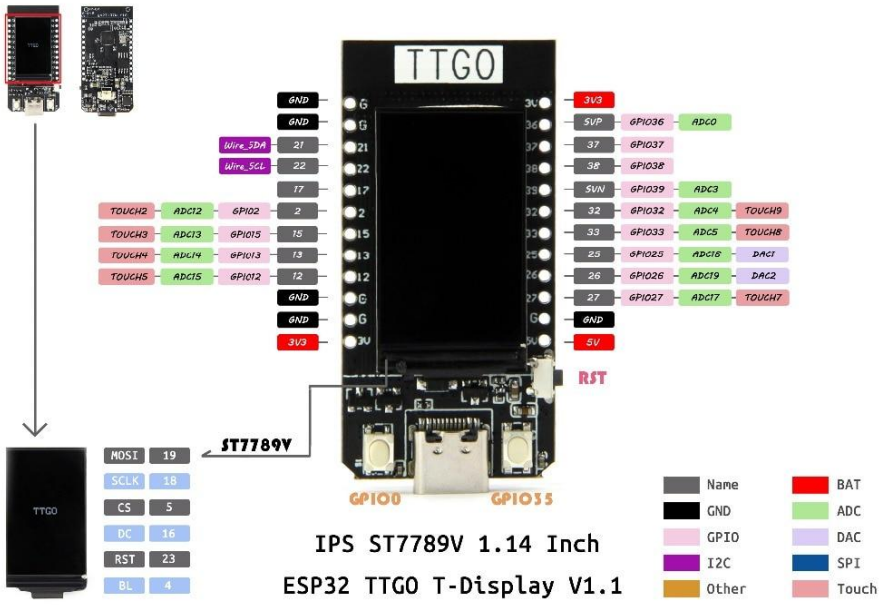

Figure 3.1: I/O Layout of a typical ESP32-based development board [21]

3.2 LoRa

LoRa is an open-source technology that allows long-range and low-power data transmission based on chirp spread spectrum (CSS) modulation with a continuously linear variation [12], [14], [22]. The most significant configuration parameters that define LoRa communication are bandwidth, spread factor and coding rate [12] and determine the way that the signal is spread over the available bandwidth. These parameters contribute towards alleviating effects such Doppler, multipath and fading [12], [22]. A high level description of LoRa configuration parameters is presented in the context of transmission distance in [23]. Large spreading factor figures are required for larger transmission distances, since this will provide increased processing gain and higher reception sensitivity. The trade-off is that these characteristics will impose a lower data rate.

LoRa operates on the unlicensed ISM band in the sub gigahertz frequency, which means that communication is free within these frequencies. The frequencies used differ depending on the region of the world that the technology is aimed to be used, such 433, 868 and 915 MHz [14].

The most important LoRa advantages are [22] :

- Long distance transmission
- Low power

- Easy and cheap adoption by devices
- Gateways with concurrent reception capability
- Resilience to Doppler effect

The characteristics that make the technology favorable towards adoption in IoT applications are the communication range and energy-related needs. LoRa communication has been achieved over 4- to 10 km distances [22], [24] depending on the configuration and environment used in. A Line-Of-Sight (LoS) scenario may support reliable communication on the 9-10km distance range with a Packet Reception Ratio (PRR) of >70% [22], while communication on a Non-Line-Of-Sight (NLoS) scenario may be achieved in the 2km range. Meanwhile, LoRa requires 120 to 150mW for transmission purposes and 10-15mW for microcontroller-related operation, which can be translated to 2-5 years of operation [22].

Additionally, the support of concurrent reception of up to 8 channels [22], makes integration of a large number of devices and scaling of existing platforms very easy and requires no additional overhead.

LoRa's characteristics with regard to bandwidth, range and use are depicted in Figure 3.2, where other wireless communication technologies are also visible for the sake of comparison.

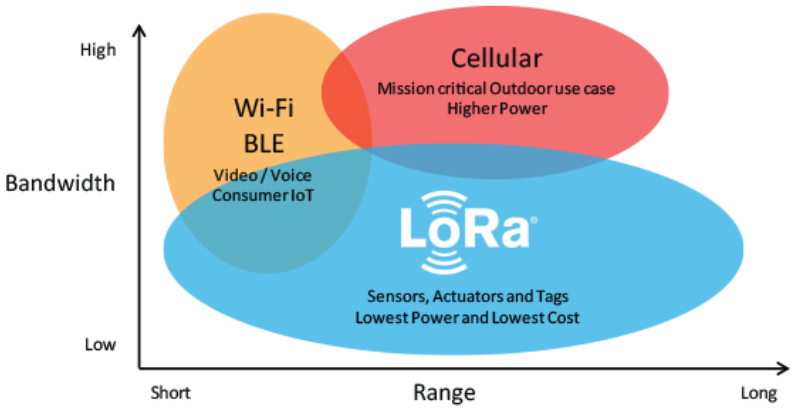

Figure 3.2: LoRa technology capabilities / constraints [25]

3.3 Message Queuing Telemetry Transport protocol (MQTT)

MQTT is a bi-directional publish/subscribe messaging protocol, which ensures reliable communication with minimal network bandwidth and power consumption [17], [26]. It is comprised of three components:

- A message publisher
- A message broker
- A message subscriber

The message publisher and subscriber fall within the greater MQTT Client category, which is responsible for emitting or consuming messages, opening / closing connections to an MQTT server etc. The message broker, otherwise known as an MQTT server, is responsible for managing subscription/unsubscription requests, establishing connections, directing messages depending on who the subscriber is and authenticating/decoupling clients. The latter occurs in three different ways [17]:

- Space decoupling – Publisher and subscriber are not aware of each other
- Time decoupling – Publisher and subscriber are not connected simultaneously
- Synchronization decoupling – Publisher and subscriber are not halted during transmission/reception

A typical use case of MQTT involves a sensor "directing" actions on other devices depending on the readings it transmits. Any subscribed end devices receive the sensor-derived published data and act as per pre-defined directives, if any. Such a use case can be seen in Figure 3.3, where a thermometer transmits temperature readings to be consumed by a web service and a smartphone, both subscribed to the topic where the thermometer-enabled device publishes.

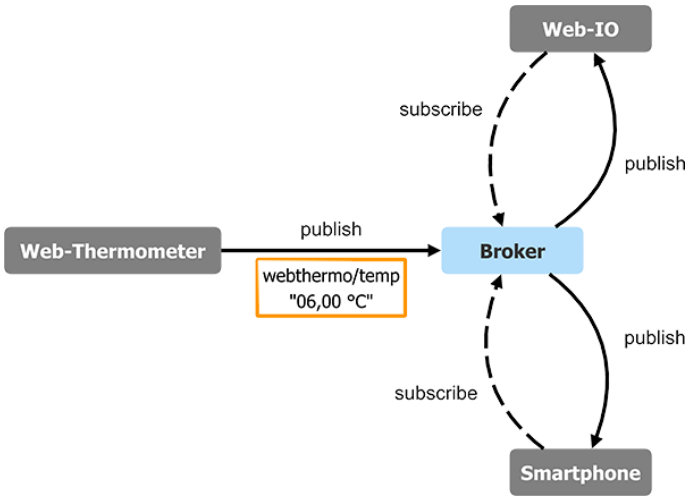

Figure 3.3: Typical MQTT use case [27]

There are three qualities that define message delivery via MQTT [26], [28]:

- QoS 0 – At most once
- QoS 1 – At least once
- QoS 2 – Exactly once

The difference between the three qualities lies within the number of times a message is expected to arrive to its destination. QoS 0 is suitable for applications where the possibility of

message loss is insignificant, while QoS 2 ensures message delivery in mission-critical applications where data redundancy is a perquisite. [26]

MQTT utilizes TCP transport features and requires minimal overhead which is beneficial towards its use by microcontrollers [17]. In addition to that, the scalability factor that the protocol provides ensures that a broadcasted message can be received by any number of devices subscribed to a topic. This makes MQTT a perfect solution for intra-device IoT communication.

3.4 Amazon Web Services (AWS)

Cloud Computing is defined as ”a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction” [29]. The rise of broadband connections in number and in speeds accelerated growth in this area and vendors such as Amazon, Microsoft and Google sought to expand their service offerings towards all cloud computing models; Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS).

IaaS refers to provisioning infrastructure to the client, such as server, storage or networking resources, via virtual machines accessible through the Internet. PaaS is the next level that builds on top of IaaS and refers to providing the clients with control over software deployment, while any relevant infrastructure is managed by the Cloud provider. Finally SaaS, otherwise known as on-demand software, refers to the provision of software where everything in the underlying architecture is managed by the Cloud provider.

AWS is the most widely used Cloud provider currently in the market. It was first established as a cloud provider in 2006 and currently offers over 175 services used by businesses in 240 countries in the world. [30]. AWS is built upon infrastructure located in locations around the world, which are called AWS Regions. Each region may contain more than one Availability Zone (AZ) which is comprised by a number of data centers to ensure redundancy, fault tolerance and scalability. Currently AWS operates across 80 Availability zones in 25 Regions around the globe [30].

Cloud technologies offer various advantages to IoT platforms, with the most significant being the always-on remote access to collected data, as well as the provision of unlimited storage [29], which in the cases of microcontroller-based devices is in some cases extremely limited. In addition to that, subsequent data processing can occur via access to unlimited on-demand processing power. There is a wide range of analytics tools provided by all Cloud services

provider, while lately there is also a significant number of machine learning service offerings, which further bolster the integration of cloud technologies with IoT platforms. Last but not least, security is ensured via continuous security assessments [29], while robustness can be ensured via the proper use of redundancy configurations such as multi-AZ deployments and event-driven alerts.

AWS IoT Core is the latest service provided by AWS towards managing and integrating IoT devices. Each device connected to it is represented by a device shadow which has its own unique identity and retains the device's last known state. Additionally it provides the means for communicating data to and from the Cloud premise [29]. Moreover, a set of SDKs is provided to support integration via C++, Embedded C, Java, Python, JavaScript, etc. Security in transit is handled via TLS 1.2 while AES-256 is used for data at rest.

3.5 React

React is an open-source JavaScript library used for building dynamic websites and reusable UI components which update on the client-side. It was first introduced by Facebook and offers simple programming and fast performance. React implements one-way reactive data flow, makes use of components and promotes their reusability. React uses the Virtual DOM which is faster than the regular DOM and allows for representing all elements and their attribute as a node tree. Components and data patterns improve readability, which helps to maintain larger apps.

4

Analysis

The current chapter provides a thorough analysis of the platform under development. This includes a description of the use case, a breakdown of the specific requirements to be met, as well as the final architectural overview.

4.1 Use case

As mentioned in the introductory chapter, this platform is aimed towards first response teams, specifically the ones that are tasked with locating individuals during or after natural disaster situations. Its use in the current implementation is particularly focused upon cases of rescuing trapped individuals from within collapsed buildings in sparsely populated areas, following recurring or single earthquakes of large magnitude. Nevertheless, it can be also be used on a multitude of similar scenarios such as ongoing blazes, wildfires or floods.

With regard to post-earthquake rescue, one of the latest cases has been what is now known as the Izmir/Samos earthquake, which resulted in the loss of 117 human lives and the endangerment of thousands more. Situations like these include cases of long-term entrapment under the rubble, such as the one of a young individual rescued 91 hours after the actual earthquake event [31]. It is ultimately clear that enhancing the effectiveness and timeliness of response operations is of the utmost importance. This could be achieved via a system that can sense human presence in specific areas and notify emergency responders in a fast, almost real-time and reliable way in order to achieve the required refinement of the search and rescue strategies.

4.2 System requirements

The proposed system needs to be able to operate completely autonomously without the need for existing infrastructure, while being robust enough to maintain operation under hazardous conditions. Meanwhile, all measured data needs to be transmitted safely while the system as a whole needs to provide adequate levels of scalability and agility to accommodate for variable

amounts of measured and processed data. Overall, the need of human intervention needs to stay at a minimum, in order to avoid human-imposed error and reduce operational costs.

These constraints ultimately lead to a specific set of requirements which can be aggregated into 4 distinct tiers:

- Tier 1 – Sensing. The system must be able to passively detect the presence of residents within individual detached buildings and provide relevant sensor data that can subsequently lead to a multitude of appropriate metrics to be processed and made available the rescuers. This must be achieved in an operationally autonomous and energy efficient manner, so that the system's capabilities remain uninterrupted during or after a disaster event.
- Tier 2 – Communication. The readings procured throughout the sensing layer must be communicated over long distances with the minimum need for pre-existing power or telecommunication infrastructure, as these may easily cease to operate in the event of a natural disaster. Meanwhile the protocols used must support the low-power requirements of the sensing layer, while also catering for asynchronous communication and scalability.
- Tier 3 – Data storage and processing. The retrieved data must be subsequently processed with the use of technologies which will ensure:
    - Reliability, scalability and robustness of the system disregarding the volume of received data.
    - The minimization of need for human administration or intervention.
    - The minimization of any upfront implementation costs via the use of a pay-as-you-go paradigm, depending on a per-case basis.
- Tier 4 – Data visualization. The processed data must ultimately be made available to the rescue services, via an online platform accessible by them via as many platforms as possible. The platform will demonstrate almost real-time visualization of the required metrics while ensuring no service downtimes. The platform will ultimately provide:
    - Depiction of detailed readings from each deployed sensor in the observed area.
    - Location pinpointing of each deployed sensor on a map, with the inclusion of the last measurement each one has provided.
    - Estimation of the distance of a specific individual from 3 surrounding sensors.

4.3 Architectural Design

Given the system requirements defined previously, the final architecture is ultimately comprised of a specific set of technologies that lead to the architecture depicted in Figure 4.1. This architecture is analyzed further throughout the next chapter.

The solution supports deployment in rural, sparsely populated areas such as villages, but may also be upscaled in the future to support denser residential ones, as described in another chapter. Following the tier segmentation described earlier, the first tier includes the deployment of low-power ESP32 microcontrollers in every residence within the surveyed area. To that end, custom firmware has been developed to allow these devices to scan all Wi-Fi channels within their range and perform MAC-address detection with regard to any Wi-Fi-enabled devices in their proximity. A logic is followed to deduce whether or not any of the detected devices refer to predefined permanent residents. Secondary metrics such as the RSSI are also gathered, which will later be used for distance calculation and other purposes.

The second tier of the architecture, involves the transmission of the measured data over a long distance towards a centralized receiving point located at a “safe-zone”, for example a town hall, police station or hospital; i.e. one that will offer communication and energy redundancy. This communication process will occur via LoRa, which taking into account reliability constraints should be aimed towards reaching distances of 4-6km [14], [24]. The centralized point of data reception has the role of gathering any LoRa packets sent from multiple residential-deployed sensors, These packets containing per-device metrics will be then relayed via MQTT towards a designated cloud provider as well as any other subscriber deemed necessary in future application upscaling scenarios.

Once the data is received within the cloud region, it is stored in a NOSQL database to be processed when necessary. A set of serverless services, undertake the task of processing the data towards expected metrics. Finally, a set of developed APIs forward the expected data towards a cloud-deployed front-end client that the rescue team will have access to. The latter 2 points make up for the 2 final tiers described in the architectural level.

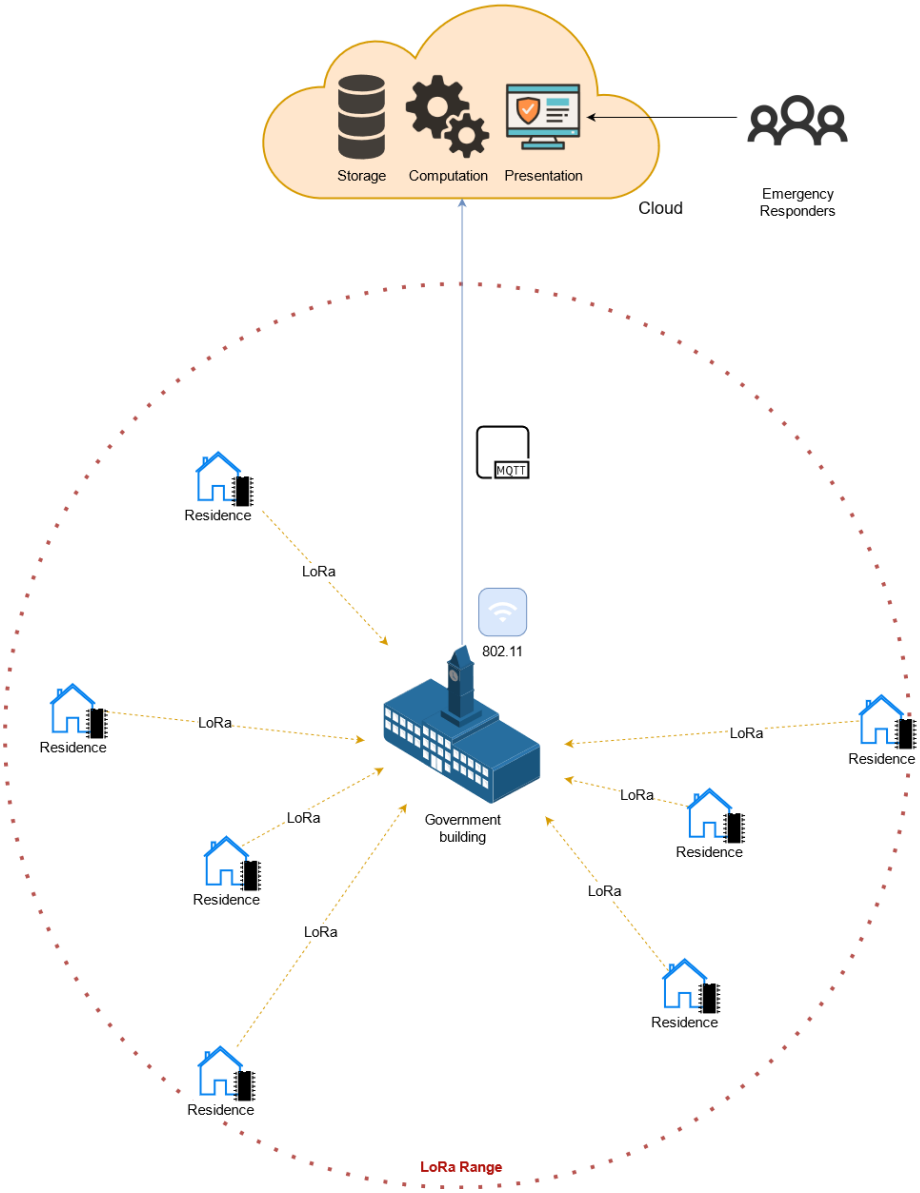

Figure 4.1: High level architectural overview

5

Design & Implementation

In this chapter, each of the architectural tiers is analyzed with regard to the final solution implemented. Each tier analysis is comprised of the specifications and the implementation subsections.

5.1 Tier 1: Sensing

5.1.1 Tier 1 specifications

As mentioned in the previous chapter, the foundational tier of the current application is responsible for gathering all necessary measurements for the platform and is comprised of a set of ESP32 microcontrollers deployed on every residence in a surveyed area. As the defined architecture suggests, the foundation of the platform lies on the installation of a single ESP32 microcontroller module on every detached residence within the monitored area. This subset of residentially-placed ESP32-based modules will henceforth be designated as "Type-A" modules throughout this paper. These modules will be connected to the power grid as well as a battery with a capacity of at least 3000mAh, which will provide continuous operation after a probable catastrophic failure of the power grid. Placement of Type-A modules in residences may occur within a casing so as to prevent damage from collapse, flooding, or human intervention. In this case the construction of the casing should be of a non-metallic nature, since it is a known fact that metal may impose sensor and signal interference[32] which may greatly affect sensor data and communication capabilities. Additionally, any expected mitigations with regard to the RSSI being affected by the casing or other obstacles upon the event of disaster may be alleviated by various means, such as trilateration as analyzed throughout multiple cases as in [33]–[37]. In the current implementation, localization has been rudimentally visited via the calculation of distance with the use of the RSSI values retrieved by Type-A modules.

The functionality for which Type-A modules are indented for, is to primarily detect a number of pre-defined permanent residents present within the building it has been deployed into. This feature is based upon an ESP32 inherent functionality, called promiscuous mode [13], which

enables scanning within the module's built-in Wi-Fi radio range for devices that are polling for a 802.11-type connection. This does not require any other Wi-Fi network infrastructure to be in range or any of the mentioned devices or module to be connected to one; instead, the feature relies on frequency hopping (otherwise known as channel hopping), a feature which has also been used for pedestrian detection and monitoring in [38]. This logic features a switching process throughout 13 channels of 802.11b/g 2.4GHz spectrum, which are spaced 5MHz to each other. This process ultimately allows the detection of every device in range without risking any one being missed due to it polling for a connection within a channel other than a single one the Type-A module might have been targeting. This process may prove to be costly in terms of energy constraints, thus it is imperative that the scanning process occurs for the shortest duration possible, with intermittent pauses present in-between each iteration. In the current solution switching through each channel occurs every 1 second, while a delay of 35 seconds is introduced between each iteration. The information retrieved provides each detected device's MAC address, as well as the RSSI measurement which signifies the signal strength between the sensor and that device.

Once the process of channel-hopping has elapsed across all channels and information has been gathered, measurements are processed so that the device may discern between known and unknown devices. The decision on whether a detected device designates a resident or not, is made on the basis of a pre-defined list of MAC addresses, hardcoded within each Type-A module's firmware. This list is dependent on the building in which the module is installed. In consequence, each building will have a number of known MAC addresses tied to it, which will refer to each of the permanent residents living into it and currently present within. In this implementation the list has been provided directly within each device's firmware and demonstrates an arbitrary one-to-one binding of 1 device to 1 resident, which could as well be referring to the person's smartphone. Future implementations may approach this in different manners, as suggested in a final chapter. The data structure regarding permanent residents includes each device's designation (or otherwise, a nickname) as bound within the aforementioned whitelist, as well as the measured RSSI value. Any additional unlisted devices detected trigger the generation of a separate higher abstraction metric, in order to demonstrate the potential presence of additional people within the structure. The related information due to be transmitted refers to the total number of these unknown devices, as there is currently no means of signifying or quantifying human presence in relation to the number of these unknown devices.

The processed data referring to known and unknown devices is due to be sent via LoRa to the centralized node, and for this purpose is formed in a string JSON-like manner, keeping in mind to meet any LoRa restrictions as described on the upcoming Tier 2 section. Once the

process of frequency hopping, data aggregation and data transmission are finished, the device enters a state of standby. This is targeted towards what was previously specified as the means of reducing battery consumption in case the module has been severed from the power grid. In the current implementation this has been approached via a simple delay timer, however future implementations may make use of the available sleep-modes that ESP32 microcontrollers can operate under, as described in the final chapter.

The process loop that a typical Type-A module operates under, based on the firmware developed for this implementation, includes the following steps in sequence, as depicted in Figure 5.1 :

1. Channel hopping to gather all Wi-Fi-enabled devices within the module's coverage area
2. Aggregation of known and unknown devices based on a predefined whitelist
3. Processing data into an appropriate LoRa packet and data transmission via LoRa towards the centralized node
4. Pause of function for a specified time before the process is re-started from step #1

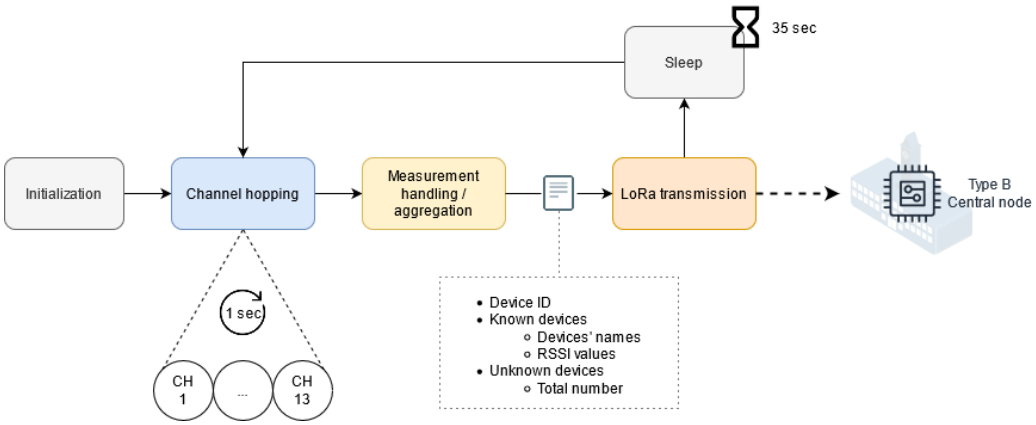

Figure 5.1: Operation of a Type-A module

5.1.2 Tier 1 implementation

This section includes a breakdown of the firmware developed for Type-A modules, along with extra notes on crucial parts of the logic. The firmware is developed in a special form of C++ which is designed to be compiled by Arduino IDE and deployed on a wide range of microcontrollers it supports.

The first part of the code depicted on Code snippet 1, handles all initializations of variables and libraries used throughout the code.

```
#include <WiFi.h>
#include <Wire.h>
#include <SPI.h>
#include <LoRa.h>
#include <Ticker.h>
#include "esp_wifi.h"
#include <Arduino.h>
#include <math.h>
#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>

#define OLED_SDA 4
#define OLED_SCL 15
#define OLED_RST 16
#define SCREEN_WIDTH 128
#define SCREEN_HEIGHT 64
#define maxCh 13
#define SLEEP_DELAY 35
#define MAC_LIST_MAX 64
#define OLED_SDA 4
#define OLED_SCL 15
#define OLED_RST 16
#define SCREEN_WIDTH 128
#define SCREEN_HEIGHT 64
#define SCK 5
#define MISO 19
#define MOSI 27
#define SS 18
#define RST 14
#define DIO0 26
#define BAND 866E6

Adafruit_SSD1306 display(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire, OLED_RST);

int device_ID = 1;
int curChannel = 1;
int channel_iterations = 0;
int rssi = 0;
String maclist[MAC_LIST_MAX][4];
int listcount = 0;
int unknown_devices_LoRa_packet = 0;
String known_devices_LoRa_packet[10][2] = {
{"", ""}, {"", ""}, {"", ""}, {"", ""}, {"", ""}, {"", ""}, {"", ""},
{"", ""}, {"", ""}, {"", ""}
};

// Devices with known MAC addresses
String KnownMac[10][2] = {
  {"Resident 1", "BC98DF8FB9BA"},
  {"Resident 2", "2645EDB9ED9E"},
  {"Resident 3", "D0F88C16AB0E"},
  {"Resident 4", "542758B358DB"},
  {"Resident 5", "BC8CCD84F539"}
};
```

```
String defaultTTL = "60";

const wifi_promiscuous_filter_t filt = {
  .filter_mask = WIFI_PROMIS_FILTER_MASK_MGMT | WIFI_PROMIS_FILTER_MASK
_DATA };

typedef struct {
  uint8_t mac[6];
} __attribute__((packed)) MacAddr;

//Management Frame definition
typedef struct {
  int16_t fctl;                //Frame Control
  int16_t duration;            //Duration
  MacAddr da;                  //DA - Destination Address
  MacAddr sa;                  //SA - Source Address
  MacAddr bssid;               //SSID
  int16_t seqctl;              //Sequence Control
  unsigned char payload[];     //Payload
} __attribute__((packed)) WifiMgmtHdr;
```

Code snippet 1: Type-A firmware initializations/declarations

This part of the code handles all necessary initializations with regard to libraries that include specific functions such as the enablement of promiscuous mode, as well as supporting variables and flags. The libraries used within Type-A module firmware include:

- Standard commonly required libraries such as Arduino.h and math.h which is necessary for compilation by the IDE and standard mathematic functions respectively.
- ESP32-specific libraries such as Wire.h and SPI.h, which are required for setting the internal communication protocol for communication of the microcontroller with any peripherals. For reference, the former library defines synchronous serial bus communication to I2C (Inter-Integrated Circuit) devices such as the OLED display, while the latter defines master-slave protocol for use with the module's LoRa shield.
- WiFi.h and esp_wifi.h for all functions related to the ESP32's Wi-Fi capabilities, such as handling promiscuous mode
- Other supporting libraries, such the Adafruit_GFX.h and the Adafruit_SSD1306.h which are required for setting up and using the OLED display present onboard the module.

Following the library declarations, all necessary OLED specifications, specific to the screen module used on the current implementation, are set. The OLED display will depict information during the module's operation, such as the designation of the polling process or

the activation of sleep/delay mode. This may also prove helpful during the operational phase of the implementation, as it may easily provide useful information for troubleshooting, which would otherwise require a direct connection to the device via a computer.

A number of supporting global variables is set, to be used throughout the code for exchanging and storing values with regard to the current channel under checking, the overall iterations elapsed since the start of the module's operation and the RSSI value captured for each detected device. The TTL is used to discern whether or not a device has ceased to appear within the Type-A module's range. Finally, maxCh variable specifies the maximum channels through which frequency hopping will occur, which is this case is 13.

A 2-dimentional array is initialized with a length relative to the maximum amount of devices to be handled, in this case 64. Each array element contains 4 elements with regard to each detected device:

- MAC address
- Time elapsed since initial detection
- TTL variable, as specified before
- Measured RSSI

Two additional two-dimensional arrays named *KnownMac* and *known_devices_LoRa_packet* are initialized. The former refers to the pre-defined list of residents expected to be possibly present within the building the module is installed in, while the latter refers to the array that will hold any of the former that are actually detected during the scanning process. The length for both arrays in this scenario has been arbitrarily defined to 10 and their structure follows the previously mentioned 2-value pairing of designation name-MAC address.

A set of definitions relative to the LoRa communication process is included, which includes the pins required for communication of the microcontroller with the LoRa transceiver module as well as the definition of the LoRa band to be used, which in this case is 866E6, referring to the frequency used in Europe. There are also 2 related arrays defined, which refer to the packets that are going to be ultimately sent towards the centralized module. The first array will contain the information on any known devices detected, and will hold 2 values per detected device; each device's designation and its RSSI measurement. The second array will hold the total count of any devices not present in the list of residents.

Finally, there is the definition of the Wi-Fi management frame header denoted as *WifiMgmtHdr*, which is used to segment and make handling of any subsequent related information easier. Variable *wifi_promiscuous_filter_t filt* will allow filtering only the specific frame types the process will need.

The next code segment includes the definition of the function that will enable promiscuous mode on the module, as can be seen in Code snippet 2 bellow.

```
void sniffer(void* buf, wifi_promiscuous_pkt_type_t type) {
  String packet;
  String mac;

  int added = 0;

  wifi_promiscuous_pkt_t *p = (wifi_promiscuous_pkt_t*)buf;

  //Packet length including Frame Check sequence
  int len = p->rx_ctrl.sig_len;

  rssi = p->rx_ctrl.rssi;

  //Management frame header
  WifiMgmtHdr *wh = (WifiMgmtHdr*)p->payload;

  len -= sizeof(WifiMgmtHdr);

  if (len < 0) {
    Serial.println("Received 0");
    return;
  }

  int fctl = ntohs(wh->fctl);

  //Read the first couple of bytes of the packet.
  //Read the whole packet replacing the "8+6+1" with p->rx_ctrl.sig_len
  for (int i = 8; i <= 8 + 6 + 1; i++) {
    packet += String(p->payload[i], HEX);
  }

  //Remove the bits from the start and end of the data to get the MAC
  for (int i = 4; i <= 15; i++) {
    mac += packet[i];
  }

  //Set MAC address to all upper case
  mac.toUpperCase();

  //check if the MAC address has been added before
  for (int i = 0; i <= 63; i++) {
    if (mac == maclist[i][0]) {
      maclist[i][1] = defaultTTL;
      if (maclist[i][2] == "OFFLINE") {
        maclist[i][2] = "0";
      }
      maclist[i][3] = rssi;
      added = 1;
    }
```

```
  }

  if (added == 0) {
    maclist[listcount][0] = mac;
    maclist[listcount][1] = defaultTTL;
    maclist[listcount][3] = rssi;

    listcount ++;
    if (listcount >= MAC_LIST_MAX) {
      Serial.println("Too many addresses");
      listcount = 0;
    }
  }
}
```

Code snippet 2: Type-A module Promiscuous mode function

Function *sniffer* is responsible for enabling the Wi-Fi promiscuous mode that ESP32 modules support. A set of variables are defined which will contain the packet length, RSSI value and the management frame header. Further processing of the packet is done in order to reach a common nomenclature for all retrieved MAC addresses, which will be in the form of a 12-digit hexadecimal capitalized value, as can been seen within the list of known devices declared in the first part of the code.

A loop traverses through all captured MAC addresses to check whether or not each detected MAC address in each channel frequency was already kept during the previous iteration. A check handles the status of pre-detected devices that were set to offline status on previous iterations, and resets the timer referring to their presence in the module's range to 0 if they are re-detected, while the RSSI value is refreshed upon each iteration. Another check handles the addition of newly discovered devices and checks whether or not the maximum limit of devices has been met.

The code segment depicted in Code snippet 3, holds the function responsible for setting up the OLED screen onboard the Type-A module.

```
void setupOLED() {

  pinMode(OLED_RST, OUTPUT);
  digitalWrite(OLED_RST, LOW);
  delay(2);
  digitalWrite(OLED_RST, HIGH);

  Wire.begin(OLED_SDA, OLED_SCL);
  if (!display.begin(SSD1306_SWITCHCAPVCC, 0x3c, false, false)) {
    Serial.println(F("SSD1306 allocation failed"));
    for (;;);
  }
```

```
  display.clearDisplay();
  display.setTextColor(WHITE);
  display.setTextSize(1);
  display.setCursor(0, 0);
  display.print("Init mitsakOps32");
  display.display();
  delay(3000);
  display.clearDisplay();
  display.setTextColor(WHITE);
  display.setTextSize(1);
  display.setCursor(0, 0);
  display.print("mitsakOps32 READY!");
}

String displayRefresh(String s) {
  String out = s;
  display.clearDisplay();
  display.setCursor(0, 1);
  display.setTextColor(WHITE);
  display.setTextSize(1);
  display.print(out);
  display.display();
}
```

Code snippet 3: Type-A module OLED display setup

In this section all necessary pins are set to the proper values in order to reset the OLED display and initialize it towards specific specifications, such as text size and the starting point of the cursor to draw any required graphics. For reference, drawing on the OLED screen of an ESP32 module, requires explicit control of the cursor to move to different parts of the screen.

The code segment depicted in Code snippet 4 handles the timer bound to each of the detected MAC addresses.

```
void purge() {
  for (int i = 0; i <= 63; i++) {
    if (!(maclist[i][0] == "")) {
      int ttl = (maclist[i][1].toInt());
      ttl --;
      if (ttl <= 0) {
        maclist[i][2] = "OFFLINE";
        maclist[i][1] = defaultTTL;
      } else {
        maclist[i][1] = String(ttl);
      }
    }
  }
}
```

```
void updatetime() {

  for (int i = 0; i <= 63; i++) {
    if (!(maclist[i][0] == "")) {
      if (maclist[i][2] == "")maclist[i][2] = "0";
      if (!(maclist[i][2] == "OFFLINE")) {
        int timehere = (maclist[i][2].toInt());
        timehere ++;
        maclist[i][2] = String(timehere);
      }
    }
  }
}
```

Code snippet 4: Type-A module per-device TTL/timer handling

The checks implemented by functions *updatetime()* and *purge()*, aid towards understanding the total time each device has been within range of the sensor module and towards designating its status to offline, should the device exceed the specified TTL value.

Moving on to the next part, the code included in Code snippet 5 refers to the function responsible for depicting pieces of information with regard to any detected device, both on the OLED screen and the IDE's serial monitor. The process also handles the aggregation of any known devices captured, as well as the calculation of the total number of unknown ones. These two parts of information will be sent via LoRa once the iteration is finished.

```
void showpeople() {
  String forScreen = "";
  int unknown_devices_initial_count = 0;
  int known_devices_count = 0;
  int unknown_devices_final_count = 0;

  for (int i = 0; i <= 63; i++) {
    String tmp1 = maclist[i][0];
    String online_status = maclist[i][2];

    if (!(tmp1 == "") && (online_status != "OFFLINE")) {

      unknown_devices_initial_count++;

      //Calculate the minutes alive for each device
      long seconds_alive = 0;
      seconds_alive = (maclist[i][2]).toInt();
      long minutes_alive = seconds_alive / 60;

      for (int j = 0; j <= 10; j++) {
        String tmp2 = KnownMac[j][1];

        if (tmp1 == tmp2) {
```

```
          known_devices_LoRa_packet[known_devices_count][0]=KnownMac[j]
[0];
          known_devices_LoRa_packet[known_devices_count][1]=maclist[i][
3];

          Serial.print("[Known device] " + KnownMac[j][0] + ":" + tmp1
+ " : " + minutes_alive + "m -> " + maclist[i][3] + "dB" + "\n");

          known_devices_count++;
        }
      }
    }
  }

  unknown_devices_final_count = unknown_devices_initial_count -
 known_devices_count;

  forScreen += ("Known devices:     " + String(known_devices_count) + "
\n");
  forScreen += ("Unknown devices:   " + String(unknown_devices_final_co
unt) +  + "\n---------------------\n");
  forScreen += ("TOTAL DEVICES:     " + String(unknown_devices_final_co
unt + known_devices_count) +  + "\n\n");

  /*Operation ON time*/
  unsigned long time_on = (millis() / 1000) / 60;

  forScreen += ("DEVICE ON:       " + String(time_on) + "m\n");
  forScreen += ("CHAN ITERATIONS: " + String(channel_iterations));

  displayRefresh(forScreen);

  unknown_devices_LoRa_packet = unknown_devices_final_count;
}
```

Code snippet 6: Type-A module data aggregation / depiction

It is worth noting that the number of unknown devices calculated in this step, is the finalized value included in the subsequent LoRa packet and is sanitized from any devices that have been designated as known.

Code snippet 6 includes the function responsible for shaping the data to a proper format and sending the information via the LoRa module.

```
void loRaSend(int dev_id, int unknown_dev, String known_dev[10][2]) {

  Serial.print("\n\n\n----------------- SENDING LORA PACKET -----------
------\n\n" );

  LoRa.beginPacket();
```

```
  LoRa.print(dev_id);
  LoRa.print(",");
  LoRa.print(unknown_dev);
  LoRa.print(",");
  LoRa.print("[");

  for (int i = 0; i < 10; i++) {
    if (known_devices_LoRa_packet[i][0] != NULL && known_devices_LoRa_p
acket[i][1] != NULL) {

      if ( i != 0 && i !=9 ) {
        LoRa.print(",");
      }

      LoRa.print("{\"dev\":\"");
      LoRa.print(known_devices_LoRa_packet[i][0]);
      LoRa.print("\",\"rssi\":\"");
      LoRa.print(known_devices_LoRa_packet[i][1]);
      LoRa.print("\"}");
    }
  }

  LoRa.print("]");
  LoRa.endPacket();

  Serial.print("\n\n------------------ LORA PACKET SENT ---------------
---\n\n\n" );
}
```

Code snippet 6: Type-A module LoRa transmission

Function *loraSend* receives 3 values as input; the ID of the Type-A device that is sending the packet, the information regarding unknown devices as was calculated on the previous step, and the information processed for known devices, which will be described further within the current code. The packet is formatted in a JSON-like syntax to accommodate for:

- Easy JSON formatting by the centralized node on the next tier
- Reduction of the amount of characters used within the packet to a minimal, due to LoRa packet size restrictions which will be discussed on the next section.

Function *sleep_delay* is responsible for setting the pause/delay between each frequency hopping iteration. The relevant code is depicted in Code snippet 7.

```
void sleep_delay(int delay_value) {

  digitalWrite(LED_BUILTIN, LOW);

  for (int i = 0; i <= SLEEP_DELAY ; i++) {
```

```
    Serial.print(".");
    delay(1000);
  }
}
```

Code snippet 7: Type-A module intra-iteration delay

Function *sleep_delay* receives the globally defined *SLEEP_DELAY* defined at the first batch of definitions within the current firmware. As mentioned, this values is set to 35 sec for this implementation.

It is worth noting that firmware targeted towards an ESP-family microcontroller contains a *setup()* and *loop()* sections, which make for the initialization phase of the entire platform and the runtime process that will be subsequently be running. These parts of the code are included in Code snippet 7.

```
void setup() {

  pinMode(LED_BUILTIN, OUTPUT);

  Serial.begin(115200);
  Serial.setDebugOutput(true);
  pinMode(0, INPUT_PULLUP);

  wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT();
  esp_wifi_init(&cfg);
  esp_wifi_set_storage(WIFI_STORAGE_RAM);
  esp_wifi_set_mode(WIFI_MODE_NULL);
  esp_wifi_start();

  esp_wifi_set_promiscuous(true);
  esp_wifi_set_promiscuous_filter(&filt);
  esp_wifi_set_promiscuous_rx_cb(&sniffer);
  esp_wifi_set_channel(curChannel, WIFI_SECOND_CHAN_NONE);

  SPI.begin(SCK, MISO, MOSI, SS);
  LoRa.setPins(SS, RST, DIO0);

  if (!LoRa.begin(BAND)) {
    Serial.println("LoRa initilization FAILED!");
    while (1);
  }
  Serial.println("LoRa Initialisation OK!");

  delay(1000);
  setupOLED();
}

void loop() {
```

```
  digitalWrite(LED_BUILTIN, HIGH);

  if (curChannel > maxCh) {
    curChannel = 1;
    channel_iterations++;

    loRaSend(device_ID, unknown_devices_LoRa_packet, known_devices_LoRa
_packet);

    for (int i = 0; i < 10; i++) {
     known_devices_LoRa_packet[i][0] = "";
     known_devices_LoRa_packet[i][1] = "";
    }
      sleep_delay(SLEEP_DELAY);
  }

  esp_wifi_set_channel(curChannel, WIFI_SECOND_CHAN_NONE);
  delay(1000);
  updatetime();
  purge();
  showpeople();
  curChannel++;
}
```

Code snippet 8: Type-A module system initialization and runtime

In this section all previously defined components are going through a first-time setup phase using the previously defined constraints. The functionalities going through initialization/setup are:

- The IDE's serial monitor for real-time tracking of the Type-A module's operation
- The system's Wi-Fi module
- ESP32 Promiscuous mode
- LoRa module
- OLED display

Following the setup phase, the loop function commences with the core functionality that the module will perform iteratively until it is shut off or runs out of power. The operation that the module will perform as dictated by this code can be described at high level of abstraction with the following sequential steps:

1. Perform channel hopping operation and MAC address scanning. Flash the onboard LED while the process is under execution.
2. Update the active state per MAC address detected and refresh relative TTL information

3. Aggregate the detected MAC addresses into known and unknown devices. Form the list of known devices with the set of required values and calculate the total of unknown ones. Display all relevant information

4. Once all 13 Wi-Fi channels are traversed:

    i. Prepare the LoRa packet into the expected structure and send it to the centralized node

    ii. Purge the information regarding any detected devices

    iii. Sleep/delay for the specified 35 second duration

5.2 Tier 2: Communication

The second level of the architecture describes the transmission of all measured data from the previous layer's microcontrollers towards the centralized module. This tier involves the use of LoRa communication to reach a single additional ESP32 microcontroller, which is tasked with preparing and sending the data to the Cloud premises.

5.2.1 Tier 2: Specifications

Following the process of identifying devices within every residence and aggregating the measured data, the architecture dictates that the information will be subsequently transmitted over a long distance to a centrally deployed node. This node is responsible for gathering all transmitted packets from any given number of Type-A modules and relaying them towards the Cloud domain and any other data subscriber that might be deemed useful in future variations of the application. The process involves the use of LoRa communication, MQTT protocol and the deployment of an additional ESP32 microcontroller in a specific centralized location.

As mentioned in the application's use case and architectural overview sections, data transmission from any deployed Type-A microcontroller should be made possible over a long distance while ensuring low-power requirements in order to support their autonomous operation even when operating on battery power. One of the technologies that ensures both requirements is LoRa, which can cater for reliable data transmission within a range of 4-6km [2][3] with very low power requirements, which in the case of ESP32 microcontrollers is measured to 393mW, 689mW and 57mW for receiving, sending and deep-sleep power figures respectively. This can be translated to more than 160 hours of operation on a 20.000mAh battery [14]. The LoRa antennas on both the transmitting and receiving ends should be placed at a relatively high point to avoid any obstructions and achieve the maximum possible

coverage, as signified in [39], where it is proven that placing the transceivers at a 1.5m length may provide at least 4km range in a clear line-of-sight landscape.

The node responsible for capturing the transmitted LoRa packets is an individual EPS32 microcontroller setup via a completely different firmware for the sole purpose of receiving LoRa packets, pre-processing the data and forwarding it towards the Cloud via MQTT protocol. This type of microcontroller will be designated as Type-B microcontroller throughout this paper. This node is different from the mentioned Type-A modules in that it is the only part of the platform that relies to existing infrastructure to a certain degree; a power supply and a Wi-Fi connection. The reason why this was deemed necessary was to eliminate the possibility of single-point-failure, since this is the point that all data passes through on its way towards their final destination. That said, any disruptions in energy or communication provision, would impose a major threat to the stability and efficiency of the whole platform. Various means of alleviating this possibility are suggested in the final chapter; in the current implementation it has been suggested that the centralized node will be placed in a safe government or public building, one that will ensure reliable energy and Wi-Fi connectivity in all but the most extreme disaster scenarios.

The LoRa packets arrive at the centralized node formatted in a JSON-like structure due to the constraints described in the previous subsection. This situation is acceptable within a peer-to-peer-based internal communication level, though should be normalized before transmission towards the open world takes place, in order to accommodate for future-proof universal data integration by any number of receiving ends. That said, the received information is first transformed into a key-value pair JSON format, while a timestamp is also generated as soon as the packet arrives to be also included within the JSON object. This timestamp signifies the time that the measurement of each Type-A module arrived at the centralized node. This is necessary to provide useful information to the rescue teams with regard to the time that the latest visible measurement took place, or any other ones preceding it.

Once the data is processed, it is published via the Wi-Fi network that the Type-B module has connected to, while MQTT protocol is used to publish the data into a topic that has been configured on par with the Cloud provider as described on the next tier's subsection. MQTT has been chosen for this application with regard to the ease of integrating variable numbers of devices on demand [26], in order to ensure that possible failures within the proposed architecture would not impose data discontinuity. Additionally, its lightweight nature [26] makes it possible for additional heterogeneous devices [40] to connect in future implementations, no matter what specifications they may be sharing, thus catering for integration with any number of additional IoT devices that may provide further automation.

Figure 5.2 depicts the functional logic that Type-B modules are programmed to follow.

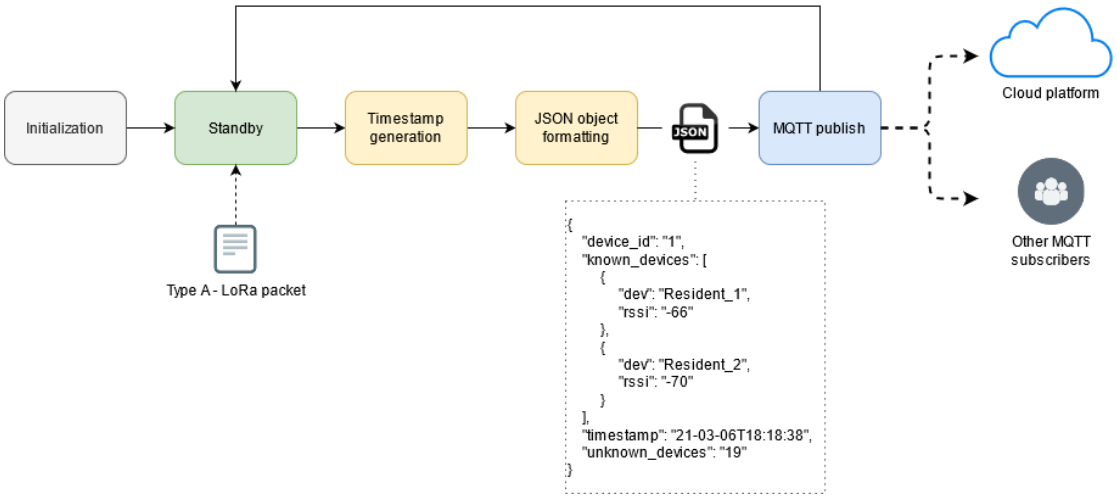

Figure 5.2: Operation of a Type-B module

5.2.2 Tier 2: Implementation

This section contains a thorough analysis of the firmware that controls the centralized Type-B and sets the specified functionalities.

The first part of the code, as depicted in Code snippet 9, contains all necessary initializations with regard to the subsequent operations due to take place.

```
// WiFi connectivity
char ssid[] = "MotoVision";
char password[] = "*************";

// AWS MQTT CREDENTIALS
char HOST_ADDRESS[] = "*************-***.iot.eu-central-
1.amazonaws.com";
char CLIENT_ID[] = "client_1";
char TOPIC_NAME[] = "$aws/things/mt-ops-****/shadow/update";

AWS_IOT m********32GW;

#include <SPI.h>
#include <LoRa.h>
#include <AWS_IOT.h>
#include <WiFi.h>
#include <Wire.h>
#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>

#define OLED_SDA 4
#define OLED_SCL 15
#define OLED_RST 16
```

```
#define SCREEN_WIDTH 128
#define SCREEN_HEIGHT 64
Adafruit_SSD1306 display(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire, OLED_RST);

// Definitions for LoRa transceiver
#define SCK 5
#define MISO 19
#define MOSI 27
#define SS 18
#define RST 14
#define DIO0 26
#define BAND 866E6

// Definitions for time acquisition
#include "time.h"
const char* ntpServer = "europe.pool.ntp.org";
const long  gmtOffset_sec = 7200;
const int   daylightOffset_sec = 3600;
struct tm timeinfo;

String LoRaData;
int count = 0;

int status = WL_IDLE_STATUS;
int msgCount = 0;
int msgReceived = 0;

char payload[512];
char rcvdPayload[512];
```

Code snippet 9: Type-B firmware initializations/declarations

This first block of code contains all necessary library definitions to enable functionality and interoperability of the various integrated components such as the Wi-Fi module and OLED display, in the same manner as described for Type-A module's. Following the same logic, specifications are set with regard to the LoRa shield, which are set among others to the same band as the one in Type-A's firmware code and the OLED display, since the ESP32 module used is identical to the ones used within Tier 1. The information that precedes the mentioned declarations holds the necessary ssid and password to enable the module to connect to the Wi-Fi network that the building it is situated in provides. Variable *status* is later used to discern whether or not a connection has been established successfully.

The next 3 lines contain all MQTT channel information on par with the relevant settings specified within the Cloud platform MQTT server. This information includes the public MQTT host name as specified within the AWS Cloud domain, the name that the Type-B module will be designated by when publishing data, as well as the topic name it is subscribed and will publish to. The host name refers to this specific device's shadow, as will be further

described in the next tier's subsection. In addition to these information, the *AWS_IOT.h* is invoked which is used to initialize, authenticate and pair the device with the AWS Cloud provider.

Following the mentioned initializations is the part where the *time.h* library is declared, along with region-specific localization settings and the server via which the time will be retrieved. This time-related information will be used in conjunction with the declared *timeinfo* variable, in order for the timestamp to be added in the JSON object to be published.

Finally, two variables are specified for the MQTT payloads; *payload* refers to the data to be sent via MQTT, while *rcvdPayload* refers to any data that will be received via the same channel. With regard to the latter, it is worth noting that the Type-B module has been programmed to also subscribe to the MQTT channel, in order to help verify that publishing of each of its messages has been successful. This may also prove useful in future implementations, where bi-directional communication with the centralized node is required.

The Code snippet 10 that follows contains the core functions that set the operation of the module.

```
void printLocalTime()
{
  if (!getLocalTime(&timeinfo)) {
    Serial.println("Failed to obtain time");
    return;
  }
  Serial.println(&timeinfo, "%A, %B %d %Y %H:%M:%S");
}

void MQTTSend(String LoraData, String ts) {

  const char* converted_timestamp = ts.c_str();

  int commaIndex = LoRaData.indexOf(',');
  int secondCommaIndex = LoRaData.indexOf(',', commaIndex + 1);

  String firstValue = LoRaData.substring(0, commaIndex);
  String secondValue = LoRaData.substring(commaIndex + 1, secondCommaIn
dex);
  String thirdValue = LoRaData.substring(secondCommaIndex + 1);

  int id = firstValue.toInt();
  int unk = secondValue.toInt();
  const char *kn = thirdValue.c_str();

  //Convert data to JSON and publish
  sprintf(payload, "{\"id\":\"%d\",\"unk\":\"%d\",\"kn\":%s,\"ts\":\"%s
\",\"cnt\":\"%d\"}", id, unk, kn, converted_timestamp, count);
```

```
  if (mitsakOps32GW.publish(TOPIC_NAME, payload) == 0)
  {
    Serial.print("Publish Message:");
    Serial.println(payload);
  }
  else
  {
    Serial.println("Publish failed");
  }
}

void mySubCallBackHandler (char *topicName, int payloadLen, char *payLo
ad)
{
  strncpy(rcvdPayload, payLoad, payloadLen);
  rcvdPayload[payloadLen] = 0;
  msgReceived = 1;
}
```

Code snippet 10: Type-B module core functions

This section of the code starts with the *printLocalTime()* function which is used to get the timestamp on the arrival of the LoRa packet or debug possible issues while attempting to connect to the previously declared time server. Function *getLocalTime()*is used to transmit a request packet of time structure to the specified NTP server and parse the received time stamp packet into to a readable format. This function is of great importance to the application use case, as the timestamp will be included within the JSON payload to be sent to the Cloud; failure to provide it would deteriorate the efficiency of the platform, as the rescue teams would not have vital chronological information to base their decisions upon.

The second function specified is the one that handles everything related to the MQTT protocol. Its input variables are the data received via LoRa from Type A modules in their original unformatted form, as well as a string representing the timestamp generated via the previously mentioned time-related function. The data procured via LoRa is subsequently split based on the comma-separated form they have been received with, and each individual part is assigned towards an individual variable. The *sprintf* command specifies and sends the final JSON-formatted object via MQTT, where each of the previously split variables set the value in each of the key-value pairs the object contains. A check is performed on whether or not publishing was successful, as well as a second verification of the published data via subscribing the client to the same MQTT topic it has published upon.

The next part of the code includes the setup and initialization phase of the Type-B module, before its core functionality commences.

```
void setup() {

  pinMode(LED_BUILTIN, OUTPUT);
  Serial.begin(115200);
  Serial.println("LoRa RCVR");
  pinMode(OLED_RST, OUTPUT);
  digitalWrite(OLED_RST, LOW);
  delay(20);
  digitalWrite(OLED_RST, HIGH);
  Wire.begin(OLED_SDA, OLED_SCL);

  if (!display.begin(SSD1306_SWITCHCAPVCC, 0x3c, false, false)) {
    Serial.println(F("SSD1306 allocation failed"));
    for (;;);
  }

  display.clearDisplay();
  display.setTextColor(WHITE);
  display.setTextSize(1);
  display.setCursor(0, 0);
  display.print("- LoRa RCVR -");
  display.display();

  //SPI LoRa pins
  SPI.begin(SCK, MISO, MOSI, SS);
  //Setup LoRa transceiver module
  LoRa.setPins(SS, RST, DIO0);

  if (!LoRa.begin(BAND)) {
    Serial.println("LoRa init - FAILED!");
    while (1);
  }
  display.setCursor(0, 20);
  display.println("LoRa init - OK!");
  display.display();

  //Connect to WiFi
  while (status != WL_CONNECTED)
  {
    Serial.print("Attempting to connect to SSID: ");
    Serial.println(ssid);
    status = WiFi.begin(ssid, password);

    delay(5000);   // wait 5 seconds for connection:
  }

  display.setCursor(0, 40);
  display.println("WiFi init - OK!");
  display.display();

  //Get the local time
  configTime(gmtOffset_sec, daylightOffset_sec, ntpServer);
  printLocalTime();
```

```
  //Connect to AWS
  if (mitsakOps32GW.connect(HOST_ADDRESS, CLIENT_ID) == 0)
  {
    Serial.println("Connected to AWS");
    delay(1000);

    if (0 == mitsakOps32GW.subscribe(TOPIC_NAME, mySubCallBackHandler))
    {
      Serial.println("Subscribe Successfull");
    }
    else
    {
      Serial.println("Subscribe Failed, Check the Thing Name and Certif
icates");
      while (1);
    }
  }
  else
  {
    Serial.println("AWS connection failed, Check the HOST Address");
    while (1);
  }

  delay(2000);
}
```

Code snippet 11: Type-B module setup

The code block in Code snippet 11, starts with all necessary OLED display initializations for any relevant information to be displayed while the module operates. Following this, is the initialization of the communication pins with the LoRa shield and also the function call to connect to the Wi-Fi channel specified earlier via the previously specified ssid and password variables. Once the module has successfully connected, the current time is retrieved from the designated time server based on the defined localization settings and a connection with the Cloud platform is attempted based on a set of settings specified in the latter.

The Type-B module firmware ends with the loop section, which defines its runtime as is shown in Code Snippet 12.

```
void loop() {

  digitalWrite(LED_BUILTIN, LOW);

  char timeString[50] = "";
  int publish_flag = 0;
  int rssi = 0;
  int packetSize = LoRa.parsePacket();  //Parse received packet

  if (packetSize) {
```

```
    publish_flag = 1;

    Serial.print("\n\n>>> Received packet: [ ");

    while (LoRa.available()) {
      LoRaData = LoRa.readString();
      Serial.print(LoRaData);
    }

    Serial.print(" ]");

    printLocalTime();

    strftime(timeString, sizeof(timeString), "%y-%m-
%dT%H:%M:%S", &timeinfo);

    display.clearDisplay();
    display.setCursor(0, 0);
    display.print("Last packet received");
    display.setCursor(0, 10);
    display.print("---------------------");
    display.setCursor(0, 40);
    display.print("ts:");
    display.setCursor(25, 40);
    display.print(timeString);
    display.display();
  }

  //Send to AWS
  if (msgReceived == 1)
  {
    msgReceived = 0;
    Serial.print("Received Message:");
    Serial.println(rcvdPayload);
  }
  if (publish_flag == 1)    {
    digitalWrite(LED_BUILTIN, HIGH);

    // Format a JSON response and send via MQTT
    MQTTSend(LoRaData, timeString);

    delay(500);
    digitalWrite(LED_BUILTIN, LOW);
  }
}
```

Code snippet 12: Type-B module runtime

During runtime, the module will await for any LoRa packets to be sent to it, which is handled by the *parsePacket* function. Once one is received, information is displayed on the OLED screen, data contained within the packet is read via calling the *readString()* function and a timestamp is generated and properly formatted for later use. Function *MQTTSend()* is called

to format the mentioned data into a JSON object and publish it to the topic. If publishing is successful, the module will receive the published message, since it is also a subscriber within the same topic. During the duration of sending the data, an onboard LED while remain on and will switch off as soon as the publish ends.

5.3 Tier 3: Data storage and processing

The third tier of the application involves the process of capturing, storing and processing any data transmitted from the centralized node into the Cloud domain. This is an essential part towards providing the final product of information towards the front-end client facing the first responders and anyone involved in the post-disaster rescue operations.

5.3.1 Tier 3: Specifications

The segmented Tier 3 of the application is based upon the utilization of a Cloud-based architecture, which was chosen to cater for scalability, security, mobility and robustness. At a higher abstraction level it can be said that since IoT devices are designed with low-power and mobility constraints, there is a need for other means of provisioning high storage capacity and high computational power and security [26]. Cloud services can provide these features as we'll as additional ones, such the integration of as machine learning techniques towards more intricate results in future implementations.

The aspect of scalability refers to the possibility to scale the application seamlessly such as the addition of extra Type-A or Type-B nodes in the field, without any additional developmental overhead or application downtime. With regard to the mentioned security aspect, any data stored within the cloud is implicitly safe from unauthorized access, which is bolstered through the application of specific architectural paradigms when applied at the communication level, such as using access roles and secret access keys. Mobility of cloud-centric data refers to the fact that all data can be made available agnostically on any platform that has Internet access, which means that the platform can be accessed by any number of authorized websites or applications. Finally, robustness refers to the fact that any data kept in the Cloud is safe against accidental loss or infrastructural failure, due to the use of characteristics such as availability zones. This is very important with respect to the current use case, as any data procured and stored in the Cloud refer to the protection of human life, thus should be safeguarded with any means necessary.

In this implementation, the Cloud provider of choice is Amazon Web Services (AWS), which at the time of this implementation's development is the most widespread on a global scale

[29]. Many services have been used for the purpose of data storage and processing, among which the most significant are:

- AWS IoT Core
- Amazon DynamoDB
- AWS Lambda
- Amazon API Gateway
- AWS Identity and Access Management (IAM)

The data flow within the AWS domain starts from the AWS IoT Core service. The technology review performed in [29] states that "each device connected to IoT Core is represented as a Device Shadow". This practically means that each IoT device that exchanges information with AWS has a unique identity and state, which ensures a number of capabilities such as ensured message delivery [29]. The service enables easy integration with a vast array of services within the AWS domain, which is necessary in order to process and forward the data towards the final front-end client. In the current implementation the centralized node has been configured with relation to the AWS IoT Core service to enable for published MQTT data to be retrieved at the Cloud.

The service allows for querying the received JSON object for any information needed that are present within, in order to subsequently direct the segmented values into a database. The database used in this implementation is Amazon DynamoDB, which is a NOSQL database chosen for its availability, scalability and security characteristics, while ensuring no downtimes no matter what the data traffic is [41]. The service supports out-of-the-box JSON object handling via the provided SDK, which makes handling of the received Type-B-transmitted JSON formatted data a lot easier. Another important feature is the fact that its non-sequential nature caters for any data missing from a received JSON object; any objects that are missing information will still be stored within the database without imposing any risks with regard to the platform's functionality. Ultimately, every data packet is stored in a structured format within the DynamoDB table, with each being placed within an individual item with a set of attributes as specified by the query performed at the IoT Core service level. Once data is stored and segmented to individual attributes within the mentioned database, they are available to be invoked or processed at will.

Since the information provided by the centralized Type-B is not in the expected form, it will need to undergo specific processing before being directed to the developed front-end client. For this purpose a set of serverless AWS Lambda functions are used, to provide on-demand data processing without the need of provisioning a server or other resources. This is important towards the pursued use case, since it fulfills the requirement of a fully autonomous platform for cost and reliability constraints to be met. Additionally, Lambdas scale automatically just

as the mentioned DynamoDB database, thus supports any number of invocations and subsequently any number of concurrent users in the platform. The Lambda functions are triggered via a set of RESTful APIs developed specifically for this implementation via the Amazon API Gateway service. Each of these APIs is tied to a specific view within the front-end client and points to an individual Lambda function, each performing a different kind of data processing.

Supporting the mentioned main services, the security aspect of the application is addressed via AWS IAM, to manage access and security permissions on any services or resources used depending on the functionality they are intended towards. Since the standard of least privilege is followed to ensure the highest security possible, these permissions are fine-grained towards each service individually. Additionally, Amazon Cloudwatch provides monitoring on all the mentioned resources and the data flow itself if required within the cloud. Figure 5.3 depicts the data flow throughout the various AWS services used in this application.

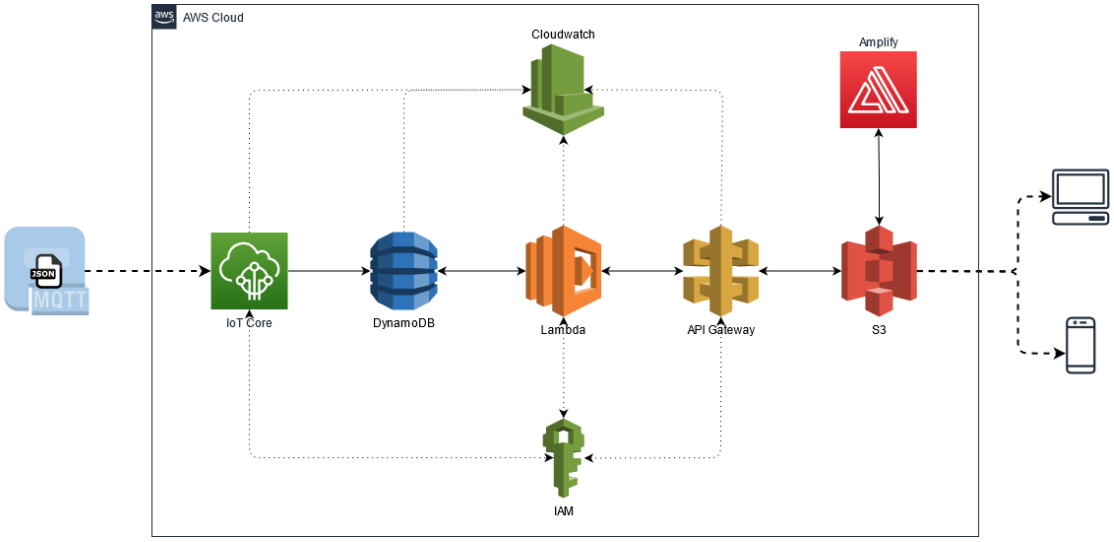

Figure 5.3: Data flow within AWS

5.3.2 Tier 3: Implementation

5.3.2.1 Data reception and storage

This section describes the configuration followed in the AWS Cloud to register the MQTT publishing Type-B module and subsequently store and process the retrieved data. The first part of the process is depicted in Figure 5.4 and is relative to setting up the device's shadow within the AWS IoT Core service.

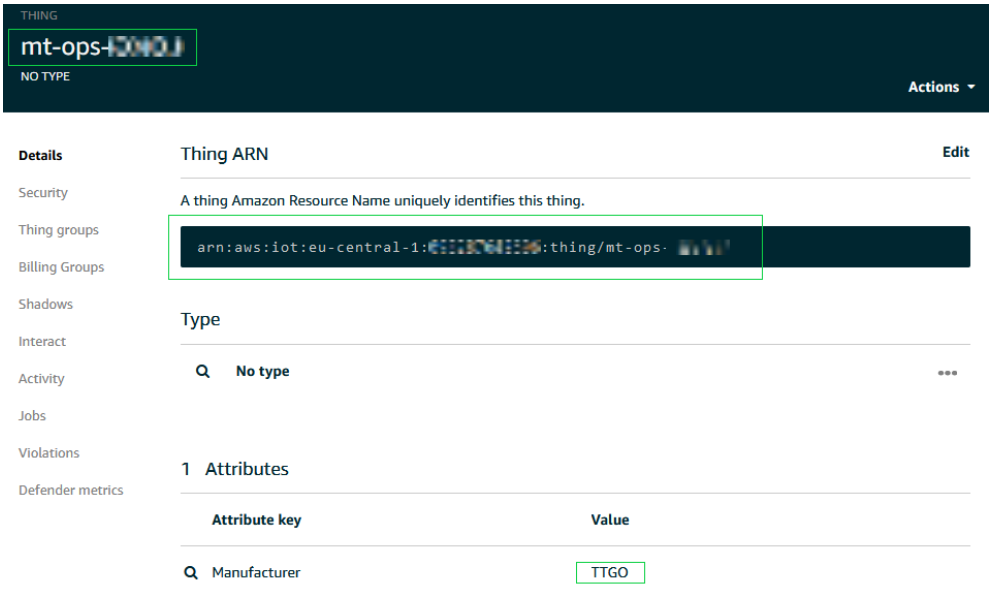

Figure 5.4: MQTT - Type-B module registration

In the registration step, the device's shadow is setup and is given a high-abstraction name to discern it among any additional devices. Each device is named as a "thing" in AWS and is registered automatically upon creation with a unique identifier ARN number. Additional information such as the microcontroller's manufacturer is included in the setup phase, for future aggregation and management of multiple devices deployed in the field. For the purpose of the current implementation the centrally deployed Type-B module has been named as *mt-ops-*****.

Once the device's shadow is created, security credentials are setup in order to ensure that AWS permits communication with the device and that the communication process itself is secure. This is depicted in Figure 5.5, Figure 5.6 and Figure 5.7. The topic that is created has the form of *$aws/things/mt-ops-****/shadow/*, where the $aws/things/ and /shadow/ are the prefix and suffix used for every MQTT topic relative to each thing within AWS, *mt-ops-***** is the name previously given to the specific Type-B module and */update* is the endpoint as specified within the firmware.

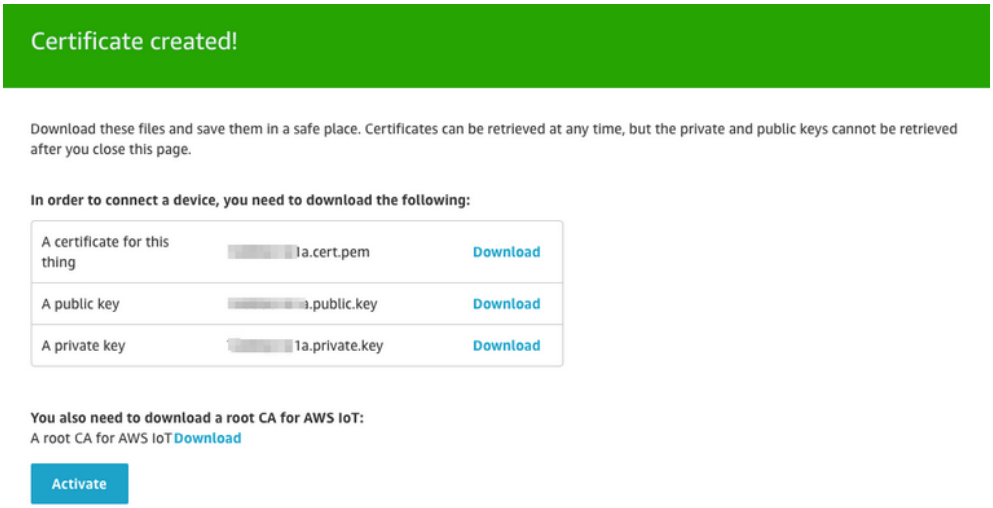

Figure 5.5: MQTT - Type-B module security credentials creation

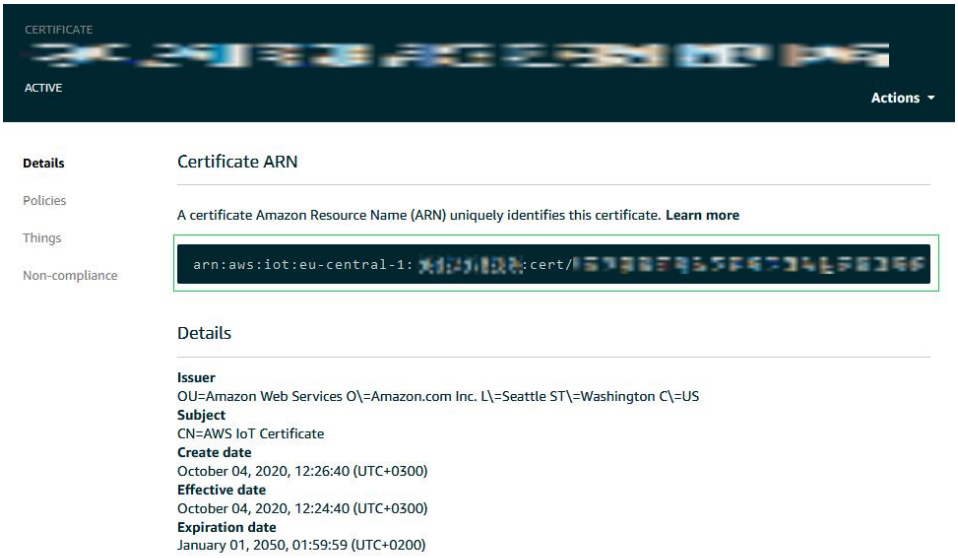

Figure 5.6: MQTT - Type-B module security credentials (Certificate)

The generated credentials include a public and private key pair, as well as a root certificate with RSA 2048 bit encryption which has its own uniquely identifying ARN number. A policy document is then setup and bound to the shadow, in order to allow the latter to access any required resources and functionalities within the AWS IoT Core or other services. The specific policy document depicted in Figure 5.7 specifies that the device is allowed to access every available resource that the IoT Core service provides. The policy is ultimately attached to the certificate created earlier, in order to allow any devices that use it to have identical permissions.

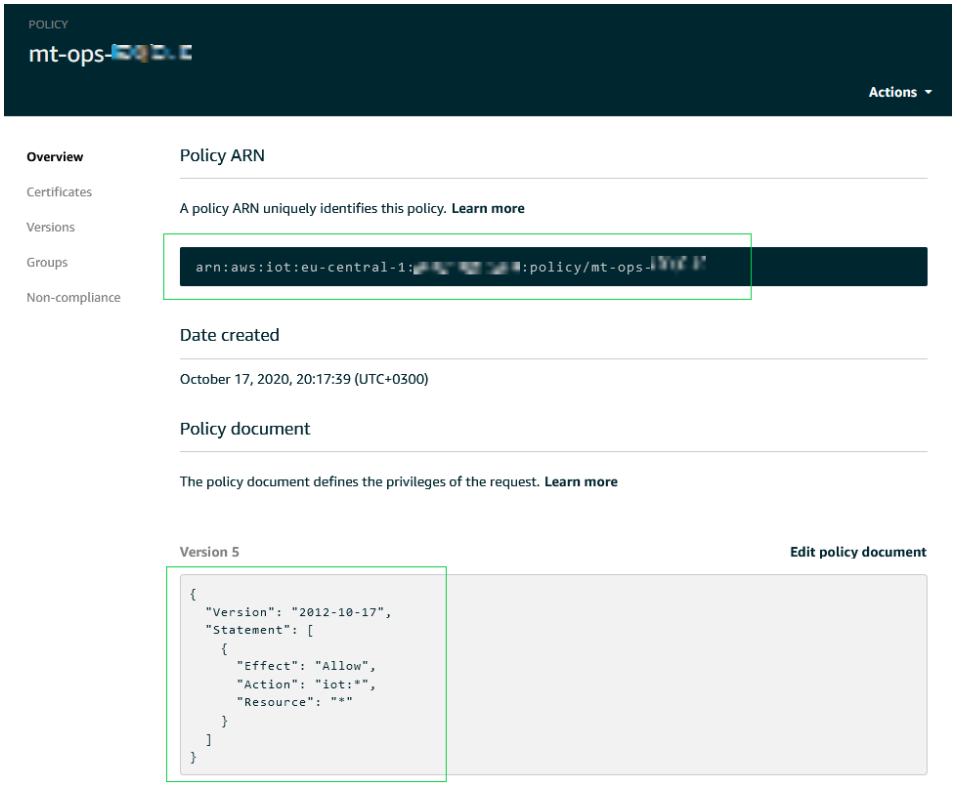

Figure 5.7: AWS IoT Core – Type-B module security credentials (Policy)

For the actual Type-B module to obtain the specified credentials, the specified credentials must be added to the device's firmware, which is done via their addition in the *aws_iot_certificates.c* file which as accessed and handled via the *AWS_IOT.h* library defined in the firmware, as mentioned in the previous subsection. The specification of the credentials is in encoded form and a typical encoded certificate within the mentioned file can be depicted in Code Snippet 13.

```
/**
 * @file aws_iot_certifcates.c
 * @brief File to store the AWS certificates in the form of arrays
 */

#ifdef __cplusplus
extern "C" {
#endif

const char aws_root_ca_pem[] = {"-----BEGIN CERTIFICATE-----\n\
**********************************************************\n\
-----END CERTIFICATE-----\n"};

const char certificate_pem_crt[] = {"-----BEGIN CERTIFICATE-----\n\
**********************************************************\n\
-----END CERTIFICATE-----\n"};
```

```
const char private_pem_key[] = {"-----BEGIN RSA PRIVATE KEY-----\n\
**********************************************************\n\
-----END RSA PRIVATE KEY-----\n"};


#ifdef __cplusplus
}
#endif
```

Code snippet 13: Specification of security credentials within Type-B firmware

AWS provides an MQTT test client for testing and verification purposes. Following the mentioned configuration, a test run shows a successful subscription to the created topic. The data received in the test run is in the JSON syntax in which it has been formatted to by the Type-B module prior to MQTT publishing.

Since the MQTT subscription has been established and data is made available to AWS, it can now be split and stored within a database. As mentioned earlier, the implementation employs a DynamoDB NOSQL database where a table has been provisioned to hold the split data in order for subsequent processes to access and manipulate accordingly. For this purpose a DynamoDB table named *MQTTDataBulk* was created using a composite key for each of its entries, which is comprised of two items; the device ID as the Partition key and the timestamp as the Sort key. Ultimately each measurement will be stored in the table and identified by a dual identifier comprised of the mentioned components, following a relationship of *device_id - timestamp.*

Moving back to the point of data reception in the Cloud, splitting and storing the data is implemented at the IoT Core level via the provided query functionality mentioned earlier. A query used in this manner enables the creation of columns within a specified DynamoDB table and ultimately the population of the table based on the key-value structure of the queried JSON object. In this case the query used dictates that columns based on the JSON keys are to be created in the *MQTTDataBulk* table previously initialized. The columns in which the data will be split towards are *device_id*, *timestamp*. *known_devices* and *unknown_devices*, based on the related keys present in each Type-B-published JSON object, as can been seen in Figure 5.8. Meanwhile, the values paired with the mentioned keys within each object are stored into each of the created columns. The query specifically used for the current data format is depicted in Code Snippet 14.

```
SELECT id as device_id,
       unk as unknown_devices,
       kn as known_devices,
```

```
      ts as timestamp
FROM '$aws/things/mt-ops-****/shadow/update'
```

Code snippet 14: Query for DynamoDB data split

The final result is the table depicted in Figure 5.8, which in this snapshot holds readings captured from 2 different Type-A modules. The data flow loop is thus now closed, starting from the measurement to the data-store level; all measurements procured by the surveyed area, were transmitted to the Type-B centralized node, formatted into JSON format, published via MQTT and captured via a subscription in AWS, to ultimately be queried upon and stored into the provisioned database.

Figure 5.8: DynamoDB structure / stored data

5.3.2.2 Data invocation and processing

Since all the required information is now stored within a database in the AWS domain, it is ready to be used via a multitude of services and served to the front-end client. For this purpose, a group of serverless Lambda functions is created to handle data processing, when invoked by an equivalent number of APIs developed for this purpose. The APIs have been developed via the Amazon API Gateway service, and have specific signatures depending on which kind of processing they will be triggering:

- /device/{id} – GET : Returns the latest measurements per Type-A device, for depiction of the raw data. The API triggers the getDeviceData Lambda function.
- /aggregate/ – GET : Returns the last measurement per Type-A device, for close to real-time depiction on a map. The API triggers the aggregateData Lambda function.
- /location/ – GET : Returns the position of a device from 3 Type-A modules in its vicinity. The API triggers the trilateratePosition Lambda function.

Designing a typical API in the AWS Management Console, leads to an API architectural flow as shown in Figure 5.9 with regard to the /device/{id} endpoint. The left pane shows all created endpoints within the API while the Lambda function to be triggered upon the API's invocation is also visible on the right. In this case the depicted API triggers the *getDeviceData* function which will be described on the next subsection. The API Gateway service also provides a testing functionality, where deployed APIs can be fed with their expected input to show what response they will return to the requester. Testing the mentioned endpoint through this test client, returns the response defined in the *getDeviceData* function in the form visible in Figure 5.10, for the case of device 2.

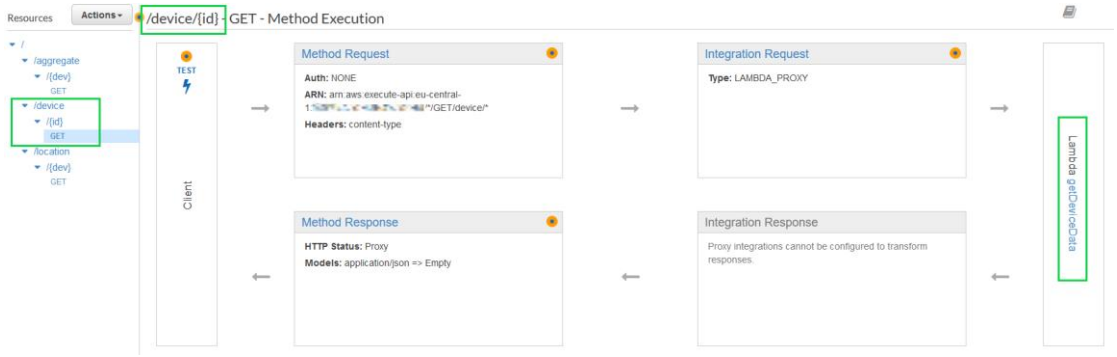

Figure 5.9: API endpoints and /device/{id} signature

Figure 5.10: Sample response for /device/2 GET request

Each of the mentioned API endpoints is pointing towards an individual Lambda function that performs a specific set of processes upon the stored data. These functions are all developed in Node.js and are described in detail in the following subsections. For the Lambda functions to

access the required resources, specific policies have been defined and attached on each via the Amazon Identity and Access Management (IAM) service, such as the one featured in Code Snippet 15.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "****",
            "Effect": "Allow",
            "Action": [
                "dynamodb:BatchGetItem",
                "dynamodb:ConditionCheckItem",
                "dynamodb:PutItem",
                "dynamodb:DescribeContributorInsights",
                "dynamodb:ListTagsOfResource",
                "dynamodb:Query",
                "dynamodb:DescribeStream",
                "dynamodb:DescribeTimeToLive",
                "dynamodb:DescribeGlobalTableSettings",
                "dynamodb:PartiQLSelect",
                "dynamodb:DescribeTable",
                "dynamodb:GetShardIterator",
                "dynamodb:DescribeGlobalTable",
                "dynamodb:GetItem",
                "dynamodb:DescribeContinuousBackups",
                "dynamodb:DescribeExport",
                "dynamodb:DescribeKinesisStreamingDestination",
                "dynamodb:DescribeBackup",
                "dynamodb:GetRecords",
                "dynamodb:DescribeTableReplicaAutoScaling"
            ],
            "Resource": [
              "arn:aws:dynamodb:eu-central-1:****:table/MQTTDataBulk",
              "arn:aws:dynamodb:eu-central-1:****:table/MQTTDataBulkM2"
             ]
        }
    ]
}
```

Code snippet 15: Lambda functions' security Policy

This policy grants any Lambda function it is attached to, the permission to perform the operations included in it, with the most prominent being the one that allows querying the data in the *MQTTDataBulk* table.

5.3.2.2.1 Detailed per-device readings

The first kind of data visualization to occur in the front-end client refers to the depiction of the latest readings procured by each of the Type-A modules within the surveyed area. This process is invoked by the device/{id} GET method which, as mentioned, triggers the *getDeviceData* Lambda function shown in Code snippet 16.

```
const AWS = require('aws-sdk');
AWS.config.update({ region: "eu-central-1" });

exports.handler = async (event, context) => {
    const ddb = new AWS.DynamoDB({ apiVersion: "2012-10-08" });
    const documentClient = new AWS.DynamoDB.DocumentClient({ region: "e
u-central-1" });

    let responseBody = "";
    let statusCode = 0;
    const { id } = event.pathParameters;

    const params = {
        ExpressionAttributeValues: { ":v1": id },
        KeyConditionExpression: "device_id = :v1",
        TableName: "MQTTDataBulk",
        ScanIndexForward: false,
        Limit: 100
    };

    try {
        const data = await documentClient.query(params).promise();
        responseBody = JSON.stringify(data);
        statusCode = 200;
    } catch (err) {
        responseBody = 'Unable to get device data.'
        statusCode = 403
    };

    const response = {
        statusCode: statusCode,
        headers: {
            "Access-Control-Allow-Origin" : "*",
            "Access-Control-Allow-Credentials" : true
        },
        body: responseBody
    }

    return response;
};
```

Code snippet 16: Lambda Function - getDeviceData

The process involves querying the *MQTTDataBulk* table that holds all the devices' measurements, depending on which device id the API endpoint is invoked by, for example *https://**********-api.eu-central-1.amazonaws.com/default/device/3*. This query returns the readings for the device with device_ID = 3, sorted by their timestamp in a descending order. The latter is set using the *ScanIndexForward* query parameter in the query. For this implementation the number of latest readings to be shown is specified to 100 via the *Limit* parameter, which at a high level refers to roughly the last 1h40m of measurements, since each microcontroller polls its Wi-Fi range approximately every minute. Successful invocation of the process will return a 200 response and the query's result in JSON format to the requester. In the current implementation, 3 different devices are supported in the relevant page of the platform, thus 3 API invocations will take place, each differentiated by the device_id in the API request path.

5.3.2.2.2 Latest reading per device

The next kind of data to appear in the front-end client is the visualization of the last reading of each module on a map. To accomplish this, the /aggregate/{id} GET request is invoked which triggers the *aggregateData* Lambda function. This function is quite similar to the previously mentioned *getDeviceData* one, with the difference that this one only returns a single latest reading in order to allow for almost real-time visualization of the measurement when the system is in full working order, or the last transmitted reading when possible failure on a Type-A module occurs. The function is shown in Code snippet 17.

```
const AWS = require('aws-sdk');
AWS.config.update({ region: "eu-central-1" });

exports.handler = async (event, context) => {
    const ddb = new AWS.DynamoDB({ apiVersion: "2012-10-08" });
    const documentClient = new AWS.DynamoDB.DocumentClient({ region: "e
u-central-1" });

    let responseBody = "";
    let statusCode = 0;

    const params1 = {
        ExpressionAttributeValues: { ":v1": "1" },
        KeyConditionExpression: 'device_id = :v1' ,
        TableName: "MQTTDataBulk",
        ScanIndexForward: false,
        Limit: 1
    };
```

```
    const params2 = {
        ExpressionAttributeValues: { ":v2": "2"},
        KeyConditionExpression: 'device_id = :v2' ,
        TableName: "MQTTDataBulk",
        ScanIndexForward: false,
        Limit: 1
    };

    const params3 = {
        ExpressionAttributeValues: { ":v3": "3" },
        KeyConditionExpression: 'device_id = :v3' ,
        TableName: "MQTTDataBulk",
        ScanIndexForward: false,
        Limit: 1
    };

    let data1,data2,data3;

    let aggr1 = [];
    let aggr2 = [];
    let aggr3 = [];

    try {
        data1 = await documentClient.query(params1).promise();
        data2 = await documentClient.query(params2).promise();
        data3 = await documentClient.query(params3).promise();
        statusCode = 200;
    } catch (err) {
        responseBody = 'Unable to get device data.'
        statusCode = 403
    };

        const known1_level1 = data1.Items;
        const known1_level2 = known1_level1.map(x => x.known_devices.ma
p(y => y.dev));
        const known1_level2B = known1_level1.map(x => x.unknown_devices
);
        aggr1.push(known1_level2[0]);
        aggr1.push(known1_level2B[0]);

        const known2_level1 = data2.Items;
        const known2_level2 = known2_level1.map(x => x.known_devices.ma
p(y => y.dev));
        const known2_level2B = known2_level1.map(x => x.unknown_devices
);

        aggr2.push(known2_level2[0]);
```

```
        aggr2.push(known2_level2B[0]);

        const known3_level1 = data3.Items;
        const known3_level2 = known3_level1.map(x => x.known_devices.ma
p(y => y.dev));
        const known3_level2B = known3_level1.map(x => x.unknown_devices
);

        aggr3.push(known3_level2[0]);
        aggr3.push(known3_level2B[0]);

        let resp=[];

        resp.push(aggr1,aggr2, aggr3);

        responseBody = JSON.stringify(resp);

   const response = {
        statusCode: statusCode,
        headers: {
            "Access-Control-Allow-Origin" : "*",
            "Access-Control-Allow-Credentials" : true
        },
        body: responseBody
    }
    return response;
};
```

Code snippet 17: Lambda Function - aggregateData

For the current implementation, the function has been configured to return the latest reading of 3 devices, since these are the specific ones to be depicted in the current development stage of the application The page that will invoke this API, is analyzed on the Tier 4 subsection of this chapter and will ultimately depict which of the known devices were last identified on each of the 3 locations covered by a Type-A module.

5.3.2.2.3 Distance from 3 sensors

The final kind of data visualization that the current platform provides, refers to the calculation of the distance of a detected known device when placed within the range of three adjacent Type-A sensors. This functionality is developed experimentally to test a different scenario, where the sensing modules have been deployed in close proximity within a congested area, thus making their detection ranges overlap. This hypothesis leads to the investigation of whether or not the measured RSSI values can lead to meaningful distance estimations for the

rescue teams. The function used for this purpose is called *trilateratePosition* and is triggered via the */location/{id}* GET method. Its context can be seen on Code snippet 18.

```
const AWS = require('aws-sdk');
AWS.config.update({ region: "eu-central-1" });

exports.handler = async (event, context) => {
    const ddb = new AWS.DynamoDB({ apiVersion: "2012-10-08" });
    const documentClient = new AWS.DynamoDB.DocumentClient({ region: "e
u-central-1" });

    let responseBody = "";
    let statusCode = 0;

    const  dev_name  = "MotoG";

    const params1 = {
        ExpressionAttributeValues: {
            ":v1": "1",
        },
        KeyConditionExpression: 'device_id = :v1' ,
        TableName: "MQTTDataBulkM2",
        ScanIndexForward: false,
        Limit: 10
    };

    const params2 = {
        ExpressionAttributeValues: {
            ":v2": "2",
        },
        KeyConditionExpression: 'device_id = :v2' ,
        TableName: "MQTTDataBulkM2",
        ScanIndexForward: false,
        Limit: 10
    };

    const params3 = {
        ExpressionAttributeValues: {
            ":v3": "3",
        },
        KeyConditionExpression: 'device_id = :v3' ,
        TableName: "MQTTDataBulkM2",
        ScanIndexForward: false,
        Limit: 10
    };

    let data1,data2,data3;

    let known1 = [];
    let known2 = [];
    let known3 = [];
```

```
    try {
        data1 = await documentClient.query(params1).promise();
        data2 = await documentClient.query(params2).promise();
        data3 = await documentClient.query(params3).promise();

        const known1_level1 = data1.Items;
        const known1_level2 = known1_level1.map(x => x.known_devices);
         const known2_level1 = data2.Items;
        const known2_level2 = known2_level1.map(x => x.known_devices);
        const known3_level1 = data3.Items;
        const known3_level2 = known3_level1.map(x => x.known_devices);

        for (let i = 0; i <= known1_level2.length+known2_level2.length+
known3_level2.length ; i++) {
            known1.push(known1_level2[i].filter(x => x.dev == dev_name)
.map((key) => key.rssi));
            known2.push(known2_level2[i].filter(x => x.dev == dev_name)
.map((key) => key.rssi));
            known3.push(known3_level2[i].filter(x => x.dev == dev_name)
.map((key) => key.rssi));
        }

        statusCode = 200;
    } catch (err) {
        responseBody = 'Unable to get device data.'
        statusCode = 403
    };

    known1 = known1.map(item => item[0]);
    known2 = known2.map(item => item[0]);
    known3 = known3.map(item => item[0]);

    // Calculations for beacon 1
    let rssi_sum = 0;
    let counter = 0;

    for (let i=0 ; i<known1.length; i++) {
        if (known1[i]) {

            let reading = known1[i].replace('-','');
            console.log("Beacon 1 Reading: ", parseInt(reading));
            rssi_sum += parseInt(reading);
         }
         else {
            rssi_sum += 0;
            counter ++;
         }
    }

    let length1 = 0;

    if (counter>0) {
        length1 = known1.length - counter;
    }
```

```
    else {
        length1 = known1.length;
    }

    let rssi_avg=rssi_sum/length1;
    console.log("RSSI AVG: " ,rssi_avg);

    // SAMPLE : d=10^((-61+69.6)/10)
    let AA = (-62.3 + rssi_avg)/(10*(3));
    let dist_1 = Math.pow(10, AA).toFixed(2);

    console.log("THE DISTANCE TO BEACON 1 IS: ", dist_1);

    // Calculations for beacon 2
    rssi_sum = 0;
    rssi_avg = 0;
    counter = 0;

    for (let i=0 ; i<known2.length; i++) {
        if (known2[i]) {
            let reading = known2[i].replace('-','');
            console.log("Beacon 2 Reading: ", parseInt(reading));
            rssi_sum += parseInt(reading);
        }
        else {
            rssi_sum += 0;
            counter ++;
        }
    }

    let length2 = 0;

    if (counter>0) {
        length2 = known2.length - counter;
    }
    else {
        length2 = known2.length;
    }

    rssi_avg=rssi_sum/length2;
    console.log("RSSI AVG: " ,rssi_avg);

    // SAMPLE : d=10^((-61+69.6)/10)
    AA = (-62.3 + rssi_avg)/(10*(3));
    let dist_2 = Math.pow(10, AA).toFixed(2);

    console.log("THE DISTANCE TO BEACON 2 IS: ", dist_2);

    // Calculations for beacon 3
    rssi_sum = 0;
    rssi_avg = 0;
    counter = 0 ;

    for (let i=0 ; i<known3.length; i++) {
```

```
        if (known3[i]) {
            let reading = known3[i].replace('-','');
            console.log("Beacon 3 Reading: ", parseInt(reading));
            rssi_sum += parseInt(reading);
        }
        else {
            rssi_sum += 0;
            counter ++;
        }
    }

    let length3 = 0;

    if (counter>0) {
        length3 = known3.length - counter;
    }
    else {
        length3 = known3.length;
    }

    rssi_avg=rssi_sum/length3;
    console.log("RSSI AVG: " ,rssi_avg);

    // SAMPLE : d=10^((-61+69.6)/10)
    AA = (-62.3 + rssi_avg)/(10*(3));

    let dist_3 = Math.pow(10, AA).toFixed(2);

    console.log("THE DISTANCE TO BEACON 3 IS: ", dist_3);

    // Transmit final results
    let distance_vector={
        "distance_1" : dist_1,
        "distance_2" : dist_2,
        "distance_3" : dist_3
    }

    responseBody = JSON.stringify(distance_vector);

    const response = {
        statusCode: statusCode,
        headers: {
            "Access-Control-Allow-Origin" : "*",
            "Access-Control-Allow-Credentials" : true
        },
        body: responseBody
    }

    return response;
};
```

Code snippet 18: Lambda Function - trilateratePosition

This function performs a set of calculations as defined in [15][42], where the signal strength measured between 2 points can be used to calculate an estimation of their distance. The equation used is :

$$P_L(d) = P_L(d_0) + 10nlog\left(\frac{d}{d0}\right) + X_\delta \quad (1)$$

where PL(d) is the RSSI between two nodes separated by distance d, $PL(d_0)$ is the RSSI at the receiving node at distance d_0 and n is the path-loss propagation exponent which varies depending on surrounding environment that the nodes are situated in. X_δ is a Gaussian random variable indicating the error margin. [15] indicates that "the measurement error in RSSI does not regularly produce a Gaussian distribution. However, by using approximation (curve fitting), the RSSI measurement error is treated as a Gaussian random variable for simplicity. Consequently, the reference distance is usually taken as one meter", which results in the following equation:

$$RSSI = A - 10nlog(d) \quad (2)$$

where RSSI signifies the signal strength at a distance d from the signal source, and A refers to the signal strength at a 1 m distance from it. That said, the distance is equivalent to:

$$d = 10^{\frac{A-RSSI}{10n}} \quad (3)$$

For the current implementation the function has been configured to perform the mentioned calculations for one predefined specific device. The process involves querying against the table that holds the measurements, for the latest 10 entries retrieved by each of the 3 adjacent Type-A modules with relation to the specified device. The sum of the RSSI values is calculated for each of the 3 sensors and some data handling is performed in case there are less than 10 values retrieved. The average RSSI value is then calculated with regard to the device and each of the 3 Type-A modules, based on the mentioned sum and count values.

An experimental process was carried out to measure the RSSI value at the reference distance d_0 equal to 1m as suggested by [15], which involved performing 30 RSSI value measurements at this distance and calculating the average. This resulted in specifying the value of A in *(2)* and *(3)* at -62.3dB. Additionally, n has been set to 3, as the typical value used in non-free space indoor environments ranges from 3 to 4 [15], [37] . Ultimately the distance is calculated

in an identical manner on 3 parts of the code, corresponding to each of the Type-A modules that sensed the known device, and the final value calculation is performed via equation (3).

The three individual results are returned as an API response of *location/{id}* GET method to the requester, and can be used in future implementations to pinpoint a device's location through trilateration [34], [35], [37], [42], [43], or other means. For this implementation a table will hold the three individual distances measured, between the known device, which signifies a resident, and the 3 nodes surrounding them.

5.4 Tier 4: Data visualization

The final part that comprises the platform is the visualization level, where all the gathered and processed data are made available to the rescue teams and anyone involved in the rescue operations. To that end, a cloud-based front-end client has been developed as will be further analyzed in the current section.

5.4.1 Tier 4: Specifications

As defined in the application's architectural overview, the front-end client aiming at first responders must share the same specifications as the rest of the underlying components, most notably robustness and security. The platform should be accessible only by the people it is aimed at and should have zero down-time since its use case for relates to actual human lives. For that reason the choice of technologies that the client has been based upon lies within the AWS service range, just as the data processing tier does. The technologies used to that end are:

- AWS Simple Storage Service (S3)
- AWS Amplify
- Amazon Cognito
- React.js
- React-leaflet.js

S3 is service designed to store data at a reliable, high performance and massive scale. It supports the added functionality to host a static website, which reduces the complexity and maintenance needs tied with a web application that would otherwise be deployed on a provisioned web server. Additionally this paradigm makes it possible to integrate any number of additional external services on the fly, since no redeployment is required on an infrastructural level. For this purpose, the React.js framework was chosen to cooperate with

the mentioned functionality, by providing the means of creating a static website through which all previously mentioned data invocation and processing can be accessed.

Consequently, the entire approach ensures that all processes ranging from data reception to data serving to the final end-users will be carried out in a serverless approach to fulfill the requirement of low to no maintenance. Moreover, the solution allows the security aspect to be managed by the AWS Amplify service as described in the following section.

5.4.2 Tier 4: Implementation

The process of creating the AWS-based front-end client, starts from provisioning an S3 bucket to host it. For this purpose a bucket was created and given the proper policies that will allow it to be accessed from outside the AWS domain, while granting specific permissions towards any related resources. Code snippet 18 shows the policy to allow access to the bucket and subsequently the application itself.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "PublicReadGetObject",
            "Effect": "Allow",
            "Principal": {
                "AWS": "*"
            },
            "Action": "s3:GetObject",
            "Resource": "arn:aws:s3:::****-host-bucket/*"
        }
    ]
}
```

Code snippet 19: Host S3 Bucket access policy

Once the bucket has been setup for hosting, the endpoint which serves as the application's URL is visible in its settings section, as shown in Figure 5.11. For reference, additional services such as Amazon Route 53 can be used to specify a custom domain, but for the current implementation the domain generated by default has been kept as is for all intends and purposes.

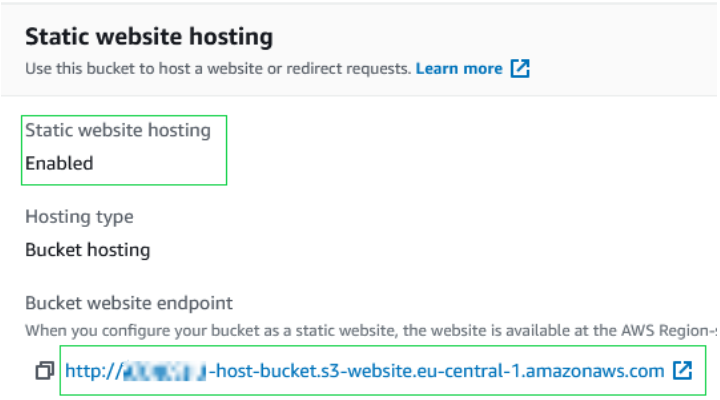

Figure 5.11: S3 bucket public endpoint

The next step to configuring an S3 bucket as a hosting service, is a matter of uploading static website code into it; the rest is handled inherently and the website ultimately goes live in a matter of seconds. The code developed for this solution, was implemented locally using the React framework and the npm package manager; any generated assets within the *src* folder of the locally deployed application are moved into the hosting bucket.

The user-facing front-end is comprised of 3 major sections in this implementation, as mentioned earlier in the previous tier's subchapter. These sections refer to :

- A view that provides the depiction of the last 1h40min of readings for each of three deployed Type-A sensors
- A view that contains a map with the location of the 3 deployed sensors, with the latest reading each one has procured.
- A view that shows the distance calculation of a device with relation to three closely deployed Type-A sensors.

Code Snippet 20 shows the main handler of the front-end client.

```
import React from 'react';
import './App.css';
import { Route, BrowserRouter as Router, Link, Switch } from "react-
router-dom";
import Nav from "./components/Nav"
import Home from "./Pages/Home"
import Detailed from "./Pages/Detailed"
import Aggregated from "./Pages/Aggregated";
import Map from "./Pages/Map";
import Amplify from 'aws-amplify';
import awsconfig from './aws-exports';
import { AmplifySignOut, withAuthenticator } from '@aws-amplify/ui-
react'
```

```
Amplify.configure(awsconfig)

function App() {

  return (
    <div>
      <Router>
        <div>
          <Nav />
          <Switch>
            <Route path="/" exact component={Home} />
            <Route path="/details" component={Detailed} />
            <Route path="/Map" component={Map} />
            <Route path="/aggregate" component={Aggregated} />
          </Switch>
        </div>

      </Router>
    </div>
  );
}

export default withAuthenticator(App);
```

Code snippet 20: Front-end client : App.js module

The module starts with the import of any necessary globally used components and libraries, such as the Amplify library. Each developed component such as the navigation bar, is created individually and imported in this handler, to enable for easier development and controllability over possible failures. The React Router is used to configure routes that show only the components we specify on the page depending on the route, namely each of the mentioned three pages.

The front-end client's security aspect is handled via AWS Amplify. The service makes use of Amazon Cognito service to leverage user creation, management and authentication. The integration of this functionality within the platform's front-end client is implemented via a set of credentials that are defined via the AWS CLI and results in the integration of a login/signup system provided by the service. The resulting configuration file that sets the permissions with regard to Amazon Cognito is shown in Code snippet 21. The AWS Cognito dashboard allows the administrator to manage any existing credentials, add or delete users, specify the required password length and expiration period, enable MFA authentication and

password recovery method and others. In the current implementation user authentication on signup is performed with the use of a one-time password (OTP).

```
const awsmobile = {
    "aws_project_region": "us-east-1",
    "aws_cognito_identity_pool_id": "us-east-1:****",
    "aws_cognito_region": "us-east-1",
    "aws_user_pools_id": "us-east-1_****",
    "aws_user_pools_web_client_id": "****",
    "oauth": {}
};

export default awsmobile;
```

Code snippet 22: AWS Amplify configuration

This section contains all the information endpoints relative to the Amazon Cognito configuration used, that will ultimately allow access control over specific users or groups to the application. AWS Amplify is invoked in the application's handler via the last three import lines shown in the snippet, and is initiated by using the mentioned credentials and invoking the *withAuthenticator()* wrapper.

In addition to the main handler, the front-end code includes three list-related components that have been implemented via *List.js*. *List2.js* and *List3.js* to handle the display of the APIs' responses as lists, and are in turn handled by the *withListLoading.js* which holds a higher-order component to be displayed while any API fetch requests are ongoing, as shown in Code snippet 24 below.

```
import React from 'react';

function withListLoading(Component) {
  return function WihLoadingComponent({ isLoading, ...props }) {
    if (!isLoading) return <Component {...props} />;
    return (
      <p style={{ textAlign: 'center', fontSize: '30px' }}>
        Hold on, fetching data may take some time :)
      </p>
    );
  };
}

export default withListLoading;
```

Code snippet 23: Fetch handling via withListLoading.js

The pages shown in the client are based on the invocation of the three developed APIs as described previously in section 5.4.2.2. The code used to serve the data is similar on all three cases and mostly differs in the API endpoint URLs and the number of asynchronous calls performed on each case. Code Snippet 24 shows the code that handles the latest 100 readings view.

```
import React, { useEffect, useState } from 'react';
import '../App.css';
import List from '../components/List';
import withListLoading from '../components/withListLoading';

export default function Details() {

    const ListLoading = withListLoading(List);
    const [appState, setAppState] = useState({
        loading: false,
        data1: null,
        data2: null,
        data3: null,
    });
    useEffect(() => {
        setAppState({ loading: true });
        const apiUrl1 = `https://****.execute-api.eu-central-
1.amazonaws.com/default/device/1`;
        const apiUrl2 = `https://****.execute-api.eu-central-
1.amazonaws.com/default/device/2`;
        const apiUrl3 = `https://****.execute-api.eu-central-
1.amazonaws.com/default/device/3`;

        Promise.all([fetch(apiUrl1), fetch(apiUrl2), fetch(apiUrl3)])
            .then((responses) => {
                return Promise.all(responses.map(function (response) {
                    return response.json();
                }))
            })
            .then((data) => {
                setAppState({ loading: false, data1: data[0], data2: data[1], d
ata3: data[2] });
            });
    }, [setAppState]);

    return (
        <div className='App'>
            <h1>Detailed Results</h1>
            <h4>Display the exact readings from each sensor</h4>

              <table className='Table'>
              <tr>
              <td><div className='container'><h1>Address 1</h1></div></td>
              <td><div className='container'><h1>Address 2</h1></div></td>
```

```
                <td><div className='container'><h1>Address 3</h1></div></td>
                </tr>
                <tr>
                <td> <div className='repo-
container'> <ListLoading isLoading={appState.loading} data={appState.data1} />
</div> </td>
                 <td> <div className='repo-
container'> <ListLoading isLoading={appState.loading} data={appState.data2} />
</div> </td>
                 <td> <div className='repo-
container'> <ListLoading isLoading={appState.loading} data={appState.data3} />
</div> </td>
                </tr>
            </table>
        </div>
    );
}
```

Code snippet 24: Latest 100 readings view

The code visible above, specifically handles asynchronous calls to the /device/{id} API, for each of the three deployed Type-A sensors. The *Promise.all()* method is used to aggregate the results of the multiple calls that are expected to be returned via individual async calls, which in this case trigger queries against the MQTTDataBulk table for each of the 3 deployed Type-A sensors. The reason why the *promise.all()* method is used, is to fulfill all requests before the code execution continues. Once all API calls respond, the returned data is stored into the application state and are rendered as lists via the mentioned *List.js* and *withListLoading.js* components.

In addition to the mentioned, the map page uses of the React-leaflet library to render a map and pinpoint the location of all deployed Type-A nodes based on their known coordinates. The latest reading per device is made available within a popup upon clicking each location pin on the generated map. Code snippet 25 shows the way this is implemented in code.

```
return (
        <div>
            <MapContainer center={[39.9417, 23.6628]} zoom={13} scrollW
heelZoom={true}>
                <TileLayer
                    attribution='&copy; <a href="http://osm.org/copyrig
ht">OpenStreetMap</a> contributors'
            url="https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png" />
                 <Circle
                        center={[39.9417, 23.6628]}
                        fillColor="white"
                        radius={6000}/>
```

```
                  <Circle
                        center={[39.9417, 23.6628]}
                        fillColor="blue"
                        radius={25}/>
                   <Marker position={[39.9594, 23.6837]}>
                        <Popup>
                              <div class="location-
titles">Location 1 - (39.9417, 23.6628)</div>
                              <table><tr><td>
                                    <div className='map-repo-
container'>

      <ListLoading isLoading={appState.loading} data={appState.data1}/>
                                    </div>
                              </td></tr></table>
                        </Popup>
                   </Marker>
                   <Marker position={[39.93045, 23.72228]}>
                        <Popup>
                              <div class="location-
titles">Location 2 - (39.93045, 23.72228)</div>
                              <table><tr><td>
                                    <div className='map-repo-
container'>

      <ListLoading isLoading={appState.loading} data={appState.data2}/>
                                    </div>
                               </td></tr></table>
                        </Popup>
                   </Marker>
                   <Marker position={[ 39.9569, 23.6242]}>
                        <Popup>
                              <div class="location-
titles">Location 3 - (39.9569, 23.6242)</div>
                              <table><tr><td>
                                    <div className='map-repo-
container'>

      <ListLoading isLoading={appState.loading} data={appState.data2}/>
                                    </div>
                               </td></tr></table>
                        </Popup>
                   </Marker>
              </MapContainer>
         </div>
    );
```

Code snippet 24: Latest 100 readings view

As seen in the snippet above, the MapContainer component is responsible for creating the leaflet map instance. Within it, any of the required specifications are specified, such as the coordinates of the center of the map to be displayed as well as the starting zoom level, via the *center* and *zoom* variables respectively. The *scrolWheelZoom* flag specifies whether the rendered map will be zoomable by the user or not. The map used in this implementation is the OpenStreetMap. Finally the popup that displays the latest measurement from each device is populated via the mentioned *List.js* and *withListLoading.js* code segments.

6

Application testing

In this section a step-by-step analysis is performed upon the information flow that occurs throughout each of the application's tiers. A specific real-world scenario is showcased starting from the detection of a set of devices to their depiction in the final front-end client.

6.1 Measurements & LoRa data transmission

During the initial phase, all residentially-deployed Type-A modules detect any Wi-Fi enabled devices in their proximity. Frequency hopping across all 13 available channels occurs every 1 second, and an aggregation occurs with regard to known/unknown devices, as well as online/offline ones based on the TTL as specified in the module's firmware. The typical setup used in the test case is depicted in Figure 6.1 and is comprised of an ESP32 microcontroller, loaded with the Type-A firmware analyzed previously and connected to a 5000mAh powerbank and a LoRa antenna.

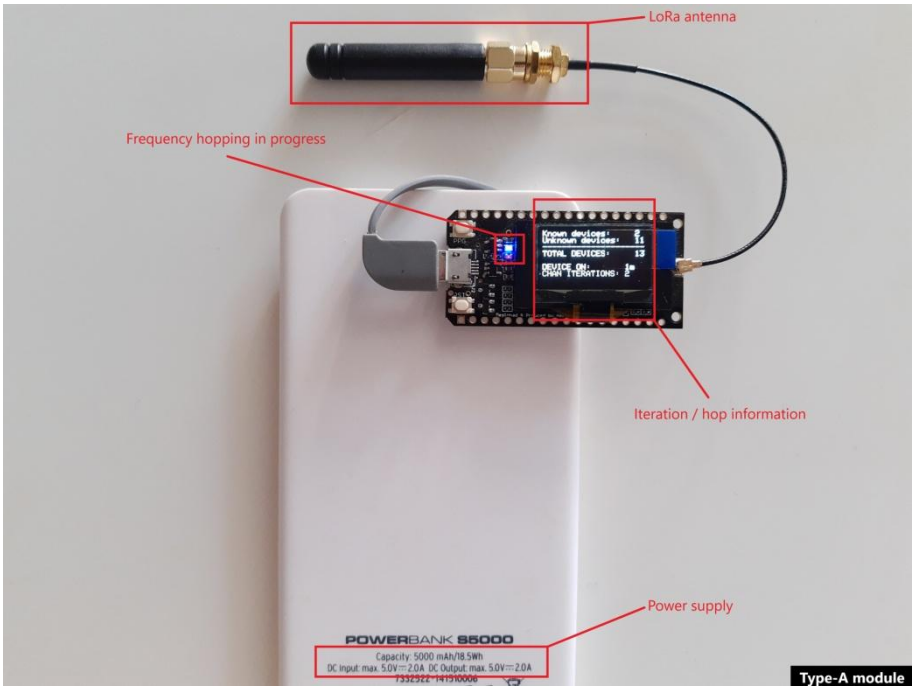

Figure 6.1: Type-A module test setup

Connecting onto a Type-A module while operational, generates informative logs via Arduino IDE's serial monitor as set in each device's firmware for the purpose of monitoring and debugging. Hopping within each of the available 13 channels is signified by flashing the onboard blue LED and the process adds any detected devices in an accumulative list which consists of:

- Each device's MAC address
- The time each device has continuously been designated as being active or within range
- Each device's RSSI value in relation to the Type-A sensing device

A sample channel hop between 2 adjacent channels can be visualized in Figure 6.2. The figure demonstrates the detection of a new device (marked in green) within the device's range and its addition to the accumulative list of any devices detected in previous iterations and/or channel hops (marked in blue).

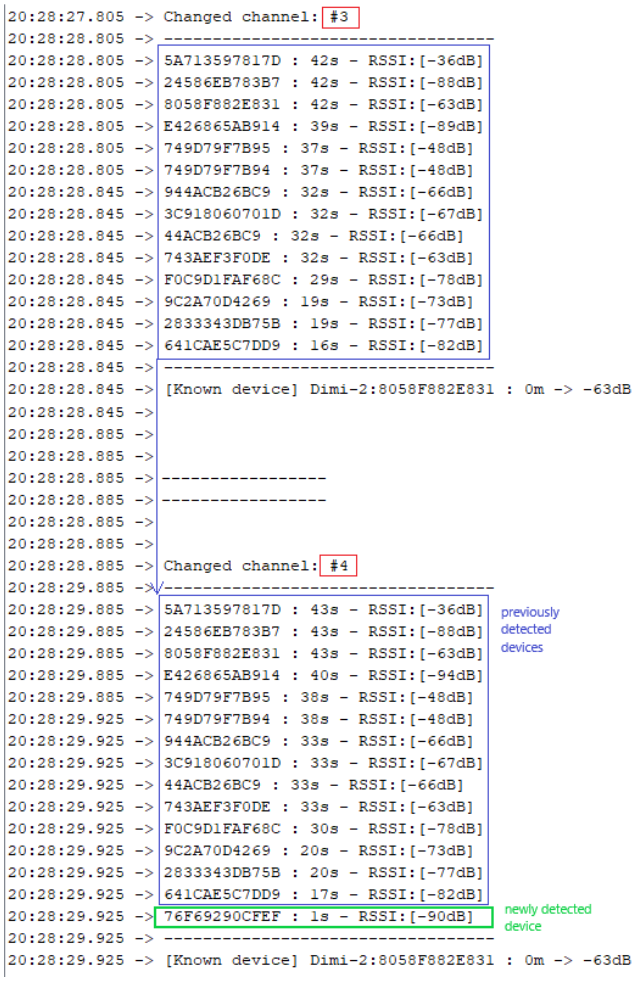

Figure 6.2: Mid-iteration information logging

Once each 13-hop iteration elapses, a complete list of devices is formed along with aggregated information. The overall information shown in Figure 6.3 is comprised of:

- Any previously detected devices still active/within range
- Any previously detected devices designated as OFFLINE due to reaching the specified TTL period
- Any devices newly detected in the latest iteration
- Any known devices detected, along with their specified designation
- The total count of unknown and known devices, as well as their sum
- The number of iterations elapsed since the start of Type-A module's operation

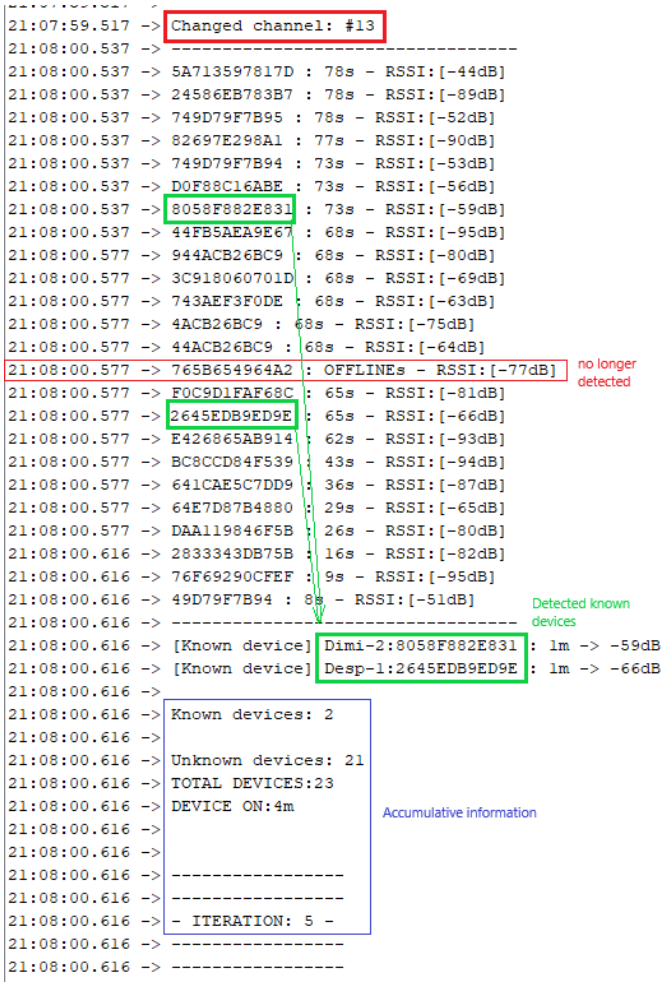

Figure 6.3: Final / end-of iteration information logging

The image above shows that in 4 minutes of operation, this Type-A module has just performed its fifth iteration of 13 channel hops. A total of 23 devices were detected in the surveyed building, two of which are bound to residents and are designated with their own unique names. The two devices have been excluded from the final count of unknown ones and

have been added towards the known devices sum. One of the devices detected in previous iterations has not been detected within range for the TTL duration of more than 60 seconds, thus will be designated as OFFLINE unless it is detected again by the module.

Ultimately, once the 13th channel hop has occurred and the mentioned information has been retrieved, the information is sent via LoRa towards the centrally deployed Type-B module. The packet that will be sent is logged in Arduino IDE's Serial Monitor, as shown in Figure 6.4.

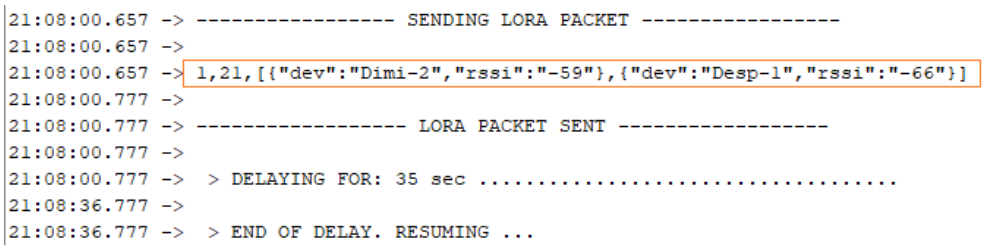

Figure 6.4: Type-A module-transmitted LoRa packet

The packet contains in sequence:

- The current/transmitting Type-A module's ID
- The sum of detected unknown devices
- An array containing all detected devices with
    - Each known device's specified designation/name
    - Each known device's RSSI measurement with regard to the current Type-A module.

In the test case depicted above, the module transmitting the packet has ID=1 and is sending the information acquired at the end of iteration #5 as is shown in Figure 6.3. After transmission takes place, the device goes into delay/sleep mode for 35 seconds, which is also visible in real-time in the serial monitor.

6.2 LoRa data reception & MQTT publishing

All data packets transmitted via LoRa by each deployed Type-A module are aimed towards reception by the centralized Type-B module for formatting and relaying to the Cloud. A typical module used in the test case is depicted in Figure 6.5, which depicts the process of publishing the latest received measurement via MQTT. The duration of publishing is designated by flashing the blue onboard LED.

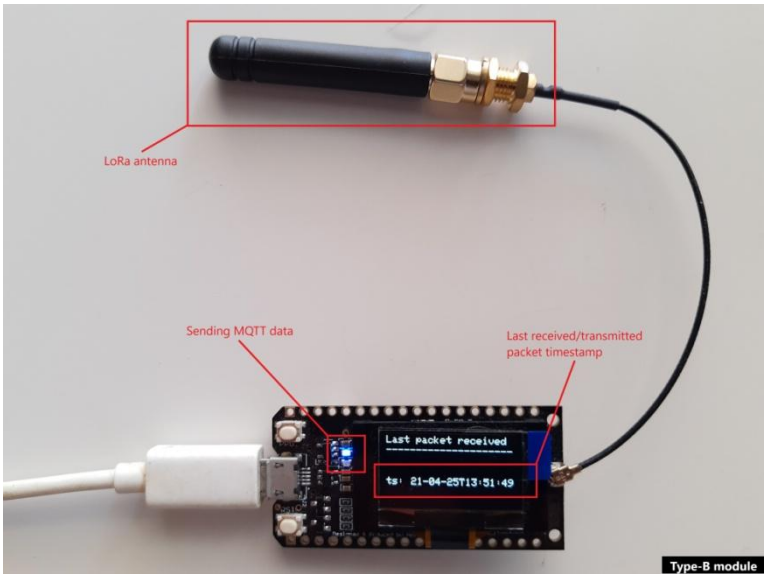

Figure 6.5: Type-B module test setup

Connecting to the Type-B module while on operational mode, provides a set of logs via the Arduino IDE's serial monitor. The initial phase involves the device's connection to the building's Wi-Fi network and the subscription to the MQTT topic specified in the device's firmware. This part is shown in Figure 6.6.

```
20:06:15.214 -> LoRa RCVR/BROKER
20:06:15.294 -> LoRa init - OK!
20:06:15.334 -> Attempting to connect to SSID: MotoVision
20:06:20.414 -> Attempting to connect to SSID: MotoVision
20:06:25.414 -> Connected to wifi
20:06:25.414 -> WiFi init - OK!
20:06:25.574 -> Saturday, April 24 2021 20:06:28
20:06:28.854 -> Connected to AWS
20:06:30.094 -> Subscribe Successfull
```

Figure 6.6: Type-B module initialization check

Each of the initialization phase's steps that the Type-B module follows, is sequentially dependent on its previous and subsequent one, as each one is essential for the module to function properly. Additionally, the initialization will not complete should one of the underlying processes not succeed. In the test case depicted above, the log segment shows that the module first attempts to establish a communication channel in the 868 MHz LoRa band as specified in the module's firmware. Upon success, the module attempts to connect to the network specified in the firmware with the given SSID and password. In this case the network that the device connects to is named "*MotoVision*". Connecting to the designated network serves the additional purpose of connecting to the specified NTP server for the acquisition of

the current time and date, which will be later used to signify the timestamp of each received measurement. The timestamp acquired is logged in the Monitor and the device attempts to connect to AWS with the supplied credentials and subscribing to the specified MQTT topic.

Once the initialization phase completes successfully, the module is at operational mode waiting to receive any LoRa packets sent by the Type-A transmitting modules. Once a packet is received, it is logged, formatted in JSON syntax and logged before being published in the specified MQTT topic. Figure 6.7 shows 3 consequent receptions of LoRa packets, and the MQTT packet to be published.

Figure 6.7: LoRa packet reception, formatting and MQTT publishing

As shown above, the received values are formatted in a format to be easily accessible in the Cloud domain or integrated by other subscribers to the same MQTT topic. A typical JSON object includes the information depicted in Code Snippet 25, which is directly equivalent to packet #4 included in Figure 6.7. Upon publishing the MQTT message, the module receives the message via its subscription to the same topic, which has been implemented for the purpose of monitoring and debugging MQTT data flow.

```
{
    "id": "1",
    "unk": "18",
    "kn": [
        {
            "dev": "Desp-1",
            "rssi": "-55"
        },
        {
            "dev": "Dimi-2",
            "rssi": "-68"
        }
    ],
    "ts": "21-04-24T22:01:01",
    "cnt": "4"
}
```

Code snippet 25: JSON-formatted MQTT packet #4

The JSON object includes key-value pairs with all information sent via LoRa by each of the Type-A modules as shown in Figure 6.4. Key *id* refers to the module that transmitted the initial measurements while keys *unk* and *kn* refer to any detected unknown and known devices respectively. The latter is an array of objects, populated dynamically depending on which known devices were detected in the sensor's range. In addition to the mentioned, the timestamp upon which the LoRa packet was received by the Type-B module is included, as well as the incremental count of packets sent via MQTT since the beginning of the current Type-B module's operation. In this case this refers to the 4th packet published to the topic.

6.3 MQTT data reception & storage

Since the information has been published in the MQTT topic by the centralized module, any subscriber can retrieve and use it accordingly. In this case the AWS IoT Core service has been tasked with capturing any MQTT packets towards their consumption in the Cloud, as described in the previous chapter. Figure 6.7 depicts a sample stream of MQTT packets received in AWS via subscribing to the configured topic. Each message received contains the JSON object transmitted by the centralized module in the form already logged in the latter's operational state as in Figure 6.7.

The information is subsequently queried upon using the parameters mentioned in the previous chapter and is stored in the provisioned NOSQL DynamoDB table named *MQTTDataBulk*. Each object is stored using the device's ID and timestamp as each entry's composite unique key and every value present in the JSON object is segmented within a separate attribute based

on the key it is paired with. In Figure 6.9 a partial segment of the table is shown, where the values appearing in Figure 6.8 are stored in the described manner.

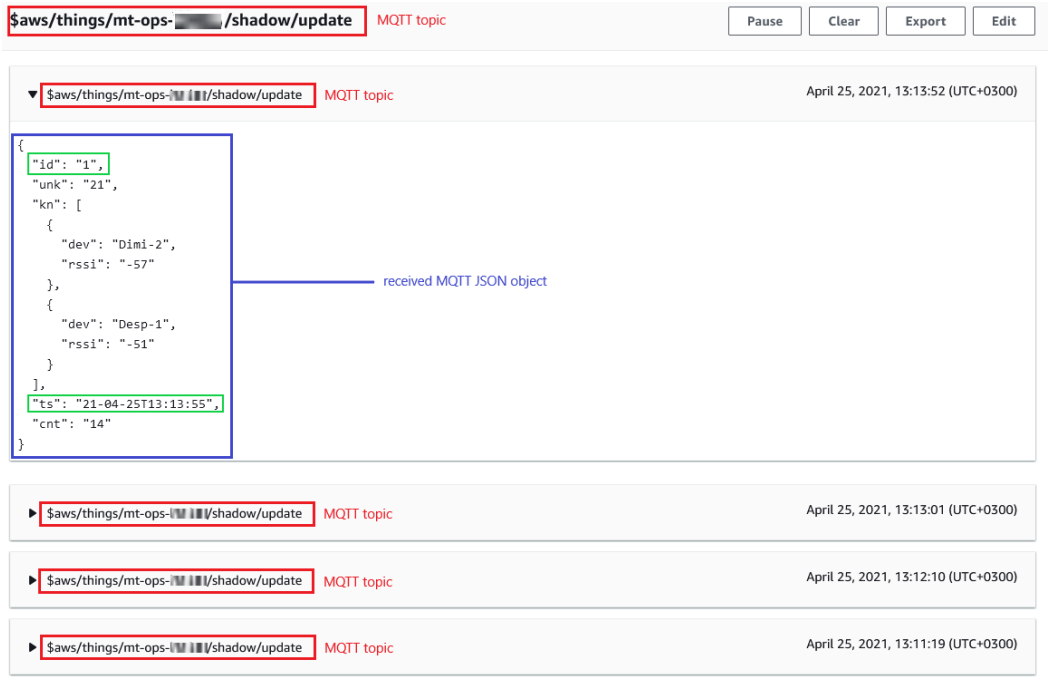

Figure 6.8: MQTT data reception in AWS IoT Core

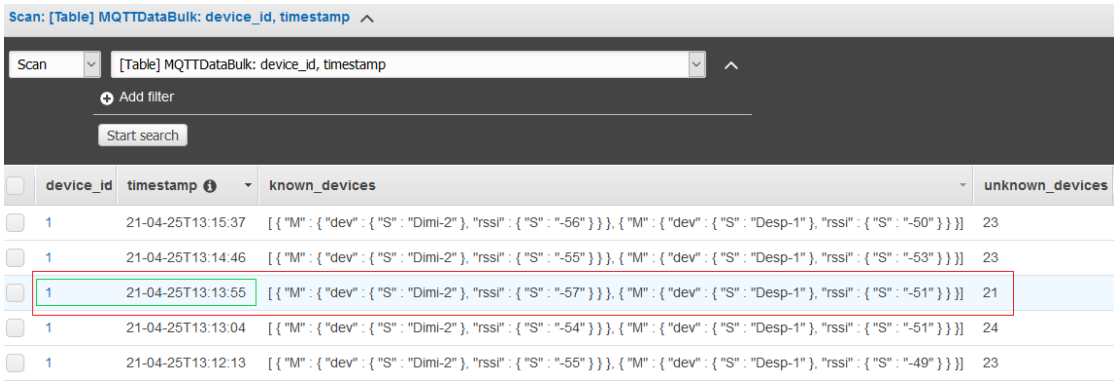

Figure 6.9: DynamoDB - queried MQTT-values' storage

The item highlighted in Figure 6.9 correlates with the specific measurement highlighted in Figure 6.8; the query segmented the JSON object's key value pairs into separate attributes and created a new item within the table using those items. As can be seen, the timestamp stored within each item does not refer to the time that the message was received in AWS; instead it refers to the timestamp that the measurement was received by the Type-B module prior to publishing. Figure 6.10 shows the same item in JSON syntax as shown within the

DynamoDB's interface. The object is a representation of all attributes stored within this specific item and is identical to the data received in the highlighted packet in Figure 6.8.

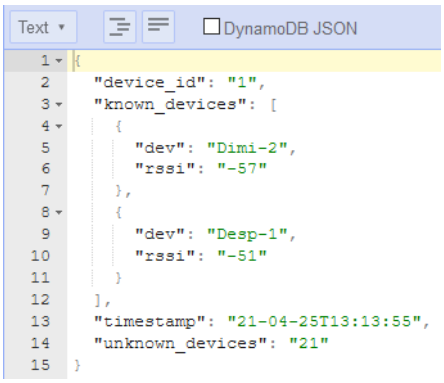

Figure 6.10: DynamoDB item

6.4 Data processing and invocation

Since the data is stored in the table in the expected structure, it is available for processing by the developed Lambdas when triggered by their relative API endpoints. Using Postman those endpoints can be tested in order to observe the mentioned Lambda function's operation. As shown in Figure 6.11, the */device/1* endpoint calls the *getDeviceData* function.

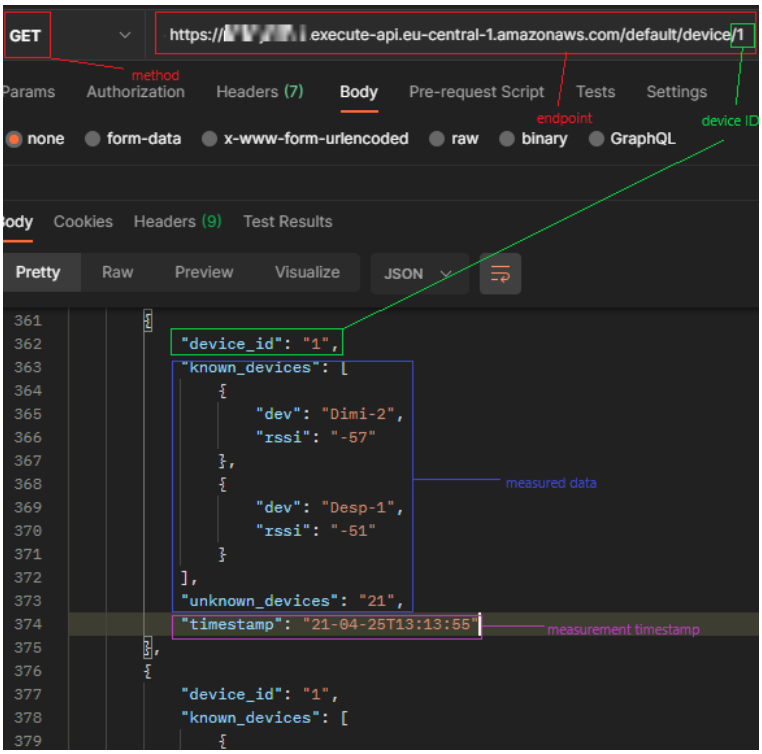

Figure 6.11: GET /device/1 sample request & response

The figure shows the response that is returned by the API when calling the mentioned endpoint as described in the previous chapter, which is expected to return the latest 100 readings from the device we call for in the endpoint. Throughout the retrieved 100 readings, the highlighted segment refers to the specific measurement previously mentioned in Figures 19, 20 and 21. This response segment is a JSON object comprised of:

- The measured data from the Type-A module with ID=1, highlighted in green and blue
- The timestamp assigned to the reading by the Type-B module, highlighted in purple

The second endpoint refers to the retrieval of the latest reading from every device deployed in the surveyed area. Calling the */aggregate/* endpoint via Postman returns the response shown in Figure 6.12.

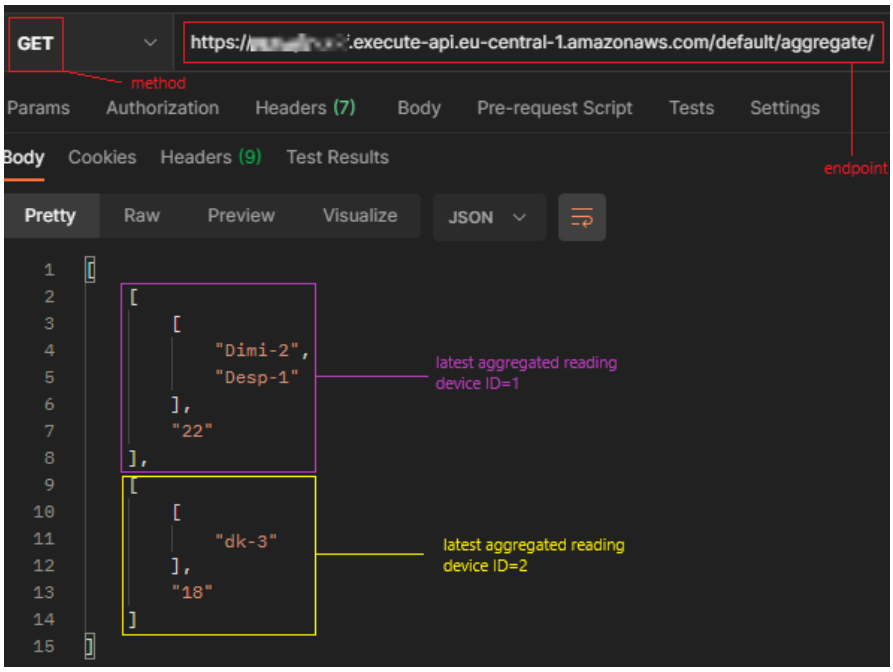

Figure 6.12: GET /aggregate/ sample request & response

The endpoint responds with a custom form of the latest reading retrieved by each of the deployed sensors; in this case the Type-A modules with IDs 1 and 2 which are the ones deployed for the current test case. The response object contains the count of unknown devices as well as dynamic arrays of known devices which are populated based the latest reading per device ID, as stored in the *MQTTDataBulk* DynamoDB table. For reference, the latest readings that correspond to the depicted response are shown in Figure 6.13 and Figure 6.14, for device ID=1 and ID=2 respectively. The two figures depict the results generated upon querying the *MQTTDataBulk* table for the latest stored sensor readings of each of the two devices, in descending order.

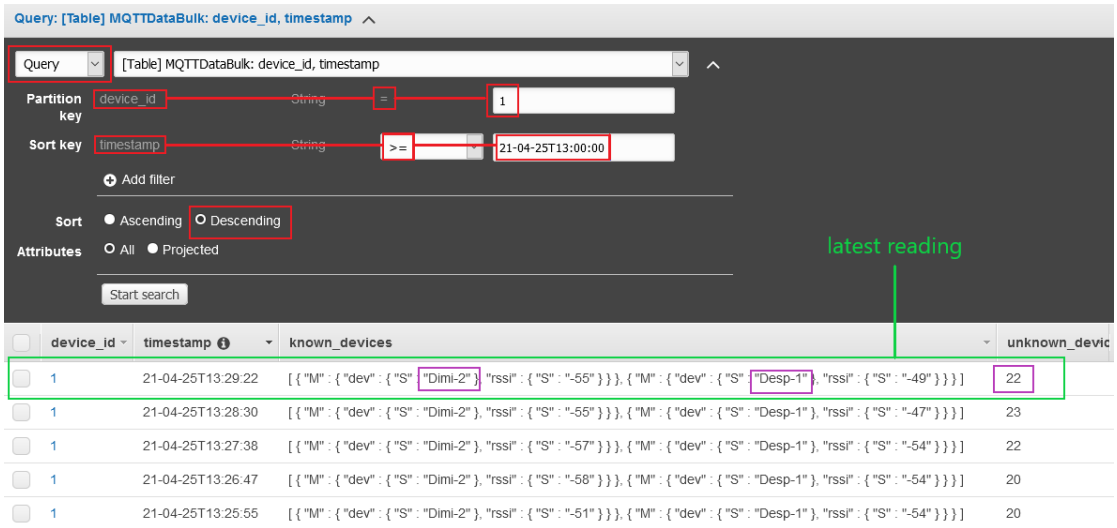

Figure 6.13: MQTTDataBulk query - Latest 5 readings (Type-A device ID=1)

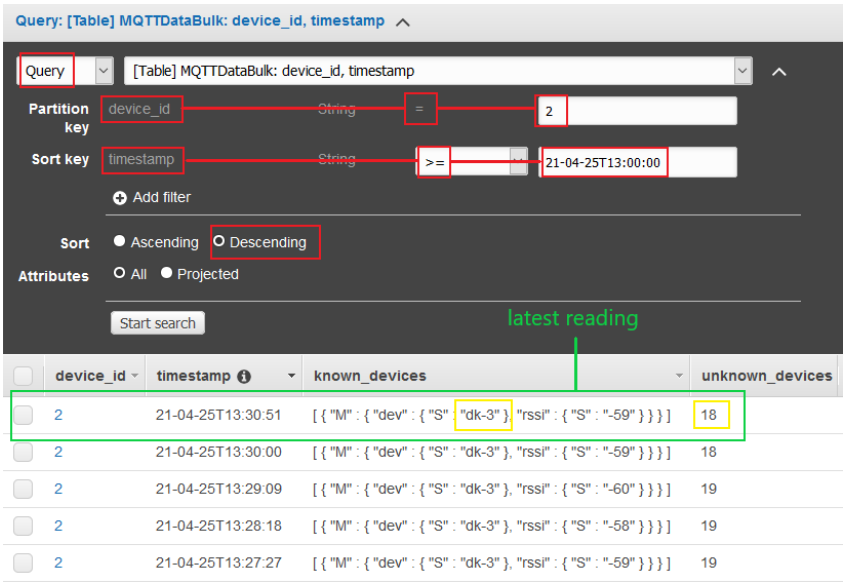

Figure 6.14: MQTTDataBulk query - Latest 5 readings (Type-A device ID=2)

The values highlighted in purple and yellow are the ones that comprise the response of GET */aggregate/* depicted in Figure 6.12 for devices with ID=1 and ID=2 in direct correlation.

The final data processing method refers to the distance calculation of a specified device in relation to three surrounding Type-A modules, in cases where the latter are deployed closely to each other thus have their Wi-Fi ranges overlapping. The method is triggered by the */location* GET method analyzed in the previous chapter. Sending a request against this endpoint returns the response shown in Figure 6.15.

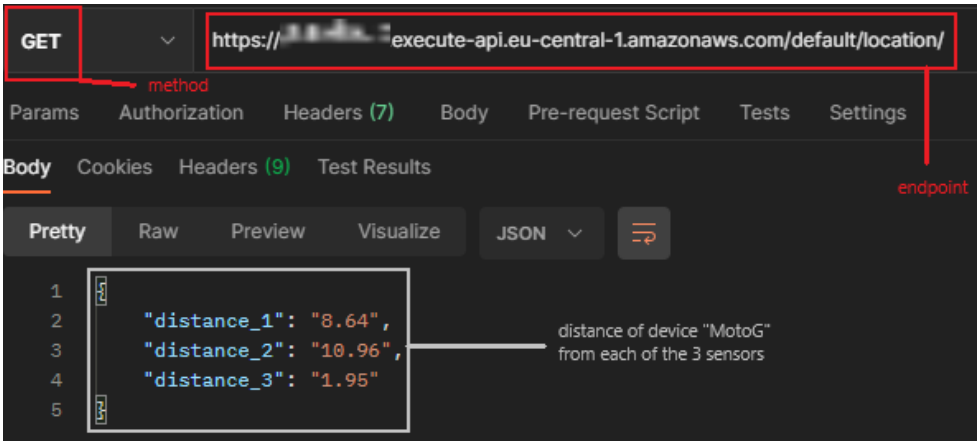

Figure 6.15: GET /location/ sample request & response

Performing the request returns 3 key-value pairs which refer to the distance of a specified device to each of the 3 surrounding Type-A modules that have detected it. In the showcased test case, a query is triggered upon the *MQTTDataBulkM2* table which is a snapshot of the original *MQTTDataBulk* used on all previous processes. The reason for using a separate table is to maintain RSSI values measured also at real-world distances to compare against the calculated values. At the time of testing the test case, the API endpoint and relevant Lambda Function support the calculation of the distances for a pre-defined device named *MotoG*. The relevant device readings originating from the 3 different Type-A sensors are stored in a manner as shown in Figure 6.16.

Figure 6.16: Sample MQTTDataBulkM2 data

As mentioned in the previous chapter, the trilateratePosition Lambda function is triggered upon performing a request against the mentioned endpoint, which leads to rssi-based distance calculation according to the 10 latest sensor readings in which the *MotoG* device was present.

6.5 *Data visualization*

The application's final layer refers to the use of the developed front-end client through which the APIs will be invoked to depict useful information to the first responder teams that will be using the platform. The first step towards using the client is the creation of a user, which involves the steps shown in Figure 6.17, Figure 6.18 and Figure 6.19.

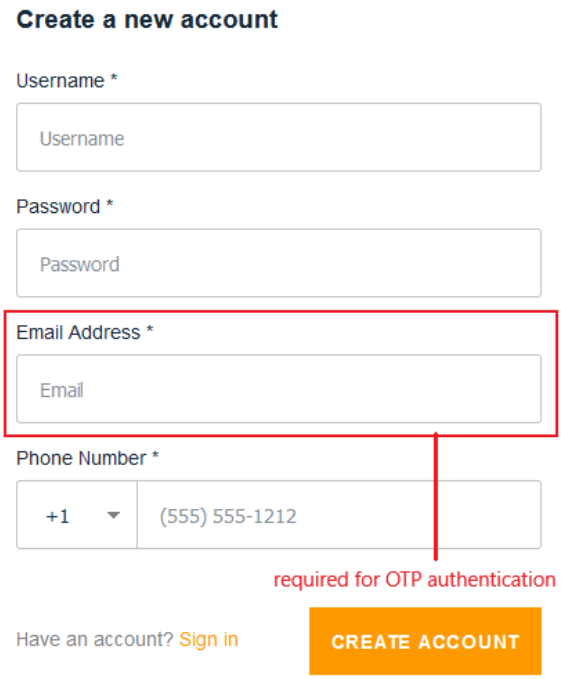

Figure 6.17: User registration

The process requires setting a username and password, while specifying a phone number and personal email address. The latter is mandatory so that authentication is carried out using a one-time password (OTP). Once all required information are provided, the OTP is received at the supplied email address and the client awaits its insertion to the relevant field in order for the registration process to finish, as shown in Figures 30 and 31.

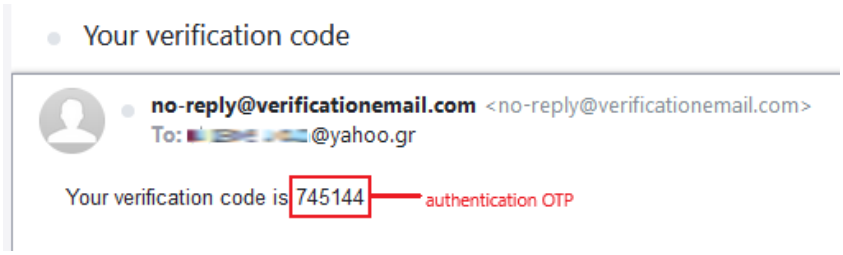

Figure 6.18: One-time password (OTP) authentication

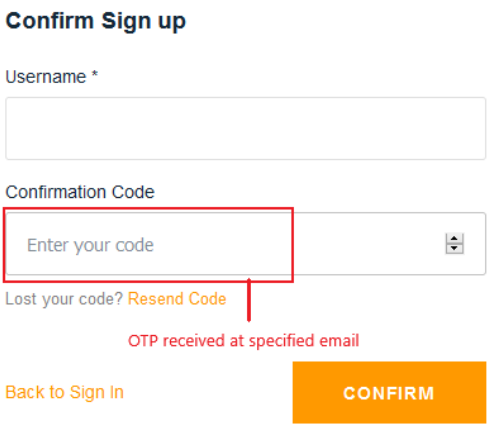

Figure 6.19: Registration confirmation

Successful registration adds the newly created user into the AWS Cognito User Pool specified during the configuration of AWS Amplify. Adding a number of multiple users' credentials results in a user group as depicted in Figure 6.20.

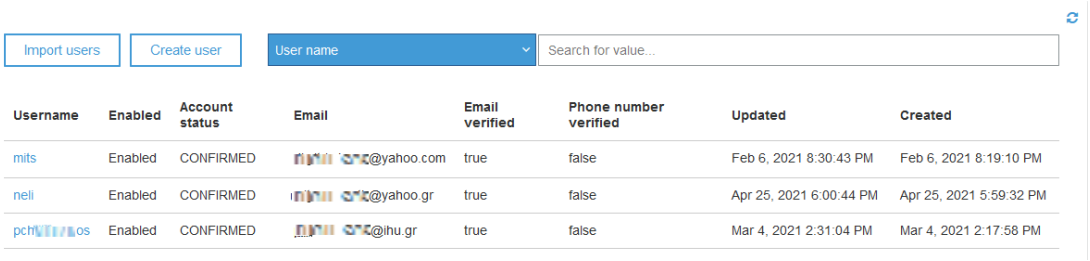

Figure 6.20: Sample registered users

The Account status field refers to whether or not the user has achieved authentication using the OTP method. The status can be managed by the administrator in order to specify different statuses as well, such as PASSWORD_RESET which will require the user to reset their password if needed to.

Once the user registers, they are automatically logged into the client. The application provides a persistent navigation bar which points to three different pages, each invoking the equivalent number of API endpoints and visualizing information accordingly. The pages are:

- Latest sensor readings
- Map view
- Indoor positioning

For the duration of fetching each API response, a message is shown to the front-end client to notify the user of the ongoing data exchange, as shown in Figure 6.21.

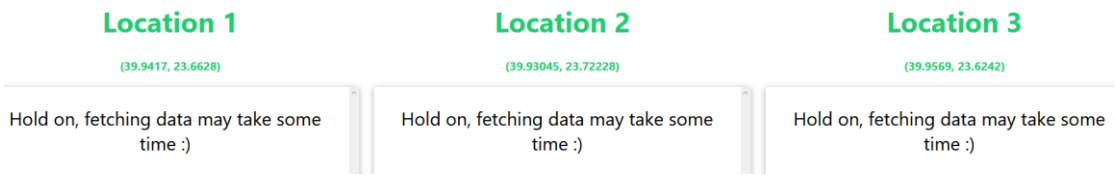

Figure 6.21: Fetch process (ongoing)

The latest sensor readings page makes use of the */device/{id]* GET method and shows the latest readings that each of the three deployed Type-A modules acquired in the location they are deployed in. Once the relevant API endpoint returns its response, each column is populated with the readings that were transmitted by each module to the centralized node, published to the topic and subsequently stored in the provisioned DynamoDB table. Each measurement block contains all the information stored in the mentioned table, as shown in Figure 6.22.

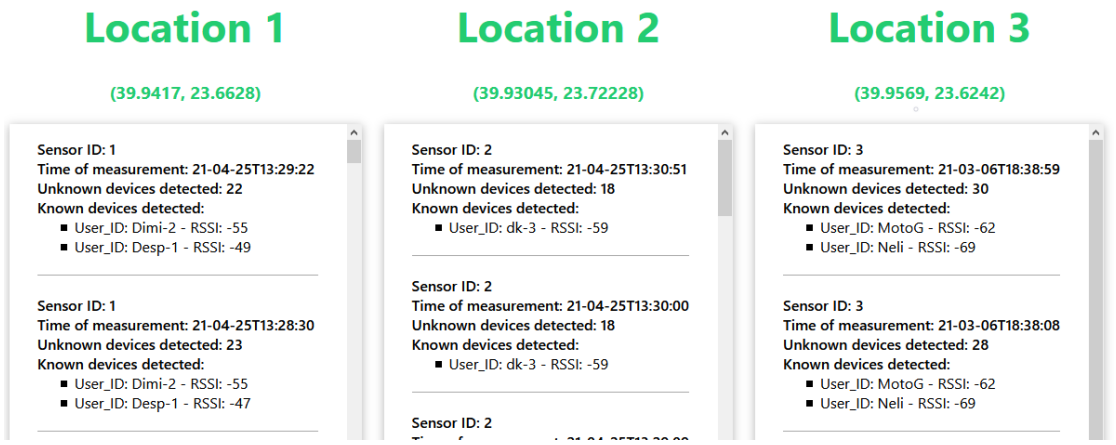

Figure 6.22: Detailed sensor data

The Map view page invokes the */aggregate/* GET method to populate the configured map with the latest reading each Type-A module has retrieved. The map generated for the test case is shown in Figure 6.23 and simulates the deployment of three Type-A modules in a 6km radius around a centrally placed Type-B in the Paliouri, Central Macedonia region. Each sensing module is assumed to be placed in non-adjacent regions and its depiction on the map allows for the display of the latest sensor reading upon clicking its marker. The values are based on the response that the API endpoint will return as demonstrated in the previous subsection's Figure 6.12.

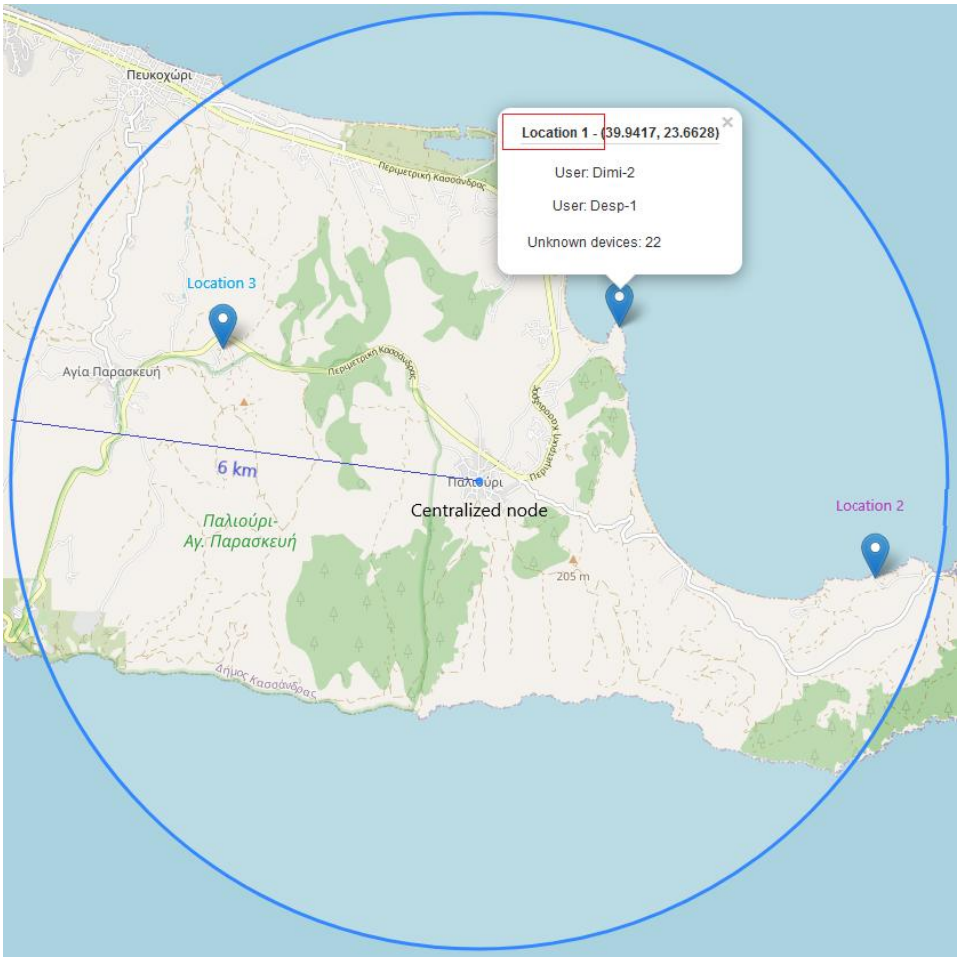

Figure 6.23: Map view – Latest sensor readings in surveyed area

The circle signifies the 6km LoRa range which shows that ultimately a large area of approximately 50 to 113 square kilometers can be covered by the currently defined architecture.

The Indoor positioning page refers to the previously analyzed RSSI-based distance calculation. The test case depicts the calculation performed when invoking the GET /location/ method as depicted in Figure 6.15, in human-readable form so that the information is deemed helpful for the first responders that will be using the platform. The end result is visible in Figure 6.24.

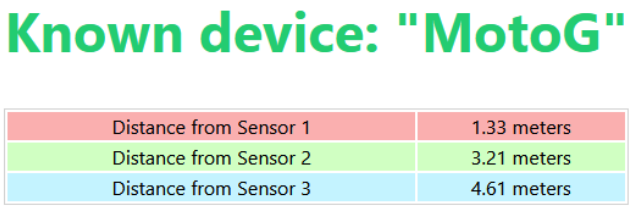

Figure 6.24: Distance calculation view – Distance to 3 adjacent sensors

7

Evaluation & future work

The purpose of the test case is to investigate the applicability of IoT technologies in the proposed use case and architecture, such as low-power ESP32 microcontrollers, LoRa, MQTT and Cloud / Serverless technologies. The end goal is to allow locating trapped people after severe natural phenomena or disasters and close-to-real-time feedback on their whereabouts. The application is based on passively detecting human presence in individual buildings through autonomous sensors and transmitting readings over very long distances, with minimal power or communication infrastructure needs. Scalability is a major concern as data can have increased or decreased volumes at any given moment, while the platform needs to also provide the capability of upscaling, depending on the number of surveyed buildings. Ultimately the platform is intended for providing detailed views of the latest measurements from each installed sensor, projections of the exact location of each sensor on a map and distance calculation between a device and 3 adjacent sensors. In this chapter an analysis is performed to evaluate whether or not these requirements have been met.

7.1 Evaluation

Testing the platform reveals useful information with regard to the generated results and their correlation to the use case the application is intended for. The first of the advantages demonstrated is the speed at which the information is relayed from each of the Type-A sensing modules to the Cloud domain and subsequently the front-end client. This proves the capability for close-to-real-time monitoring of the surveyed area. Additionally, MQTT-based communication caters for the possibility of high data traffic present while relaying measurements to the Cloud, meaning that no information can be lost at this level and that the data is readily available to any subscriber, should an upscaling need arise in the future.

At the Cloud domain, the information is stored in a non-sequential database, which allows for flexibility in case new or unexpected data is transmitted from the nodes. Scalability is addressed at this point as well, as any information published in the MQTT topic will be stored in the provisioned DynamoDB table disregarding the amount of messages and can be subsequently handled by any means necessary without the need to provide a different data

storage solution or re-structure the existing one. Additionally, the data is impossible to be lost or destroyed, as the DynamoDB can be configured to keep automatic backups or archive the data. Meanwhile there is virtually no possibility of infrastructural failure or physical theft since that level is handled by the Cloud provider. These characteristics further bolster the robustness and security aspects of the design.

Data processing takes a minimal amount of time which addresses the close-to-real-time prerequisite of the architecture. Testing data requests via the front-end client shows that the data is shown instantly. This is a result of the high performance that all the related Cloud-based components have, with the most prominently being the Lambda functions service since it is the most resource-intensive. The processes undertaken by the service would otherwise require long execution times, especially in case they were called to be executed concurrently on an on-premise platform. The performance of each of the 3 developed functions has been measured via AWS-inherent monitoring metrics, as depicted in Table 1 below.

Lambda function	**Mean execution time**	**Max memory used**
getDeviceData	139 ms	< 100 MB
aggregateData	211 ms	< 100 MB
trilateratePosition	298ms	< 100 MB

Table 1: Lambda functions execution times

The low-cost requirement has also been fulfilled; each ESP32 sensor cost less than 13€ to procure at a retail non-bulk price, while the power bank used to test the sustainability of the modules cost 16€. This means that the cost to deploy a Type-A module on each building would amount to less than 30€ should no additional equipment be used, such as an extended LoRa antenna. Additionally, the Cloud services included are free of charge during the first year of use, thus the platform can be tested for free prior to a final production-level deployment. Nevertheless these services would have minimal cost for the implemented platform after the trial period, as can be seen in Table 2.

Service	**Eligible Tier**
AWS IoT Core	Device connection : $0.08 per million minutes MQTT messaging **:** $1.00 (per million messages), up to 1 billion messages

Amazon DynamoDB	First 25 GB stored per month free - $0.306 per GB-month thereafter $1.525 per million write request units $0.305 per million read request units
AWS Lambda	$0.0000000021 per 1ms of execution time for 128MB of provisioned memory
Amazon API Gateway	$3.50 for the first 333 million requests

Table 2: Per-service costs

A rough cost estimation can be carried out based on the data shown in Tables 1 and 2, assuming that a production-level deployment would involve:

- A surveyed area of 100 residences, and a deployment of 100 Type A + 1 Type B devices transmitting data 24/7
- A post-disaster relief scenario involving 100 rescuers using the platform 24/7 for 2 weeks.

Final costs are estimated by taking into account the most significant costs incurred by the services, assuming that in the mentioned scenario there are 2 distinct operational states:

- Normal monitoring operation, where no use of the platform is required or performed
- Post-disaster operation, during which the platform will be fully utilized by the rescuers for the mentioned 2-week duration.

The total cost in both cases is extrapolated to a 30-day period as shown in Table 3.

Service	**Normal operation**	**Post-disaster operation**
AWS IoT Core	$4.67	$4.67
Amazon DynamoDB	$5.54	$5.94
AWS Lambda	$0	$3.29
Amazon API Gateway	$0	$17.50
Total cost / month	**$10.21**	**$31.4**

Table 3: Total cost per month

The figures show that normal surveillance operation would require an upkeep of roughly $10 per month, unless a disaster event occurs which would require an escalation in database read requests and API/Lambda calls.

Ease of use has been met by developing a simple user interface that follows commonly established front-end norms, through the use of a static navigation bar and dynamically loaded content handled by the React router, as described in the previous chapter. This makes the application easily accessible to first responders with minimal technical know-how and eliminates the need for platform-specific training. Additionally, using React.js as the framework of choice renders the front-end client readily available to any Internet browser-capable device ranging from desktop computers to smartphones and tablets. This eliminates the need to provide costly use-specific devices and allows the platform to be accessible via a large variety of devices at any given moment.

The security aspect has been addressed via the used of key-value and pairs and authentication certificates in the communication between the centralized node and the AWS domain. All data stored and processed within the Cloud is inherently secure, while the final internet-facing application can be used only by authorized users whose credentials can be managed via the Amazon Cognito service by authorized administrators.

On the other hand, one of the drawbacks presented by the current implementation refers to the use of a single centralized node to handle relaying measurements from the sensors to the Cloud. This characteristic introduces a single-point of failure in the architecture which may well occur in situations where the Type B node is destroyed or unable communicate to the Cloud provider due to the 802.11 infrastructure failure. This could be alleviated by providing a secondary backup connection to the node, such as a satellite connection, or by adding a failover node placed at a different location to handle communication in case the original node is offline. The latter can be implemented with little overhead, as the LoRa packets are transmitted in a broadcast manner. However precaution should be taken so that all Type A nodes are able to reach the fallback Type B module as well.

In addition to that, LoRa communications has been proven to transmit data reliably on a 4-6 km range [14], [24] which is demonstrated in the test case as well. However the case requires that all deployed LoRa antennas share a line of sight in order for those distance coverages to be achieved. Failure to provide this, will result in a significant reduction of the coverage to as low as 1 km [14].

Finally, the device positioning feature shows that using the set of calculations and parameters described in previous sections yields low accuracy results at higher distances as shown in Table 4.

Actual distance	Measured distance
1.5 meters	1.56 meters
3.7 meters	3.21 meters
6.5 meters (isolated sensor)	4.61 meters

Table 4: Correlation of calculated and actual distances

The calculated distances demonstrate increasing discrepancies relative to increasing real distance and obstructing structures in between the sensors and the detected device. Even though the distance calculation at 1.5m is accurate, the calculated value of 4.61 meters is 1.89 meters less than the actual 6.5 measurement, due to the fact that the sensor was deployed further away and behind walls. This indicates that the values may be even more inaccurate at greater distances and under collapsed structures or debris in the event of an earthquake, which could be partially alleviated by specifying a different n variable per sensor depending on the deployment area, as the RSSI measurements are sensitive and directly affected by the surrounding environment. Additionally it can be addressed by other means described in the following subsection of this chapter.

Ultimately, the application serves the use case it was intended for and specifically fulfills the following prerequisites:

- Autonomous measurement procurement and data transmission
- Robust asynchronous communication of data to the Cloud
- Secure and reliable data aggregation and storage
- Fast and user-friendly depiction of important information to rescue teams
- Authorized users application access

7.2 Future work

7.2.1 Energy optimization

As mentioned in chapter 5, each of the deployed sensing ESP32 microcontrollers enter a wait state after each frequency-hopping iteration has elapsed, in order for energy consumption to be kept to a minimum. For the current version of the platform this has been addressed via the use of a timer that introduces a 35 second delay in between each iteration. The energy savings presented in this manner are relative to the microcontroller being idle during this period of time. However, additional energy gains can be achieved by utilizing the inherent sleep modes that ESP32 provide. As stated in [18], energy consumption can be reduced from 200mAh to

0.8mAh upon switching operation from active to sleep mode respectively. This is translated to a 99.6% energy consumption reduction and can greatly prolong the modules' operation on battery power should, if deemed necessary.

7.2.2 Over the air (OTA) whitelist management

As demonstrated in the testing phase, all known devices information is currently inserted manually into each of the deployed Type-A modules' firmware. The latter holds an array that associates the MAC address of each resident's device of choice to a specified designation/nickname. In a real-world scenario managing this information would be very difficult in this manner; any changes required with regard to new or altered resident/device information would have to be performed via a manual connection to each of the affected Type A modules. A future implementation could make use of a third-party mobile application to set or alter this information directly on the Type-A module in range. Another approach could make use of the already established LoRa and MQTT-based communications in a bi-directional way, so that the front-end client manipulates the information and the information traverses the Cloud domain down to the firmware level.

7.2.3 Secondary readings

Resident sensing is currently based upon the detection of Wi-Fi-enabled devices within range of each residentially deployed Type A sensor. The accuracy of the metrics in question could however be further enhanced by implementing a secondary detection phase of Bluetooth Low Energy (BLE) devices, which mainly refers to wearables such as smartwatches. The metrics procured could then be correlated with the already available known/unknown devices' information to further deduce the presence of additional people within the monitored building. Other than BLE device detection, additional readings could also be retrieved via the use of extra sensors connected to each Type-A module, such as proximity, temperature, pressure and humidity sensors. This could allow for more robust detection logic implementations, based on sets of multiple procured parameters.

7.2.4 Emergency messages/notifications

The utilization of Cloud services provides a range of additional capabilities that can be very useful towards rescue, the most useful being the use of targeted notifications triggered by additional sensors in the event of a natural disaster. In this way email, SMS or other notifications could be sent to anyone involved in rescue operations when disaster strikes, with

the end goal of narrowing the gap between the event and the start of operations. This could be translated in human lives, as time is of the utmost importance when it comes to these scenarios. Moreover, notifications could be triggered by other events such as a microcontroller running out of power or ceasing operation, which could minimize the time that a building could remain unmonitored.

7.2.5 Indoor positioning enhancement

A range of techniques can also be used in calculating the distance of devices to their sensing modules. Trilateration and fingerprinting have been the norm in estimating a device's location within buildings based on signal strength values, as described in [15], [35], [37], [42]–[45]. These techniques can be further enhanced by the used of machine learning techniques such as kNN [42], [46], [47], while there are AWS-based services such as Amazon SageMaker that are intended specifically for implementing ML-based solutions, thus making collaboration with the already used services very efficient.

References

[1] U. Zafar, M. A. Shah, A. Wahid, A. Akhunzada, and S. Arif, “Exploring iot applications for disaster management: Identifying key factors and proposing future directions,” *EAI/Springer Innov. Commun. Comput.*, pp. 291–309, 2019.

[2] P. M. Jacob and P. Mani, “A reference model for testing internet of things based applications,” *J. Eng. Sci. Technol.*, vol. 13, no. 8, pp. 2504–2519, 2018.

[3] B. E. El-Shweky *et al.*, “Internet of things: A comparative study,” in *2018 IEEE 8th Annual Computing and Communication Workshop and Conference (CCWC)*, 2018, pp. 622–631.

[4] J. W. Kim, S. H. Sul, and J. B. Choi, “Development of unmanned remote smart rescue platform applying Internet of Things technology,” *Int. J. Distrib. Sens. Networks*, vol. 14, no. 6, 2018.

[5] P. P. Ray, M. Mukherjee, and L. Shu, “Internet of Things for Disaster Management: State-of-the-Art and Prospects,” *IEEE Access*, vol. 5, no. i, pp. 18818–18835, 2017.

[6] M. Rosyidi, R. H. Puspita, S. Kashihara, D. Fall, and K. Ikeda, “A Design of IoT-Based Searching System for Displaying Victim’s Presence Area,” *Proc. - Int. Comput. Softw. Appl. Conf.*, vol. 2, pp. 8–13, 2018.

[7] L. Oliveira, J. Henrique, D. Schneider, J. De Souza, S. Rodriques, and W. Sherr, “Sherlock: Capturing Probe Requests for Automatic Presence Detection,” *Proc. 2018 IEEE 22nd Int. Conf. Comput. Support. Coop. Work Des. CSCWD 2018*, pp. 178–183, 2018.

[8] L. Oliveira, D. Schneider, J. De Souza, and W. Shen, “Mobile Device Detection through WiFi Probe Request Analysis,” *IEEE Access*, vol. 7, pp. 98579–98588, 2019.

[9] E. Vattapparamban, B. S. Çiftler, I. Güvenç, K. Akkaya, and A. Kadri, “Indoor occupancy tracking in smart buildings using passive sniffing of probe requests,” *2016 IEEE Int. Conf. Commun. Work. ICC 2016*, pp. 38–44, 2016.

[10] M. Karakaya, G. Şengül, and E. Gökçay, “An IoT application for locating victims aftermath of an earthquake,” *2017 IEEE 1st Ukr. Conf. Electr. Comput. Eng. UKRCON 2017 - Proc.*, pp. 876–880, 2017.

[11] V. Babu and V. Rajan, “Flood and Earthquake Detection and Rescue Using IoT Technology,” *Proc. 4th Int. Conf. Commun. Electron. Syst. ICCES 2019*, no. Icces, pp. 1256–1260, 2019.

[12] P. Boccadoro, B. Montaruli, and L. A. Grieco, “QuakeSense, a LoRa-compliant

Earthquake Monitoring Open System," *Proc. - 2019 IEEE/ACM 23rd Int. Symp. Distrib. Simul. Real Time Appl. DS-RT 2019*, pp. 1–8, 2019.

[13] R. K. Kodali, S. C. Rajanarayanan, A. Koganti, and L. Boppana, "IoT based security system," *IEEE Reg. 10 Annu. Int. Conf. Proceedings/TENCON*, vol. 2019-Octob, pp. 1253–1257, 2019.

[14] J. Höchst, L. Baumgärtner, F. Kuntke, A. Penning, A. Sterz, and B. Freisleben, "LoRa-based Device-to-Device Smartphone Communication for Crisis Scenarios," *17th Int. Conf. Inf. Syst. Cris. Response Manag.*, no. May, pp. 996–1011, 2020.

[15] M. N. Amr, H. M. El Attar, M. H. Abd El Azeem, and H. El Badawy, "An enhanced indoor positioning technique based on a novel received signal strength indicator distance prediction and correction model," *Sensors (Switzerland)*, vol. 21, no. 3, pp. 1–26, 2021.

[16] L. Sciullo, A. Trotta, and M. Di Felice, "Design and performance evaluation of a LoRa-based mobile emergency management system (LOCATE)," *Ad Hoc Networks*, vol. 96, 2020.

[17] R. K. Kodali and S. Yerroju, "IoT based smart emergency response system for fire hazards," *Proc. 2017 3rd Int. Conf. Appl. Theor. Comput. Commun. Technol. iCATccT 2017*, pp. 194–199, 2018.

[18] E. Gatial, Z. Balogh, and L. Hluchy, "Concept of Energy Efficient ESP32 Chip for Industrial Wireless Sensor Network," *INES 2020 - IEEE 24th Int. Conf. Intell. Eng. Syst. Proc.*, pp. 179–183, 2020.

[19] EspressifSystems, "ESP32 Series Datasheet," 2021.

[20] EspressifSystems, "ESP32-PICO-D4 Datasheet." [Online]. Available: moz-extension://512cafab-e4f3-405c-af32-c2c7d1e1ca8f/enhanced-reader.html?openApp&pdf=https%3A%2F%2Fwww.tme.eu%2FDocument%2F50217a87a59b4fe9179a85fbf8331ac0%2Fesp32-pico-d4_datasheet_en.pdf. [Accessed: 29-Mar-2021].

[21] lilygo.cn, "LILYGO® TTGO T-Display ESP32 WiFi and Bluetooth Module Development Board For Arduino 1.14 Inch LCD - ESP32 Module - Shenzhen Xin Yuan Electronic Technology Co., Ltd," 2020. [Online]. Available: http://www.lilygo.cn/prod_view.aspx?TypeId=50044&Id=1126&FId=t3:50044:3. [Accessed: 19-Jun-2021].

[22] J. P. Shanmuga Sundaram, W. Du, and Z. Zhao, "A Survey on LoRa Networking: Research Problems, Current Solutions, and Open Issues," *IEEE Commun. Surv. Tutorials*, vol. 22, no. 1, pp. 371–388, 2020.

[23] Semtech, “LoRa and LoRaWAN: Technical overview,” no. December 2019, pp. 1–26, 2019.

[24] R. Madoune Seye, B. Ngom, B. Gueye, and M. Diallo, “A study of LoRa coverage: Range evaluation and channel attenuation model,” *ICSCC 2018 - 1st Int. Conf. Smart Cities Communities*, pp. 1–4, 2018.

[25] Semtech, “Why LoRa? | Semtech LoRa Technology | Semtech,” 2021. [Online]. Available: https://www.semtech.com/lora/why-lora. [Accessed: 19-Jun-2021].

[26] B. Mishra and A. Kertesz, “The Use of MQTT in M2M and IoT Systems: A Survey,” *IEEE Access*, vol. 8, no. November, pp. 201071–201086, 2020.

[27] F. Thiel, “MQTT: Communication in the Internet of Things,” 2020. [Online]. Available: https://www.wut.de/e-577ww-05-apus-000.php. [Accessed: 19-Jun-2021].

[28] “MQTT Version 5.0,” *Oasis-Open*, no. March, p. 137, 2019.

[29] D. Bastos, “Cloud for IoT - A survey of technologies and security features of public cloud IoT solutions,” *IET Conf. Publ.*, vol. 2019, no. CP756, pp. 1–2, 2019.

[30] S. Mathew, “Overview of Amazon Web Services - AWS Whitepaper,” no. January, pp. 1–30, 2021.

[31] “Three-year-old girl rescued 91 hours after Turkey quake,” *Al Jazeera*, 2020.

[32] M. F. Khan, G. Wang, M. Z. A. Bhuiyan, and X. Li, “Wi-fi signal coverage distance estimation in collapsed structures,” *Proc. - 15th IEEE Int. Symp. Parallel Distrib. Process. with Appl. 16th IEEE Int. Conf. Ubiquitous Comput. Commun. ISPA/IUCC 2017*, pp. 1066–1073, 2018.

[33] T. Wei and S. Bell, “Indoor localization method comparison: Fingerprinting and Trilateration algorithm,” *Rose.Geog.Mcgill.Ca*, 2006.

[34] P. Cotera, M. Velazquez, D. Cruz, L. Medina, and M. Bandala, “Indoor Robot Positioning Using an Enhanced Trilateration Algorithm,” *Int. J. Adv. Robot. Syst.*, vol. 13, no. 3, 2016.

[35] M. E. Rusli, M. Ali, N. Jamil, and M. M. Din, “An Improved Indoor Positioning Algorithm Based on RSSI-Trilateration Technique for Internet of Things (IOT),” *Proc. - 6th Int. Conf. Comput. Commun. Eng. Innov. Technol. to Serve Humanit. ICCCE 2016*, pp. 12–77, 2016.

[36] E. Mok and G. Retscher, “Location determination using wifi fingerprinting versus wifi trilateration,” *J. Locat. Based Serv.*, vol. 1, no. 2, pp. 145–159, 2007.

[37] S. Chan and G. Sohn, “Indoor Localization Using Wi-Fi Based Fingerprinting and Trilateration Techiques for Lbs Applications,” *ISPRS - Int. Arch. Photogramm.*

Remote Sens. Spat. Inf. Sci., vol. XXXVIII-4/, pp. 1–5, 2012.

[38] Z. Xu *et al.*, "Pedestrain Monitoring System using Wi-Fi Technology And RSSI Based Localization," *Int. J. Wirel. Mob. Networks*, vol. 5, no. 4, pp. 17–34, 2013.

[39] G. Callebaut, G. Leenders, C. Buyle, S. Crul, and L. van der Perre, "LoRa physical layer evaluation for point-to-point links and coverage measurements in diverse environments," *arXiv*, no. October, 2019.

[40] T. Khan, S. Ghosh, M. Iqbal, G. Ubakanma, and T. Dagiuklas, "RESCUE: A Resilient Cloud Based IoT System for Emergency and Disaster Recovery," *Proc. - 20th Int. Conf. High Perform. Comput. Commun. 16th Int. Conf. Smart City 4th Int. Conf. Data Sci. Syst. HPCC/SmartCity/DSS 2018*, pp. 1043–1047, 2019.

[41] "What Is Amazon DynamoDB? - Amazon DynamoDB." [Online]. Available: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html. [Accessed: 11-Apr-2021].

[42] A. E. M. El Ashry and B. I. Sheta, "Wi-Fi based indoor localization using trilateration and fingerprinting methods," *IOP Conf. Ser. Mater. Sci. Eng.*, vol. 610, no. 1, pp. 0–20, 2019.

[43] S. Gao and S. Prasad, "Employing spatial analysis in indoor positioning and tracking using Wi-Fi access points," *Proc. 8th ACM SIGSPATIAL Int. Work. Indoor Spat. Awareness, ISA 2016*, pp. 27–34, 2016.

[44] S. Sadowski and P. Spachos, "RSSI-Based Indoor Localization with the Internet of Things," *IEEE Access*, vol. 6, pp. 30149–30161, 2018.

[45] S. Subedi and J. Y. Pyun, "Practical Fingerprinting Localization for Indoor Positioning System by Using Beacons," *J. Sensors*, vol. 2017, 2017.

[46] J. P. Grisales Campeon, S. Lopez, S. R. De Jesus Melean, H. Moldovan, D. R. Parisi, and P. I. Fierens, "Indoor Positioning based on RSSI of WiFi signals: How accurate can it be?," *2018 IEEE Bienn. Congr. Argentina, ARGENCON 2018*, 2019.

[47] H. Chenji, W. Zhang, R. Stoleru, and C. Arnett, "DistressNet: A disaster response system providing constant availability cloud-like services," *Ad Hoc Networks*, vol. 11, no. 8, pp. 2440–2460, 2013.