TÉCNICO LISBOA

Crowdnet: crowd-powered network

Nuno Miguel Ribeiro Silva

Dissertation to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisor: Prof. Alberto Manuel Ramos da Cunha

Examination Committee

Chairperson: Prof. João António Madeiras Pereira

Supervisor: Prof. Alberto Manuel Ramos da Cunha

Member of the Committee: Prof. Renato Jorge Caleira Nunes

October 2020

Resumo

A plataforma Arduino tem ganho popularidade, sendo usada no ensino de temas relacionados com micro-controladores e sistemas embebidos em ambientes laboratoriais. Para interligar várias placas Arduino pode usar-se uma ligação com fios, como o barramento I^2C. No entanto, estas ligações necessitam que sejam passados fios entre cada placa. Isto complica a montagem e apenas permite que sejam ligadas placas na mesma sala. Além disso, devido à recente pandemia de COVID-19, escolas e universidades têm-se focado em permitir que os estudantes consigam trabalhar remotamente a partir das suas casas. Neste caso, uma simples ligação sem fios não seria suficiente para permitir comunicações entre placas em salas ou casas diferentes, pois complicaria a interface oferecida aos utilizadores, entre outros problemas. A Crowdnet pretende, portanto, oferecer um *middleware* para comunicação entre várias placas Arduino, suportando comunicação dentro da mesma sala ou em salas diferentes através da Internet, e usando uma interface de programação semelhante ao I^2C. Consiste numa biblioteca Arduino que se liga por BLE a uma aplicação Android. Por sua vez, a aplicação liga-se a um agente MQTT pela Internet. Cria-se assim uma rede lógica que permite que várias placas Arduino troquem mensagens entre si. A avaliação da Crowdnet mostrou que é possível atingir velocidades até 4.4 kB/s e latências entre 25–136 milissegundos, ainda que os resultados obtidos variem dependendo do telemóvel utilizado. Porém, os resultados obtidos são adequados para projetos laboratoriais simples, sendo que a utilização de MQTT em conjunto com BLE oferece melhores velocidades do que apenas com BLE.

Palavras-chave: Meio de comunicação, Biblioteca Arduino, Bluetooth Low Energy, Telemóvel Android, MQTT, I2C

Abstract

The Arduino platform has gained popularity and is used to teach students about microcontrollers and embedded systems in laboratory environments. To connect various Arduino boards together, a wired connection, such the I^2C bus, can be used. However, wired connections require running wires between each device, complicating the setup of many devices, and only allow connecting devices in the same room. Moreover, due to the recent COVID-19 pandemic, allowing students to work remotely and out of a laboratory is becoming an important consideration for schools and universities. In this case, a simple wireless interface would also not be sufficient to allow communication between devices in different rooms or homes, as it would offer a complicated interface, among other issues. Crowdnet intends to provide a middleware for communication between Arduino boards, supporting communications both in the same room and in different rooms over the Internet, while offering a simple I^2C-like programming interface. It consists on an Arduino library that connects to an Android application via BLE. In turn, the smartphone connects to an MQTT Broker via the Internet. This creates a logical network that allows sketches running on various Arduino boards to exchange messages with each other. Evaluation showed that Crowdnet is able to achieve reasonable throughput speeds up to 4.4 kB/s and latencies of 25–136 milliseconds, even though results varied depending on the smartphones that were tested. Nonetheless, the achieved results are adequate for simple laboratory projects, and using MQTT in conjunction with BLE provides greater throughput than just BLE.

Keywords: Communication middleware, Arduino library, Bluetooth Low Energy, Android smartphone, MQTT, I2C

Contents

List of Figures

List of Tables

Acronyms

ACIC	Applications and Computation for the Internet of Things
AI	Ambient Intelligence
AP	Access Point
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
BLE	Bluetooth Low Energy
EUI	Extended Unique Identifier
FOSS	Free and Open-Source Software
GATT	Generic Attribute Profile
HTTP	Hypertext Transfer Protocol
I^2C	Inter-Integrated Circuit
IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronics Engineers
IC	Integrated Circuit
IoT	Internet of Things
IP	Internet Protocol
IST	Instituto Superior Técnico
LAN	Local Area Network
LPWAN	Low-Power Wide Area Network
LoRaWAN	Long Range Wide Area Network
MAC	Media Access Control
MEIC	Master of Science program in Computer Science and Engineering
MQTT	Message Queuing Telemetry Transport
MTU	Maximum Transmission Unit
OS	Operating System
QoS	Quality of Service
RNL	Rede das Novas Licenciaturas
SIG	Special Interest Group

SoC	System on Chip
SPI	Serial Peripheral Interface Bus
TCP/IP	Transmission Control Protocol over IP
TCP	Transmission Control Protocol
TWI	Two-Wire Interface
UART	Universal asynchronous receiver-transmitter
UI	User Interface
URL	Uniform Resource Locator
UUID	Universally Unique Identifier
VM	Virtual Machine
WLAN	Wireless Local Area Network
WPAN	Wireless Personal Area Network

1 Introduction

Nowadays, embedded devices are present everywhere. With the expansion of the Internet of Things (IoT) in recent years, this trend has only increased, and the number of embedded devices in use is now greater than ever. The Arduino platform has also gained popularity, not only because it makes it simple for beginners to start making projects using a microcontroller, but also because it is a great tool to teach students about microcontrollers and embedded systems in laboratory environments, serving as a starting point for developing IoT applications.

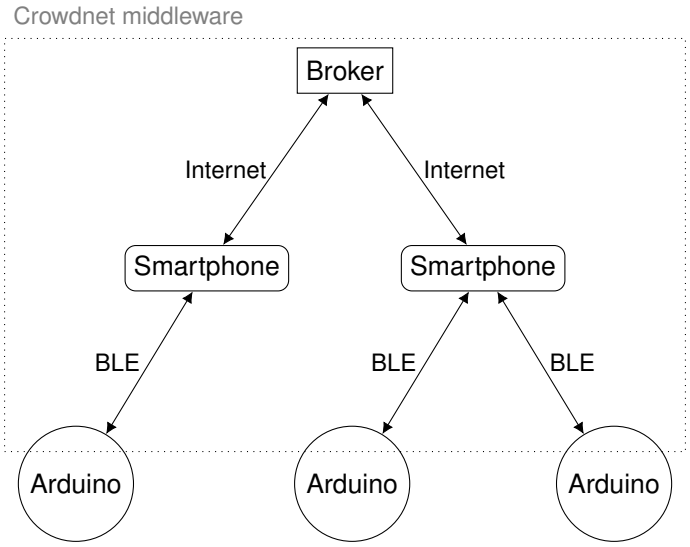
A fundamental part of IoT devices is communication. For devices located close to each other, one solution is to use a wired connection, such the Inter-Integrated Circuit (I^2C) bus, which is one of the easiest communication solutions already supported in the Arduino environment. However, wired connections require running wires between each device, which is particularly cumbersome and tedious when more than a couple devices need to be connected, and even more so in laboratory environments involving many students. A wireless network such Wi-Fi, Zigbee, Bluetooth, etc, could be used, but would offer a very different and more complicated interface than what beginners are used to, among other problems.

Due to the recent COVID-19 pandemic (Rahiem, 2020), allowing students to continue to work remotely and out of the laboratory is becoming an important consideration for schools and universities. Using a wired bus, students could still work at home, but would be unable to have their projects interact with each other, which is a fundamental part of the learning process. In this case, it is clear that a simple wireless interface would also not be sufficient to allow communication between devices in different homes, and a more elaborate solution is needed.

At the same time, most people own a smartphone and carry it with them during the day. Smartphones are especially popular among university students. Using smartphones in the Crowdnet system also means that students will be more engaged with the projects they need to develop, since the system can later be expanded to provide additional functionality in the smartphone. This is further discussed in Section 6.1.

Taking all these things into consideration, we have built a middleware that is able to use the communication interfaces of smartphones and use them as a means of communication between Arduino boards, as depicted in Figure 1.1. Instead of communicating directly, Arduino boards connect to a smartphone via Bluetooth Low Energy (BLE). Then, smartphones communicate with each other via the Internet to forward messages coming from the Arduino. This effectively allows Arduino boards to exchange arbitrary messages with each other indirectly using a wireless interface, while also offering a familiar I^2C-like programming interface that is not limited to a single room, allowing students to test and

Figure 1.1: Connections between components of the Crowdnet middleware



integrate their work with other students remotely. The context and motivation behind Crowdnet is further detailed in Section 1.1.

Crowdnet consists of three interconnected components: an Arduino library, an Android Application (App) and an MQTT Broker server. The Arduino library offers an I^2C-like programming interface and connects to a nearby smartphone via BLE. The smartphone runs a Crowdnet App that accepts connections from the nearby Arduino boards and forwards messages between them. Lastly, the server allows smartphones to exchange messages destined to remote Arduino boards.

1.1 Context and motivation

This Section provides some context for the work, explaining the environments where Crowdnet is intended to be used, as well as some of the motivation behind it.

The primary motivation of this work is for it to be used in laboratory classes at Instituto Superior Técnico (IST), though its applications are not limited to this. At IST, Arduino Uno boards have been used for teaching laboratory classes in multiple courses. In some courses of the Master of Science program in Computer Science and Engineering (MEIC), for example, students develop their projects using an Arduino Uno.

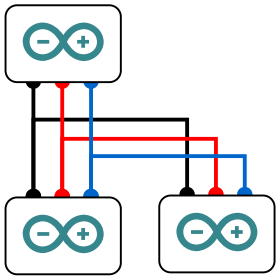
One of such courses is Applications and Computation for the Internet of Things (ACIC)[1]. In this course, during the laboratory classes, students are given an Arduino board to be used in their projects (Cunha, 2017). The focus of this course is on cyber-physical and embedded systems for the IoT. In the first few laboratory guides, students are introduced to the Arduino environment and interact with a few sensors and actuators. Then, students are taught how to connect multiple Arduino boards to each other using the wired I^2C bus. After these few introductory exercises, students are assigned a larger

[1] `https://fenix.tecnico.ulisboa.pt/disciplinas/ACPIC7/2020-2021/1-semestre`, accessed on 1st September 2020

project which consists on implementing an automatic traffic lights system, including some fault tolerance measures and fail-safe behavior. To do this, they must implement an application-specific protocol on top of the I^{2}C bus, which based on a specification used by all the students. The protocol is expected to be interoperable between each group of students and, therefore, students need to regularly test their projects with other groups of students.

In this context, the first limitation of the I^{2}C bus is that it requires three wires to be connected between each board — Ground, SDA, and SCL — as per Figure 1.2. When only two or three boards are

Figure 1.2: Three Arduino boards connected via I2C



involved, this is not too hard. However, when more boards need to be connected, it becomes difficult or impractical to connect so many wires between the boards. For example, in a laboratory with 10 groups of students, each with two Arduino boards, $10 \times 2 \times 3 = 60$ wires would need to be connected and checked, which quickly becomes impractical.

The second limitation is as follows: Students are allowed to take the Arduino board home during project development. However, since the exercises and project they need to complete focuses on communicating using the I^{2}C bus and integrating projects of different students together, students are not able to properly test their project at home or outside the laboratory. Moreover, students are limited by the tight schedule of the laboratory in order to test their projects. This problem is even more evident with the onset of the current the COVID-19 pandemic. Crowdnet would solve this problem by allowing Arduino Nodes to communicate with each other in the same room or remotely via the Internet, using the students' smartphones. The use of smartphones means that no additional hardware is required to work at home, besides an Internet connection. This would allow students to test their projects at any time, 24 hours per day, 7 days per week, without being restricted by the laboratory schedule and without having to leave their homes.

Another course where Crowdnet may be useful is in the Ambient Intelligence (AI) course [2]. This course explores concepts such as Home Automation, Smart Houses, Smart Cities, Automotive Intelligence, the IoT, and other related technologies using embedded systems. As part of the course, students develop a project of their choosing, which must be related to the general concepts discussed in the course. Many of the students choose to use the Arduino platform, and therefore could use the

[2] `https://fenix.tecnico.ulisboa.pt/disciplinas/AI514/2019-2020/2-semestre/`, accessed on 2nd September 2020

Crowdnet system to provide them with more advanced communication between various Arduino boards, for example in a Smart Home application.

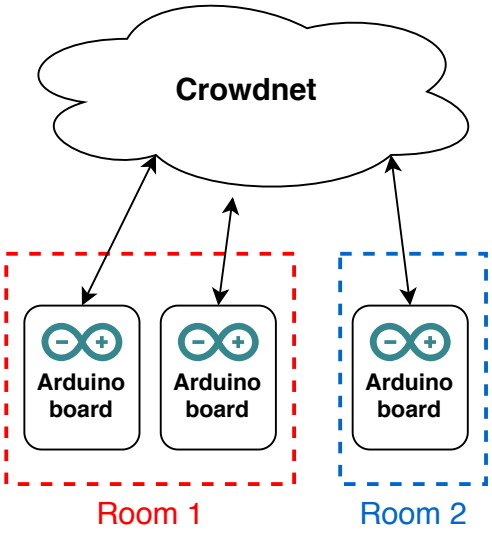
Since these courses already make use of the Arduino platform, Crowdnet was developed on this same platform. This makes the system simpler and easier for the students to use.

1.2 Requirements and Objectives

The objectives of Crowdnet are to create a logical network that:

- enables Arduino boards to exchange arbitrary messages between each other, as per Figure 1.3:
  - in the same room or in a laboratory environment;
  - in different rooms, buildings, or homes;
- offers a simple Application Programming Interface (API) to be used by Arduino sketches;
- is compatible with the Arduino Integrated Development Environment (IDE);
- uses the smartphones of the students to avoid needing additional hardware and to enable future expansion of the system.

Figure 1.3: High level view of the Crowdnet middleware



In terms of communication parameters, and having in mind the use cases for which Crowdnet is intended, the requirements of the system are:

- latencies in the order of a few hundred milliseconds, but less than one second;
- speeds in the order of a few kilobytes per second;

These requirements mean that Crowdnet may not be suitable for critical real time communications. In particular, since it is intended to be used by students in a laboratory environment, the flexibility of the

system is more important than the performance that could be achieved using a wired communication bus such as I^2C.

1.3 Layout of the document

This document is organized as follows: Chapter 2 describes a few communication methods that were considered. In Chapter 3, the architecture of the system is described. Then, Chapter 4 provides a more in depth description of how the system was implemented and how the various components communicate with each other. Chapter 5 describes how the system was evaluated and presents the results of the evaluation. Finally, Chapter 6 summarizes the results and presents a few suggestions for future improvements.

2 State of the Art

In this Chapter the state of the art is described by looking into some of the technologies that were considered during the development of Crowdnet. We start by exploring how Arduino boards could communicate with each other or with a smartphone, and then we look into how two or more smartphones can communicate via the Internet.

2.1 Arduino communications

This section presents some of the existing alternatives that can be used to communicate with an Arduino board. Some of these methods allow communications between various Arduino boards, while others allow communications between Arduino boards, smartphones, or even the Internet. However, since we already assume a smartphone will somehow be used for communications, in this section we are mainly concerned about communications between a smartphone and an Arduino board.

2.1.1 I2C bus

Inter-Integrated Circuit (I^2C), sometimes also called Two-Wire Interface (TWI), is a low-speed serial communication bus that is widely used to connect microcontrollers to peripheral Integrated Circuits (ICs) or to other microcontrollers that are within a maximum distance of a few meters (Leens, 2009).

The bus is made up of two lines — or wires, one for clock (SCL) and one for data (SDA). Devices using the bus can have one of two roles:

- Master, which initiates communication with slaves and generates the clock signal;
- Slave, which reads the clock signal and replies with data when addressed by a master.

There can be any number of master devices in the bus and roles are interchangeable in between messages, meaning that a device can switch from master to slave and vice versa. Slaves are identified by a 7-bit address, and the typical raw bit speed is 100 kbps, though this varies depending on device support. On the Arduino Uno, for example, the I^2C interface supports raw data transfers up to 400 kHz, even though the default bit rate used by the Wire library is 100 kHz.

I^2C is simple, widely supported, and easy to use in the Arduino IDE, since it is supported by the official Arduino Wire library [1]. However, setting up the physical bus requires running at least three wires from each board — ground, SDA and SCL. While this is simple for two or three boards, it can prove cumbersome when more boards are involved.

[1] `https://www.arduino.cc/en/reference/wire`, accessed on 6th July 2020

On the other hand, because it requires short physical connections, I^2C is limited to a single room. Since we want to support students working in different rooms, outside the lab, or even from their homes, a central bus such as I^2C is not appropriate for these scenarios. Moreover, despite being commonly used on the Arduino, the I^2C bus is not available or exposed to the user on Android smartphones and, therefore, can not be used to connect Arduino boards to smartphones. Nonetheless, since Arduino users are mostly already familiar with the Wire interface, the Crowdnet Arduino library was based on it, as described in Section 3.1.

2.1.2 ZigBee

Based on the IEEE 802.15.4 protocol, ZigBee is intended for use in a low-rate Wireless Personal Area Network (WPAN) for supporting simple devices consuming minimal power at a range of about 10 meters (Lee et al., 2007), though it can reach up to 100 meters under ideal conditions. It operates on the 2.4 GHz radio frequency with a maximum data rate of 250 kbps (Wagner et al., 2012) and provides multi-hop, self-organizing and reliable mesh networking with long battery lifetime while supporting different modes of operation.

Arduino support is possible, for example, using the official *Arduino Wireless SD Shield* [2], which features a ZigBee module. However, ZigBee is not compatible with the Android Operating System (OS) and it is not easy to interface with a modern smartphone. Furthermore, while its communications range of about 10–100 meters would be able to support communication between Arduino boards in different rooms in the same building, this would not be possible with Arduino boards in different homes, which may be located tens or even thousands of kilometers apart. Therefore, ZigBee is not suitable for the use cases we intended to support nor to enable communication between Arduino boards and smartphones.

2.1.3 Wi-Fi

Based on the IEEE 802.11 standards, Wi-Fi is a standard for Wireless Local Area Networks (WLANs) which is intended to replace traditional wired Local Area Networks (LANs) in some cases (Lee et al., 2007). Reaching speeds in the order of hundreds and even thousands of megabits per second, it is most commonly used to provide Internet access to users connected to a central Access Point (AP) located within a range of a few meters, though it also supports ad hoc communications.

Nowadays, Wi-Fi is widely deployed for providing easy Internet access in places such as homes, cafes, universities and other public areas. However, it is also possible to use Wi-Fi without an Internet connection to connect multiple devices together in a LAN or WLAN.

The advantage of Wi-Fi is that it can be used on Arduino boards via an external shield, such as the official Arduino WiFi Shield [3], and is available on practically all Android smartphones, as its inclusion on these devices is very much recommend as per the Android 10 Compatibility Definition[4]. Thus, it could

[2] `https://store.arduino.cc/arduino-wireless-sd-shield`, accessed on 10th August 2020

[3] `https://store.arduino.cc/arduino-wifi-shield`, accessed on 10th August 2020

[4] `https://source.android.com/compatibility/10/android-10-cdd#7_4_2_ieee_802_11_(wi-fi)`, accessed on 1st September 2020

be considered for connecting Arduino Nodes to smartphones, or even to support Arduino to Arduino communications.

Despite this, using Wi-Fi would require a complex and relatively heavy Internet Protocol (IP) stack implementation to run on the Arduino, which would not be able to run on every Arduino environment, for example due to memory constraints. Moreover, connecting the Arduino and smartphone to the same network would require that students connect the Arduino and their smartphone to the same Wireless AP. This would not be feasible due to authentication restrictions if they are using the Eduroam network [5], as is usual in universities: to connect the Arduino to the Eduroam network, users would need to store their personal authentication credentials on the Arduino, which raises security concerns. As an alternative, an ad hoc connection or software AP could be used on the smartphone or on the Arduino, but it would make the students unable to access other Wi-Fi networks or the Internet while using Crowdnet. Additionally, not all smartphones provide software AP functionality. Due to these reasons, another wireless communication method is preferred for this work.

2.1.4 BLE

BLE is a low-power WPAN aimed at short-range communications for IoT applications (Dian et al., 2018). Despite sharing part of the name, BLE is not compatible with classic Bluetooth, even though they can co-exist, and, as of the Bluetooth 4.0 specification [6], devices can implement either or both systems. Nowadays, both BLE and classic Bluetooth are implemented in most smartphones (Gomez et al., 2012). BLE support on the Arduino environment is possible either using third party modules such as the *Bluefruit LE* from Adafruit [7], or the more recent and cheap *ESP32* System on Chip (SoC) from Espressif Systems, which is available in a number of form factors (Maier et al., 2017).

BLE technology operates in the 2.4 GHz radio frequency with a maximum application layer throughput of about 221 kbps. While this throughput is relatively low when compared with other wireless technologies such as Wi-Fi, it is enough for the purposes of this work.

In a BLE connection, exactly two devices are communicating with each other. To keep the connection alive, devices must periodically exchange a packet, even if they have no useful data to exchange. These periods where devices talk to each other are known as a connection event. To minimize power usage, BLE allows tuning the time between each connection event — the connection interval — in a range between 7.5 milliseconds and 4 seconds (Gomez et al., 2012). A higher value will use less power, but will result in higher latency and lower throughputs.

There are several advantages to using BLE in Crowdnet:

- in a laboratory setting, the BLE interface could be used for other applications, such having the Arduino interact with user interfaces running on a smartphone;

[5] `https://www.eduroam.org/`, accessed on 7th October 2020
[6] `https://www.bluetooth.com/specifications/archived-specifications/`, accessed on 10th August 2020
[7] `https://www.adafruit.com/product/1697`, accessed on 11th August 2020

- even though we are targeting a laboratory setting, where every device can be plugged into utility power if needed, BLE consumes much less energy than other wireless interfaces such as Wi-Fi, which makes it more convenient, as it will conserve battery on the smartphones;
- contrary to Wi-Fi, BLE does not depend on an existing or central AP to operate; this allows students to work remotely, even in places where acquiring an AP may not be feasible or possible. Also, provided they only work with local Nodes, they can even work in places where an Internet connection is not available;
- also contrary to Wi-Fi, a complex IP stack is not necessary to use BLE, which should make it compatible with most Arduino environments;
- finally, BLE is present on most smartphones.

When compared to classic Bluetooth, BLE is also more suited for this work. In particular, BLE Central devices are able to connect to several Peripheral devices at the same time, as explained in Section 4.2, and the use of the Generic Attribute Profile (GATT) allows a more structured and efficient communication protocol to be defined on top of BLE, as explained in Section 4.2. This is contrary to mainstream classic Bluetooth shields or modules such as the HC-06 [8], which typically offer a UART-like interface when communicating with the Arduino.

For these reasons, BLE was used in this work. The details of BLE communications are further detailed in Section 4.2.

2.2 Internet communications

In this section we describe a few of the possible ways to communicate with other devices over the Internet. Though some of these methods can also be used for communication between Arduino boards using an appropriate shield, our main goal is to discuss some of the existing options for smartphones to communicate with each other.

2.2.1 LoRaWAN

Long Range Wide Area Network (LoRaWAN) is one of the most successful technologies being used in Low-Power Wide Area Networks (LPWANs). Featuring a maximum data rate of 27 kbps, it offers very low power wireless operations with long communication ranges of 2–15 kilometers (Adelantado et al., 2017). Use cases of LoRaWAN include real time monitoring of agriculture, leak detection, environment control, smart city (e.g. smart parking), etc, provided that low bandwidth and relatively high latency are not a problem.

LoRaWAN uses a star-of-stars topology, where gateway nodes relay messages between end-devices and a central network server. End-devices send data to gateways over a single wireless hop, while gateways are connected to the network server through a non-LoRaWAN network, such as the Internet. This topology is very similar to what we want to achieve: Arduino boards could connect to a

[8] `https://components101.com/wireless/hc-06-bluetooth-module-pinout-datasheet`, accessed on 9th October 2020

LoRaWAN shield, which used a LoRaWAN gateway to relay messages to a server through an Internet connection. However, there are a few key aspects of LoRaWAN that make it unsuitable for the use cases we want to support:

- to work from home or outside the laboratory, students would need to acquire a LoRaWAN gateway if one is not already available in their area; instead, using smartphones which most students already own solves this problem;
- at such low data rates, which may potentially be divided by multiple end-devices on the same gateway, LoRaWAN is limited to relatively high latencies;
- finally, LoRaWAN is not compatible with the Android OS, so it can also not be used for enabling direct communications between an Arduino and a smartphone.

2.2.2 HTTP

Hypertext Transfer Protocol (HTTP) is an application protocol designed primarily for transferring information in the World Wide Web. It uses a client-server model and runs on top of the Transmission Control Protocol (TCP) protocol.

Due to its popularity and wide compatibility, HTTP was initially considered for communication between smartphones by using an intermediary HTTP server.

However, when HTTP is used for communication in IoT applications, in which a huge number of tiny data blocks are normally transferred, the protocol overhead and resulting performance degradation are not negligible. In this application, HTTP has the following problems (Yokotani and Sasaki, 2016):

- it requires more bandwidth due to the text-based nature of the protocol;
- it requires more processing power and is more complex to encode and decode requests;
- since it uses a new short-lived TCP connection in the worst case scenario, the TCP 3-way handshake for connection establishment is potentially repeated for every application data block that needs to be transferred. Recent HTTP versions such as HTTP/1.1 are able to reuse the same TCP connection for more than one request by using a keep-alive mechanism. However, the connection is still terminated after a short period of inactivity, and embedded HTTP libraries may not support the keep-alive mechanism.

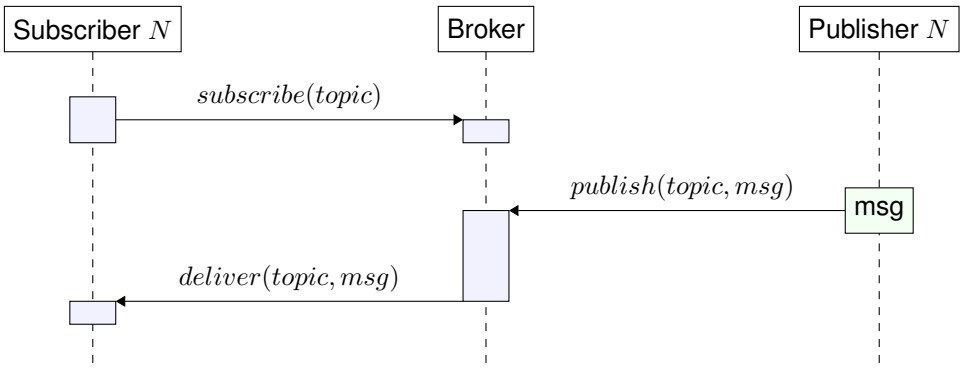
Since the MQTT protocol solves most of these problems, we quickly realised that it is, therefore, more suited for the task, as detailed in Section 2.2.3.

2.2.3 MQTT

Message Queuing Telemetry Transport (MQTT) is a lightweight network protocol based on a publish-subscribe communication pattern (Yassein et al., 2017). It is a machine-to-machine communication protocol, meaning that any device in the network can communicate with each other to exchange data. This

is done by having every device establish a Transmission Control Protocol over IP (TCP/IP) connection with a central Broker, which is responsible for delivering messages to their final destinations.

Figure 2.1: Sequence diagram of the publish-subscribe pattern used by MQTT



The publish-subscribe pattern, as depicted in Figure 2.1, provides a simple, flexible and efficient method of exchanging messages. MQTT organizes messages in *topics*, on which a client device can *publish* messages consisting of arbitrary binary data. These messages are sent to the Broker, which distributes them to other clients that have previously *subscribed* to the *topic*. This is done without clients being aware of which clients are subscribed to which *topics*. Because it uses a single long-lived outgoing TCP connection to the Broker, this pattern allows MQTT-enabled devices to subscribe to and receive messages from any other device without the need for connections between devices, which would require a mechanism for device discovery to be implemented, and would need complex address and connection management on the devices. Similarly, when a device needs to publish a message to another device, it simply sends it to the Broker, without the need to know the address of the receiver device.

Polling is the process of actively checking the status or state of some device, communication channel or activity, for example, periodically and repeatedly making a request to a server to check for new data. Another advantage of the publish-subscribe pattern used by MQTT is that, since the connection with the Broker is kept alive, devices do not need to implement polling mechanisms to check for new messages and can simply be notified of new incoming messages by the Broker. This helps keeping energy consumption and resource usage to a minimum, which is especially important for IoT applications.

MQTT also supports three different Quality of Service (QoS) levels for delivering messages, allowing us to choose between assured delivery or reduced overhead.

There are off-the-shelf MQTT Broker implementations, such as *Eclipse Mosquitto* (Light, 2017), as well as MQTT client implementations that can run on Android smartphones, such as the *Eclipse Paho* project [9].

These characteristics make MQTT very suitable to support efficient communications between smartphone devices in this work.

[9] `https://www.eclipse.org/paho/`, accessed on 12th August 2020

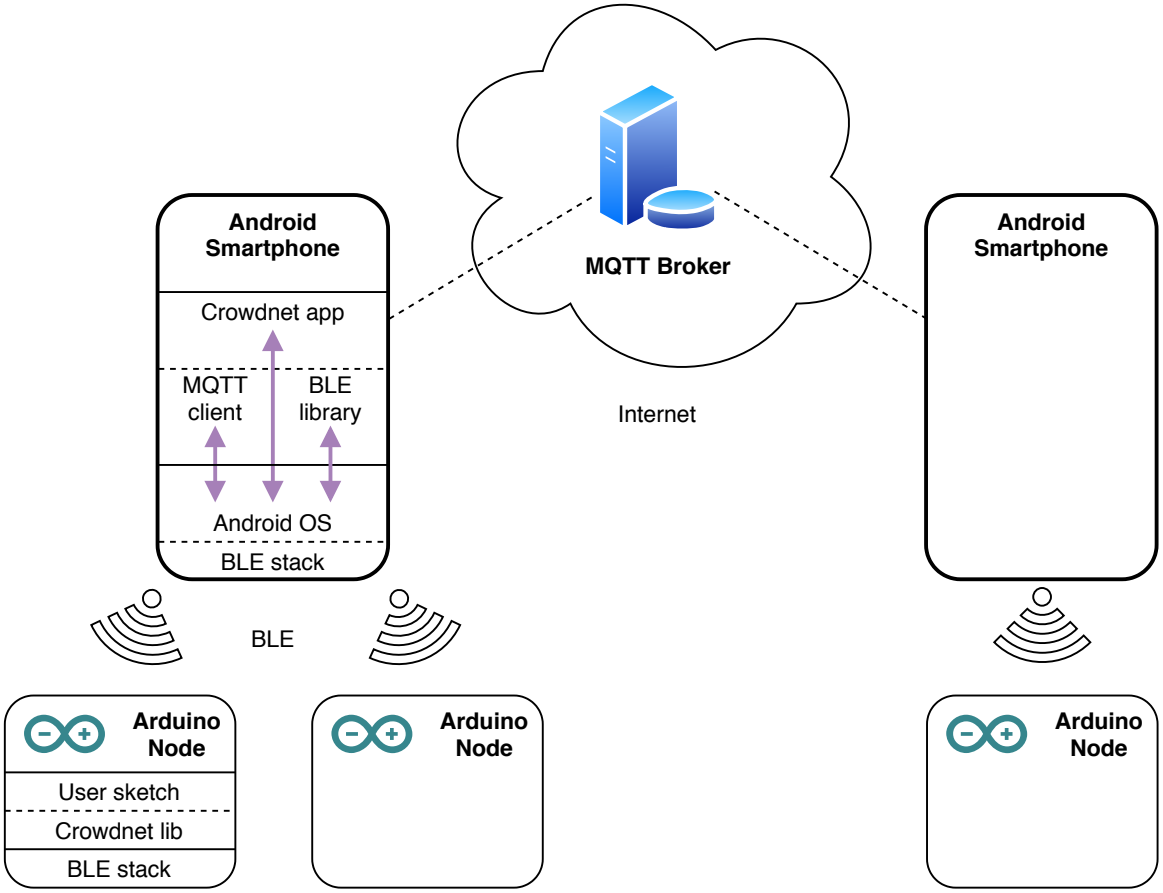
3 Architecture

This Chapter presents the architecture of the Crowdnet system and describes the interaction between its various components.

The architecture of this system consists of three main components, as depicted in Figure 3.1:

1. an Arduino library;
2. an Android smartphone application;
3. a server back-end.

Figure 3.1: Overview of the system architecture



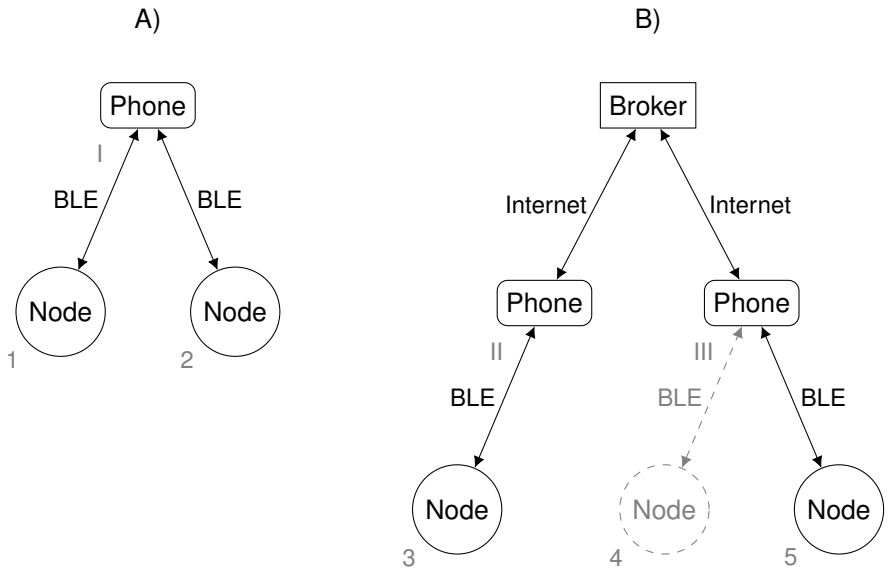
The Arduino library connects to a smartphone via BLE. In turn, the smartphone connects to the server (Broker) via the Internet. The server and the Arduino library do not communicate directly. Instead, they rely on the smartphones to serve as a middleware for communication. This creates a logical network that allows sketches running on the various Arduino Nodes to communicate with each other.

Using this architecture, we are able to support two main message delivery strategies, described in Figure 3.2:

A) Local Nodes – two or more local Nodes communicating with each other via a single smartphone. Nodes can not communicate directly with each other via BLE. This is because Nodes are acting as BLE peripherals, which can only establish a connection with a single device at a time, as further explained in Section 4.2. The smartphone, however, is acting as a BLE central and is able to connect to multiple peripherals at the same time. Therefore, in this case, Nodes are connected to the same smartphone, which is able to forward messages between the Nodes using only BLE communication, and with no intervention from the server.

B) Remote Nodes – one Node (5) communicating with a remote Node (3) via their two corresponding smartphones. In this case, smartphones connect, via the Internet, to a common server in order to forward messages between each other. Therefore, when Nodes are not in reach of the same smartphone — because they are in different rooms or belong to different students — their corresponding smartphones can still communicate indirectly with each other via the Internet, enabling communication between remote Nodes.

Figure 3.2: Main Crowdnet use cases



In a more concrete example, let us consider that various Nodes will be used by students, organized in groups, in a laboratory setting. Each group uses one smartphone and a few Nodes. Use case A) corresponds, for example, to communication between Nodes operated by same group of students in the same room, using one smartphone. Use case B), on the other hand, corresponds to Nodes operated by different groups communicating with each other. In the latter case, Nodes (3, 5) are connected to different smartphones (II, III) and, therefore, messages will need to be forwarded from one smartphone to the other. By allowing smartphones to communicate via the Internet — either by Wi-Fi, mobile data, or any other form of Internet access — students need not be in the same room and can even work on

a remote setting, for example, from their homes. Other than increasing the communication latency, this works the same way as if smartphones were in the same room.

A combination of both use cases is also supported. This is represented by the dashed Node (4) in Figure 3.2, which is able to communicate with both the local Node (5) that is connected to the same smartphone, and, via the Broker, with the remote Node (3) connected to another smartphone. In this case, smartphone II would subscribe to messages destined to Node 3, while smartphone III would subscribe to messages destined to Nodes 4 and 5. As explained in Section 2.2.3, due to the publish-subscribe pattern, to send a message to a remote Node, a smartphone does not need to be aware of which smartphone the remote Node is connected to. It simply sends the message to the Broker. After smartphones are properly subscribed, whenever the Broker receives a message, it is able to forward it to the corresponding smartphone, which finally sends it to the corresponding local Node.

3.1 Arduino Library

The first component of the system is an Arduino library which users can include while developing their own sketches. *Sketch* is the name that Arduino uses for a program or project that runs on an Arduino board. Using the Arduino IDE[1], users are able to write, compile, and upload Sketches to an Arduino board.

The Crowdnet Arduino library sits on the edge of the logical Crowdnet middleware, as per Figure 1.1, and communicates with the Android application via BLE. An Arduino board running a user sketch that communicates using the Crowdnet Arduino library is called a *Node*. The library allows a Node to indirectly communicate with another Node via the logical Crowdnet network, or middleware.

The library uses an API which closely matches the interface offered by the Arduino Wire library [2] for the I^2C protocol. This is due to the following reasons:

- The target users of the Crowdnet system are already familiar with the I^2C protocol and the Wire library.
- Since the interface is very similar, sketches originally designed to communicate via I^2C using the Wire library can easily be modified to use the Crowdnet library instead.
- Using BLE directly is not a trivial task, and requires the user to be aware of various details of this protocol, as well as the implementation details of the underlying platform — in our case, the BLE library for ESP32, as explained in Section 4.1. Typically, most of the users of the Arduino environment — such as electronics hobbyists and beginner programmers — do not have the required background knowledge nor the time to understand all the aspects of BLE. Therefore, having a simple interface that abstracts away all the details of BLE and makes it look like the simpler interface of the Wire I^2C library makes it easier for everyone to start using the Crowdnet library right away.

[1] `https://www.arduino.cc/en/main/software`, accessed on 8th September 2020
[2] `https://www.arduino.cc/en/reference/wire`, accessed on 6th July 2020

For I²C compatibility, Nodes are assigned a 7-bit identifier (ID), similar to the device address used by I²C, which identifies the Node in the Crowdnet middleware. This is further detailed in Section 4.2.

3.1.1 Using the Arduino Library

For the end user, using the Crowdnet Arduino library should be as simple as using any other Arduino library[3]. However, since the version of Crowdnet we implemented is made to run on an ESP32 board, the user needs to first setup the Arduino IDE to be able to compile Sketches targeting the ESP32.

The Arduino core for ESP32 allows the user to compile sketches for ESP32. A Makefile is included with the Crowdnet implementation that automatically downloads and installs `arduino-cli`, a command line alternative for the Arduino IDE, and also installs the Arduino core for ESP32. Refer to the provided `README.md` file. However, it is also possible to do this using the Arduino IDE by following a few steps:

1. open the Arduino IDE and navigate to the *Preferences* window (under the *File* menu);
2. under the *Additional Board Manager URLs* field, add the URL for the ESP32 package index[4];
3. in the *Tools* menu, select *Board* and then *Boards Manager*;
4. in the *Boards Manager* window, look for and install the ESP32 platform;
5. after the installation finishes, select the appropriate ESP32 board before compiling a sketch.

Additional instructions are available on the Arduino-ESP32 repository[5].

To install the Crowdnet library, the user should use the provided Makefile, or refer to the common instructions for installing an Arduino library.

3.2 Android smartphone application

The Android application supports communication between:

- local Nodes, which are within range of the BLE signal of a smartphone;
- remote Nodes, which are not directly accessible by a given Node or smartphone, but can be reached via the server.

Therefore, the Android application serves as a bridge or gateway between the Arduino library and the server in the following ways:

- it receives packets from local Crowdnet-enabled Nodes, containing a message and the address (or ID) of a destination Node;
  - if the destination Node is not a local Node, the message is sent to the server;

[3] `https://www.arduino.cc/en/reference/libraries`, accessed on 8th September 2020
[4] `https://raw.githubusercontent.com/espressif/arduino-esp32/gh-pages/package_esp32_index.json`, accessed on 8th September 2020
[5] `https://github.com/espressif/arduino-esp32#readme`, accessed on 8th September 2020

  - otherwise, it is sent to the local Node;
- using the server, it subscribes to messages that are destined for local Nodes;
- when notified by the server of a new incoming message from a remote Node, it receives the message and sends it to the corresponding local Node.

Communication with the Arduino library running on the Nodes is done using BLE, while communication with the server is done via the Internet using the MQTT protocol, as described, respectively, in Section 4.2 and Section 4.3.

It is also worth noting that, for the smartphone to serve a gateway for local communication via BLE, that is, forwarding packets between local Nodes via BLE, no intervention from the server is needed. An Internet connection to the server is only necessary when communication with remote Nodes is requested. This allows students to still work offline if, for example, they have no Internet connection available at home.

3.3 Server back-end

To be able to communicate with each other, the various instances of the Android application interact with a server back-end.

This server is a simple MQTT Broker that receives, temporarily stores, and forwards messages destined to remote Nodes via their local Android smartphone — the smartphone to which a Node is connected via BLE.

An *Eclipse Mosquitto* (Light, 2017) MQTT Broker instance can easily be hosted running in a Virtual Machine (VM) at Rede das Novas Licenciaturas (RNL)[6] to be used by students. However, the Android application supports any MQTT-complaint Broker running on an arbitrary server and, should they wish to, students can host their own server.

The implementation of the server is described in Section 4.3.

[6] `https://rnl.tecnico.ulisboa.pt/servicos/maquinas-virtuais/`, accessed on 18th August 2020

4 Implementation

This Chapter explains the implementation details of the Crowdnet architecture. We start by looking into how the Arduino library was implemented and how it communicates with the Android smartphone via BLE. Then, the server communication and the various components of the Android application are explained.

The source code for the implementation described in this Chapter, including the Arduino library and the Android Application, is available at the author's homepage [1].

4.1 Arduino library

The Crowdnet Arduino library was implemented as a standard Arduino library, following the Arduino Library specification [2]. It is implemented in C++ by the `Crowdnet` class. To offer the same interface as the `Wire` library, the `Crowdnet` class:

- implements the `Print::write()` method, which allows the user to call the familiar `print` and `println` methods offered, for example, by the `Wire` and `Serial` classes;
- inherits the `Stream` class, which supports a few familiar methods such as `available()` and `read()`;
- implements most methods offered by the `Wire` library for I^2C, such as
  - `begin(...)`
  - `beginTransmission(...)`
  - `endTransmission()`
  - `onReceive(...)`
- Additionally, we provide an optional `setup(...)` method that allows the user to change the BLE device name, as explained in Section 4.2.3.

The Crowdnet Arduino library was implemented to run on an *ESP32* SoC from Espressif Systems, which is available in a number of form factors (Maier et al., 2017), such as the `ESP32-WROOM-32`[3] module that we used.

The reason for choosing the ESP32 board instead of an Arduino board with an additional BLE shield was that it was not possible to acquire a BLE shield for the existing Arduino boards in a timely manner due to bureaucracies, budget and stock constraints. The ESP32, on the other hand, is very

[1] `https://web.tecnico.ulisboa.pt/~nuno.m.ribeiro.silva/crowdnet/`, accessed on 25th October 2020
[2] `https://arduino.github.io/arduino-cli/latest/library-specification/`, accessed on 9th October 2020
[3] `https://www.espressif.com/en/products/modules`, accessed on 8th September 2020

cheap and was readily available to be acquired by the author. Additionally, the ESP32 incorporates both functionalities of an Arduino and BLE shield in the same board, simplifying the setup.

Even though the ESP32 is not an official Arduino board, it is compatible with the Arduino environment by installing the Arduino core for ESP32, as explained in Section 3.1.1. User sketches that make use of the Crowdnet library for Arduino can be compiled and uploaded to run on the ESP32.

To be able to use BLE hardware available on the ESP32, the Crowdnet library for Arduino depends on a BLE wrapper library[4], written in C++, that is part of the Arduino core for ESP32. This library abstracts a few low level details of manipulating the BLE stack and hardware, and offers an API that is similar to other BLE libraries available for some Arduino boards[5]. However, other APIs are not 100% compatible and will also be different when working with other BLE shields. Therefore, even though we are using the Arduino environment, our implementation will naturally need some porting if another BLE shield or module is to be used. Nonetheless, the same protocol that we implemented on top of BLE can be used. This is further discussed in Section 6.1.

Note that these implementation details are completely transparent to users of the Crowdnet Arduino library, since all they need to do in their sketches is to interface the library using the I^2C-like API it offers.

The details of BLE communications in Crowdnet are common to both the Arduino library and the smartphone application, and are explained below.

4.2 BLE communication

In order to enable the Arduino Crowdnet library to communicate with the Android application, and vice versa, a custom protocol was developed on top of BLE.

Using the Crowdnet BLE protocol, the Nodes and the Android Application are able to send messages containing arbitrary content to each other. The length of the messages is limited by the underlying BLE protocol:

- the default Maximum Transmission Unit (MTU) of application data sent over the BLE stack is 20 bytes (Dian et al., 2018);
- it is possible to change the MTU to an arbitrary value up to 512 bytes, depending on the version of the underlying BLE stack;
- Crowdnet meta-data sent along with messages introduces 2–8 bytes of overhead;
- in practice, this allows messages of up to 504 bytes to be sent, as long as we change the MTU before sending any messages. This is done by the Android application.

In BLE communications, devices address each other using an EUI-48 Media Access Control (MAC) address. The Extended Unique Identifiers (EUIs) are assigned to BLE and other network-enabled de-

[4] `https://www.arduino.cc/reference/en/libraries/esp32-ble-arduino/`, accessed on 25th August 2020
[5] `https://www.arduino.cc/en/Reference/ArduinoBLE`, accessed on 8th September 2020

vices during manufacturing, and their assignment is managed by the Institute of Electrical and Electronics Engineers (IEEE) .

In the Crowdnet middleware, Nodes are identified throughout the system by the EUI-48 MAC address of their BLE interface. However, for I^2C compatibility, a 7-bit Node identifier (ID) is also supported and is mapped to the corresponding MAC address. When an Arduino sketch uses the Crowdnet library, the user can choose to send a message using either the ID or the MAC address. Since Nodes only communicate directly with the smartphone they are connected to, there is no need for a Node to be aware of the *ID to MAC* mappings in order to transmit a message. Smartphones, on the other hand, may receive messages destined to a given ID, and need to determine which BLE device to forward the message to. For this to be possible, smartphones need to keep track of the *ID to MAC* mappings of their respective local Nodes, that is, the Nodes connected via BLE. The mapping for remote Nodes is handled by the smartphones and the server, and is detailed in Section 4.3.

BLE devices can serve one of two Connection Roles:

- the Central Role, which is used by the device that scans for advertisements and initiates connections;
- the Peripheral Role, which is used by the device which advertises itself and accepts connections from Central devices.

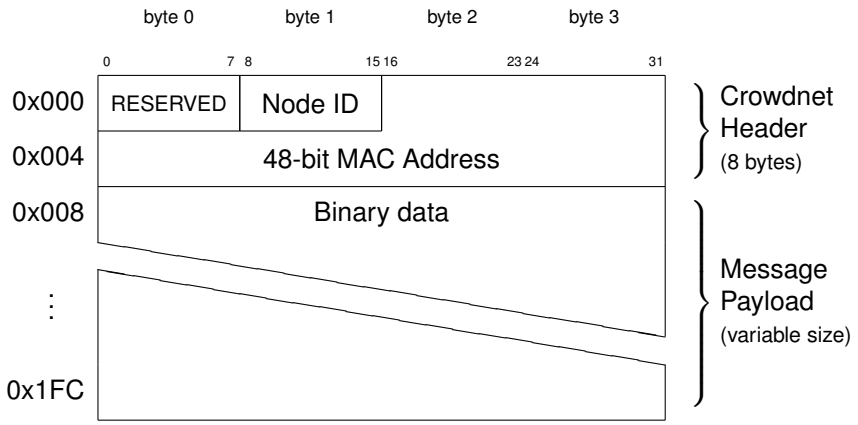
A Central device can connect to multiple Peripherals, but one Peripheral can only connect to one Central. Since we need smartphones to connect to multiple devices, these use the Central role. The way two BLE devices transfer data back and forth is governed by the Generic Attribute Profile (GATT), which defines a few concepts that enable communications (Dian et al., 2018). Independently of the Connection Roles, a device can act either as a GATT Server or as a GATT Client. The Server stores data locally, while the Client accesses or sends data to the Server. In Crowdnet, Node devices use the Peripheral role with one GATT Server, while the smartphone assumes the Central Role as a GATT Client. Upon user request, smartphones scan for Nodes that are advertising the Crowdnet Service and connect to them.

4.2.1 Packet structure

When the Crowdnet Arduino library and the Android Application need to exchange messages, they send packets to each other using a well-defined structure. A packet is a Crowdnet message, as well as some meta-data, that is sent over the `Node Tx` and `Node Rx` characteristics described in Section 4.2.2.1. The structure of the content of these packets is illustrated in Fig. 4.1.

The packet begins with one byte that is reserved for future use and must be zero. Next is one byte representing an I^2C-like Node ID. An ID equal to zero is reserved to mean that no Node ID was assigned. The next 48-bits (6 bytes) encode a MAC address. The Node ID and MAC address fields correspond to:

Figure 4.1: Crowdnet BLE packet structure



- the *destination* address, when the packet is transmitted *from* an Arduino; In this case, the MAC address field is only present if the ID is zero. When the ID is not zero, the MAC address field is omitted and the payload starts at byte 0x002.
- the *source* address, when the packet is transmitted *to* an Arduino. In this case, the MAC field is always present, even when the ID is non zero.

Next are the actual message contents — the payload — which can have a variable size between 0 and $512 - 8 = 504$ bytes (0x1F8), when using an MTU of 512 bytes, as previously discussed. Even though it is of variable length, the exact size of the payload field does not need to be part of the packet itself, since it can be derived in runtime from the total size of the packet, as received by the BLE stack, by subtracting the size of the other fields. Note that, though it is not very useful for now, empty payloads are accepted.

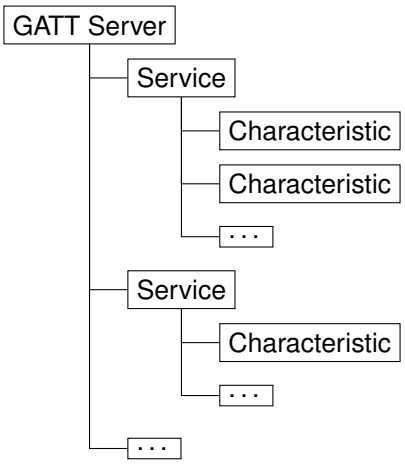
Since we do not support direct BLE communication between Nodes, when two Nodes communicate with each other via Crowdnet, they always need to send or receive a BLE packet to or from a smartphone. When a Node is transmitting, the *source* address corresponds to the address of the Node's BLE interface and is, therefore, already known by the smartphone which receives the packet. Similarly, when a Node is receiving a packet from a smartphone, the *destination* address is also already known by the Node — itself. For this reason, the Crowdnet BLE packets only need to carry one address at a time.

4.2.2 GATT Service

In BLE communications, devices exposing a Server offer one or more GATT Services, which are identified by a Universally Unique Identifier (UUID). These services can then expose one or more Characteristics, as depicted in Figure 4.2.

When a BLE peripheral device is not connected to any other device, it can send BLE advertisements of a few bytes, which are broadcast to whichever BLE devices are listening. Included in these

Figure 4.2: Simplified hierarchy of a BLE GATT Server



advertisements is the BLE device name, MAC address and, among other things, a list of custom UUIDs identifying the Services offered by the peripheral.

The Crowdnet Nodes offer and advertise a single GATT Service — the Crowdnet GATT Service, which is identified by the 128-bit pseudo-randomly-generated UUID `00000000-0000-4043-ae4f-57444e455430`.

If a device is advertising this Service UUID, then we — the Crowdnet smartphone application — know that it is a Crowdnet Node and the smartphone can connect to it. Therefore, this UUID is used by the smartphone to scan for, and connect to, BLE devices which are running the Crowdnet GATT Service. During the scan, devices can also be filtered by a *Group ID*. This is explained later in Section 4.2.3.

4.2.2.1 GATT Characteristics

The Crowdnet GATT Service is created by the Crowdnet Arduino library to enable communication with the smartphone via BLE. Each Characteristic exposed by a Service is identified by a UUID and contains zero or more Properties and a value. Characteristics can be thought of as variables or data containers, and Properties describe what operations can be made regarding the Characteristic. The following GATT Characteristics are exposed by the Crowdnet Service, as listed in Table 4.1:

Table 4.1: GATT Characteristics exposed by the Crowdnet GATT Service

Description	UUID	Properties
Node Tx	`00000000-0002-4043-ae4f-57444e455430`	`N`
Node Rx	`00000000-0003-4043-ae4f-57444e455430`	`W`
Node Attributes	`00000000-0004-4043-ae4f-57444e455430`	`R`

Node Tx – Node to Gateway communication. A Node writes to this characteristic when it needs to transmit a packet to the smartphone. This characteristic supports notifications (`N`), meaning that

the smartphone does not need to poll it for new data, and is notified by the Node's BLE stack when new data is available.

Node Rx – Gateway to Node communication. This characteristic is writable (`W`). A smartphone writes to this characteristic when it needs to send a packet to the Node.

Node Attributes – Contains read-only (`R`) information about the Node, such as its I²C-compatible address — the Node ID — and its Group ID. More attributes can be added if needed for future use.

Values passed in the `Node Tx` and `Node Rx` characteristics use the Crowdnet packet format described in Section 4.2.1.

4.2.3 Group ID filtering

When setting up the Arduino library, besides choosing an arbitrary BLE device name for the Node, it is possible to assign it a *Group ID*. Not to be confused with the *Node ID*, the *Group ID* is not related to the workings of I²C or BLE, but it is used by the smartphone to filter Nodes found during a scan. If enabled, this features allows the Android application to only connect to Nodes belonging to a Group ID that was previously chosen by the user in the application. This feature allows multiple smartphones and Nodes to operate in the same room — that is, in BLE range of each other — without trying to connect to Nodes belonging to another group. This is especially useful in a laboratory setting involving various groups of students, each with a different smartphone and set of Nodes.

Because the *Group ID* needs to be read by the smartphone during scanning and before actually connecting to the device, the simplest way of having Nodes exposing the *Group ID* at this stage is by concatenating the *Group ID* to the BLE device name chosen by the user, which is already part of the advertisement packet. To facilitate parsing, the *Group ID* in the device name is encoded in its base-10 American Standard Code for Information Interchange (ASCII) representation, and is prefixed by a space and the letter "`G`". For example, assuming a user chooses the name "Nuno" for a Node and assigns it *Group ID* `123`, the actual BLE device name that would be advertised is "`Nuno G123`". Note, however, that naming nodes is a convenience, optional, feature of Crowdnet that is not necessary for communications to work, and, therefore, the device name may be left unchanged by the user.

The *Group ID* is also present in the `Node Attributes` characteristic detailed above. However, characteristics can only be read by the smartphone after a connection is established with the peripheral. At this point, the scanning and filtering process was already completed and, therefore, this can only used to double check that the *Group ID* that was previously parsed from the device name is correct.

As described in Section 4.2.2, BLE advertisements can include a list of Service UUIDs. This list can include more than one UUID, depending on their size. Therefore, an alternative approach to broadcast the *Group ID* of the Nodes would be to include it in the advertisements as a separate Service UUID, along with (but separate from) the Crowdnet Service UUID. This would not only simplify the process of filtering by *Group ID*, but would also leave the BLE device name to be freely chosen by the user in its entirety. However, UUIDs shorter than 128 bits can only be assignment by the Bluetooth Special

Interest Group (SIG), subject to a formal application process, or chosen from a publicly accessible list of 16-bit UUIDs that are already assigned for common use cases (for example, temperature sensors, battery level indicators, etc). Due to the small size of the advertisements, and since we do not intend to formally apply for a shorter UUID with the SIG, in practice this means that our work is only able to fit a single 128-bit UUID in the advertisements, which is already used to advertise the Crowdnet Service. Therefore, this approach was discarded.

Another alternative that was considered was to calculate a custom GATT Service UUID derived from the *Group ID*, that would be advertised instead of a fixed Crowdnet Service UUID. This combined 128-bit UUID would still fit in the advertisement packet. However, it would make the process of scanning for Crowdnet devices more complex and inefficient when the *Group ID* feature was disabled, as in this case the smartphone would need to scan the entire *Group ID* namespace in order to be able to find every possible *Group ID* and its corresponding UUID combination.

4.3 Server communication

The Android application instances, running on various smartphones, communicate with each other using an MQTT client that connects to an MQTT Broker, which we call *server*.

MQTT uses a lightweight *publish-subscribe* model to deliver arbitrary *messages* via TCP, as explained in Section 2.2.3. Crowdnet messages are organized by the Broker in *topics*. A client can *subscribe* to topics it is interested in or *publish* messages on a given *topic* in the Broker. These messages are sent to, or *published* on, the server — the MQTT Broker — which distributes them to other clients that have previously *subscribed* to the *topic*. This is done without clients being aware of which clients are subscribed to which topics.

Whenever a new message is published on a given topic, it replaces any previously published value in that topic. This helps keeping the MQTT protocol simple, but unfortunately means that any messages published to the Broker before any client has subscribed to the corresponding topic are lost. However, since Crowdnet is intended for real time communications, this is not a concern, and an analogy can be made that this would be equivalent to trying to send messages across an I^2C bus without connecting the wires. Messages published to the server while a client is temporarily disconnected (for example, when it temporarily loses Wi-Fi connectivity) are also lost. This could be solved by using different QoS levels in MQTT and by enabling server sessions, allowing the Broker to store messages to be delivered after the client reconnects. However, it remains as future work.

In Crowdnet, two topics are reserved for each destination Node: one topic for sending messages to a Node addressed by MAC and another topic for addressing Nodes by ID. These topics are, respectively, identified by the strings `"crowdnet/messages/mac/<MAC>"` and `"crowdnet/messages/id/<ID>"`, where `<MAC>` and `<ID>` correspond, respectively, to the BLE MAC address or I^2C-compatible ID of the Node to which the message should be forwarded to. By organizing topics in this manner, the Broker

is automatically able to efficiently and effectively deliver messages to the correct smartphone, which in turn delivers it to the Node.

Whenever the smartphone receives a message for an unknown Node via BLE, the message is published to the server on one of the topics corresponding to the destination Node. Similarly, when the smartphone connects to a Node via BLE, if the smartphone is able to reach the server, the former subscribes to the topic corresponding to this Node's MAC address and, if an ID is also being used, to the topic corresponding to this Node's ID. This way, the smartphone is automatically notified whenever a new message for this Node is published in the server by another smartphone, allowing smartphones to forward messages between each other and deliver them to the corresponding Nodes.

For example, the topic `"crowdnet/messages/id/23"` is used for messages destined to the Node whose ID is 23. The smartphone which is directly connected to this Node via BLE subscribes to this topic and forwards received messages to the Node.

Additionally, since smartphones subscribe to both ID and MAC topics belonging to their local Nodes, the former need not be aware of the *ID to MAC* mapping of remote Nodes: to send a message to a remote Node, they can simply use the ID topic to publish the message without knowing the corresponding MAC address, and the server will take care of delivering the message.

The Crowdnet MQTT Broker server is implemented by a Linux VM running an instance of *Eclipse Mosquitto* (Light, 2017), a Free and Open-Source Software (FOSS) lightweight MQTT Broker implementation written using the C programming language.

The Crowdnet Android application uses an MQTT client implementation provided by the *Eclipse Paho* project, as explained in Section 4.4.1.

4.4 Android application

In this section, the various components of the Crowdnet Android application are described. The application was implemented using the Java programming language, as is usual with most Android applications. It is able to run on Android 6.0 Marshmallow — API level 23 — or newer, which, at the time of writing, should cover 85% of devices that are active in the Google Play Store[6].

The application is divided into a few separate Java packages or components: the Crowdnet domain, two Android Services (one for Crowdnet logic and one for MQTT), and the User Interface (UI). These components depend on each other and on a few external dependencies that are bundled with the application.

4.4.1 External dependencies

The Crowdnet Android application depends on the following external libraries:

[6] `https://developer.android.com/about/versions/marshmallow`, accessed on 4th September 2020

- for MQTT communications, an MQTT client implementation provided by the *Eclipse Paho* project[7] is used. It creates a separate Android Service to handle the MQTT connection to the Broker and process requests from our application.
- the BLESSED[8] library, which is used for interfacing with the BLE adapter. It is a wrapper around the standard Android BLE classes [9] that takes care of a few aspects of working with BLE on Android, such as:
  - Queueing commands — Android expects only one BLE operation to be active at a time. For example, starting a write operation before a read has finished will result in the write being ignored. BLESSED takes care of this by automatically queueing commands for execution, one at a time.
  - Helper classes for parsing data received via BLE Characteristics, such as the Crowdnet packets described in Section 4.2.1.
  - Managing Characteristic descriptors automatically, for example, when enabling notifications for a Characteristic.
- the Android Jetpack[10] library, which is used to draw the UI and, among other things, to make the application code compatible with older Android versions.

These libraries are included as dependencies of the Crowdnet Android application, as is usual in the Android ecosystem, meaning that when the user installs the Crowdnet Android application on their smartphone, there is no need to manually install anything else — they are already bundled with the application package.

4.4.2 Domain

The Crowdnet domain is implemented in a separate Java package and is responsible for abstracting all the details of the Crowdnet middleware in the Android application. It interfaces with both BLE and MQTT using, respectively, the BLESSED and Paho libraries mentioned above.

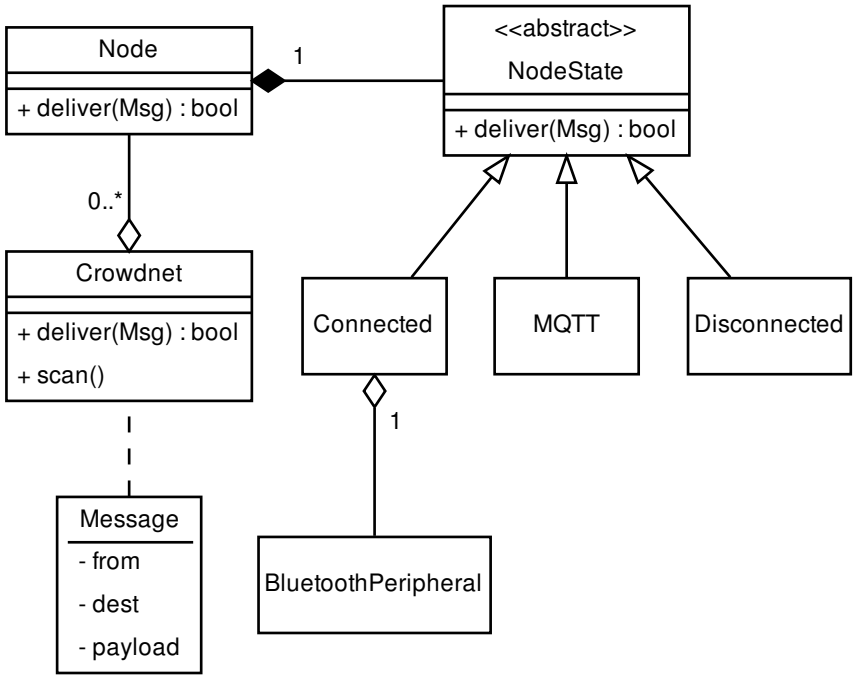
The domain is made up of the following main classes, as per Figure 4.3. For simplicity, only the most relevant classes are described:

Message represents a message that needs to be forwarded to a `Node`. It contains the source address, destination, and payload. Can either be created from a packet received via BLE or when data is received from the MQTT Broker.

Node represents and stores information about an Arduino `Node`, either remote or local, that the application is aware of; `Node` instances are created either by performing a BLE scan or by sending `Messages` to an unknown destination.

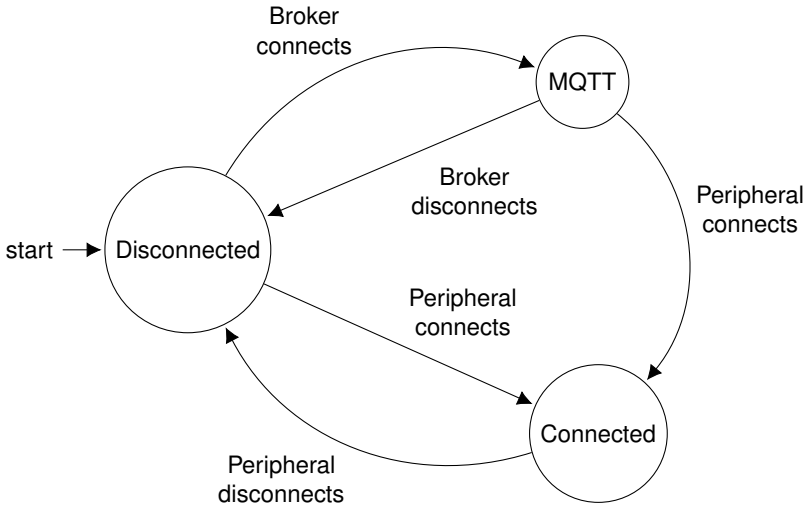
[7] `https://www.eclipse.org/paho/`, accessed on 12th August 2020
[8] `https://github.com/weliem/blessed-android`, accessed on 1st September 2020
[9] `https://developer.android.com/guide/topics/connectivity/bluetooth-le`, accessed on 1st September 2020
[10] `https://developer.android.com/jetpack/`, accessed on 4th September 2020

Figure 4.3: Simplified UML Class diagram of the Crowdnet Domain in the Android application
Note: some classes and methods are omitted for simplicity.



NodeState is an abstract class implementing the State and Strategy Patterns; each `Node` is associated with one `NodeState`; the operation of delivering a `Message` to a `Node` is delegated to the `Node`'s concrete `NodeState`; the `NodeState` prefix is omitted from the class names for simplicity (e.g. the `Connected` class is actually implemented as `NodeStateConnected`); the transitions between states are depicted in Figure 4.4:

Figure 4.4: Crowdnet Node states and transitions in the Android application



Connected `Nodes` transition to this state when connected via BLE to the smartphone. This state stores a reference to a BLE Peripheral. When this state is created, if the MQTT Broker is

connected, the `Node` is subscribed to the topics corresponding to its MAC and ID, as described in Section 4.3. This state is able to deliver `Messages` by generating a BLE packet, as per Section 4.2.1, that is sent over the `Node Rx` Characteristic, as per Section 4.2.2.1.

MQTT used by remote `Nodes` when the smartphone is connected to the MQTT Broker. It is only possible to transition into this state if the `Node` is not connected via BLE. This state is able to deliver `Messages` by publishing them to the MQTT Broker using the `CrowdnetMQTT` class.

Disconnected used by `Nodes` that have been found in a BLE scan but are not connected, or by remote `Nodes` if the MQTT connection dies.

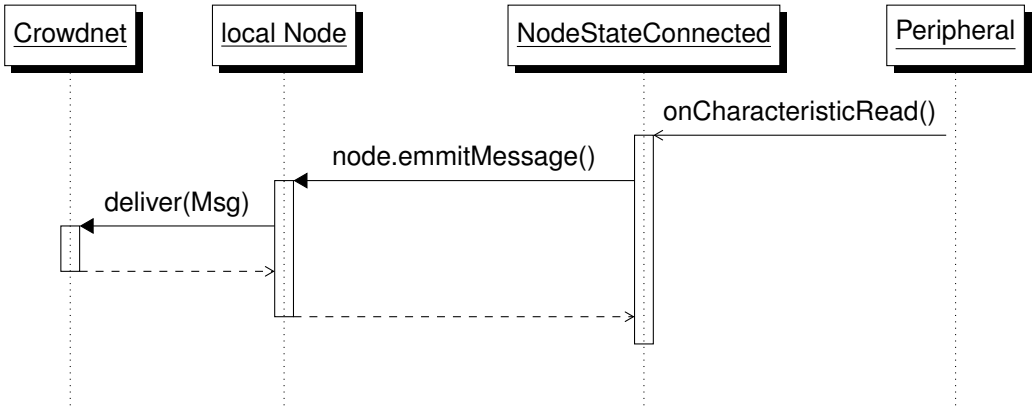
Crowdnet is a Singleton Facade class for the Crowdnet domain, which is used by the Crowdnet Service.

CrowdnetBLE takes care of managing the BLE Central client by handling BLE scans and initiating connections to local `Nodes`.

CrowdnetMQTT handles communication with the MQTT Broker. It also notifies all the `Nodes` when the MQTT connection with the Broker changes. Nodes then delegate this to the `NodeState`, which decides whether a state transition is needed.

To understand how Crowdnet delivers Messages to local or remote Nodes, we will first look into how a Message is received from a local Node. When a Node sends a Message to the smartphone via BLE, its corresponding BLE Peripheral class from the BLESSED library sends a callback to the State associated with the `Node` instance for that Node. Then, the state parses the BLE packet to create a `Message` instance. The `Node`'s `emmitMessage()` method is then called, which asks `Crowdnet` to deliver the message to its destination. This is depicted in Figure 4.5.

Figure 4.5: Sequence diagram of Crowdnet receiving a Message via BLE



To deliver a Message, `Crowdnet` tries to find the corresponding destination `Node` instance by MAC or ID. If one is not found, it is assumed to be a remote Node, and a new `Node` instance is created and transitioned to the MQTT state. Once `Crowdnet` finds the `Node` instance, its `deliver(Msg)` method is called, which delegates the Message delivery process to the current Node State. Then, depending on the State, the Message is delivered via BLE or MQTT, as depicted, respectively, in Figures 4.6 and 4.7.

The advantage of this delegation using the Strategy pattern is that the delivery algorithm is automatically selected depending on the current state of the Node.

Figure 4.6: Sequence diagram of Crowdnet delivering a Message via BLE

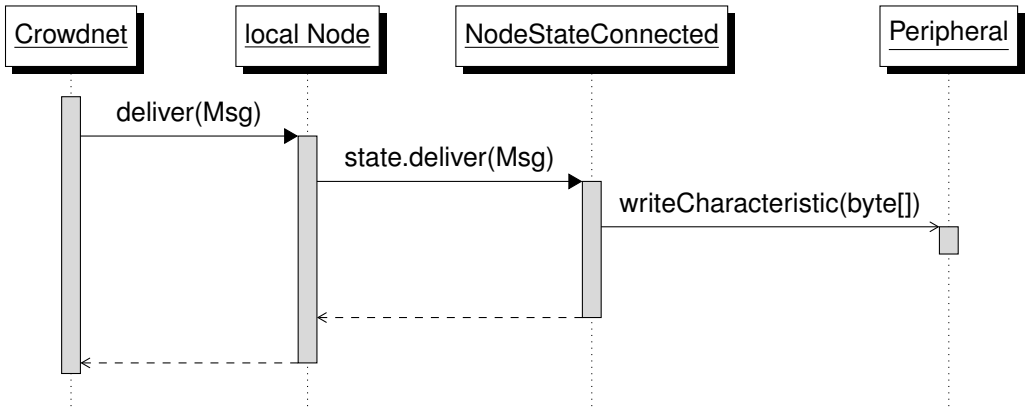
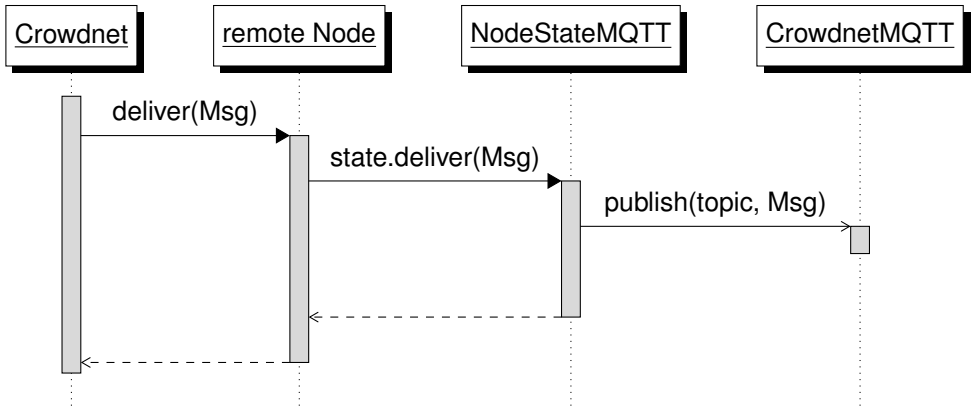


Figure 4.7: Sequence diagram of Crowdnet delivering a Message via MQTT



To receive Messages from remote Nodes, the `MQTT` Node State first subscribes to topics for the corresponding Node, as explained in Section 4.3. Since it is the Node class itself that receives callbacks for these topics, in this case there is no need for intervention from the `Crowdnet` class: when a Message is published to the topic, the Broker notifies the smartphone and a callback is received by the corresponding `Node` instance, which can then deliver the Message to itself using its state, as per Figure 4.8.

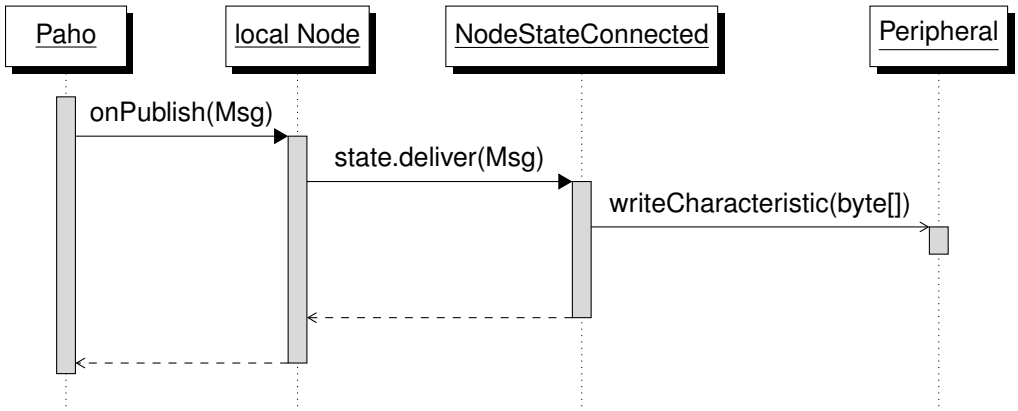
4.4.3 Crowdnet Service

The Crowdnet Service is a simple but important component of the Crowdnet Android application. It is implemented by the `CrowdnetService` class, which extends the Android Service class[11], and serves as a bridge between the UI classes and the Crowdnet Domain. It also keeps Crowdnet running in the background whenever the user navigates away from the Crowdnet Activity[12].

[11] `https://developer.android.com/guide/components/services`, accessed on 4th September 2020

[12] `https://developer.android.com/guide/components/activities/intro-activities`, accessed on 4th September 2020

Figure 4.8: Sequence diagram of Crowdnet receiving a Message via MQTT



This Service supports operations such as initiating a BLE scan, connecting to the MQTT Broker, changing the Group ID, etc. These operations are initiated by the UI by invoking the Service, which delegates them to the Crowdnet Domain.

4.4.4 User Interface

The Crowdnet User Interface (UI) is implemented by a single Android Activity that changes between displaying one of multiple Fragments[13] using a navigation drawer. Each Fragment corresponds to a separate screen:

- the Setup screen, which allows the user to change a few key settings of Crowdnet, turn on the Bluetooth adapter, and connect to the MQTT Broker, before finally initiating the first BLE scan.
- the Nodes screen, which displays a list of Nodes that are currently connected to the smartphone.

These Fragments are responsible for managing the user experience in each screen by drawing and updating the interface, handling taps on the screen, etc. The implementation details of these Fragments is not particularly relevant to the Crowdnet project and, therefore, will not be detailed in depth.

[13] `https://developer.android.com/guide/components/fragments`, accessed on 6th September 2020

5 Evaluation

This Chapter describes a few tests that were performed to evaluate the performance of Crowdnet. We begin by evaluating the speed throughput of Crowdnet using BLE and then both MQTT+BLE. Then the latency of Crowdnet is evaluated and the results are compared with reference values for the I^2C bus.

5.1 Communication throughput

This section describes various tests where two separate setups were used to compare the throughput of Crowdnet while communicating via both BLE and MQTT, using different payload sizes. The results are then compared with a reference BLE throughput value.

Two ESP32 Nodes were programmed to communicate with one another under different conditions. One Node is configured to send packets, while the other Node is configured to receive packets while measuring:

- the time between the first and last packet;
- the total number of bytes received;
- the total number of packets received.

From these parameters, we can calculate the throughput of the system under various conditions, as explained below.

In each setup, one Node was configured to send a burst of 9 packets to a receiver Node. Each burst corresponds to one sample, and each sample uses a different payload size. The samples in the graphs below were computed by the receiver Node by measuring the total number of bytes that it received in the payloads and then dividing by the difference between the timestamp (in milliseconds) when the last (j) and first (i) packets were received in each burst. The value is then multiplied by the number of milliseconds in one second, resulting in the average speed in bytes per second, as shown in Equation 5.1.

$$\text{average\_burst\_speed}(i,j) = \frac{\left(\sum_{x=i}^{j} size_x\right) - size_i}{timestamp_j - timestamp_i} \times 1000 \tag{5.1}$$

Since the time measured by the receiver corresponds to the instant when the transmission of the packet has ended, the payload size of the first packet ($size_i$) is excluded from the measurements, as it was exclusively transmitted before the measurement started ($timestamp_i$). Also, since the time between packets is also included in the measurement, the computed value also accounts for the packet processing time in the smartphone (and server), not only the air time of the packets.

For these throughput tests, the default `CONNECTION_PRIORITY_BALANCED` [1] setting was used on Android, which instructs the BLE stack to use the connection parameters recommended by the Bluetooth SIG. Note, however, that it is possible to achieve greater throughputs by using `CONNECTION_PRIORITY_HIGH` instead, at the cost of higher energy use.

5.1.1 BLE only

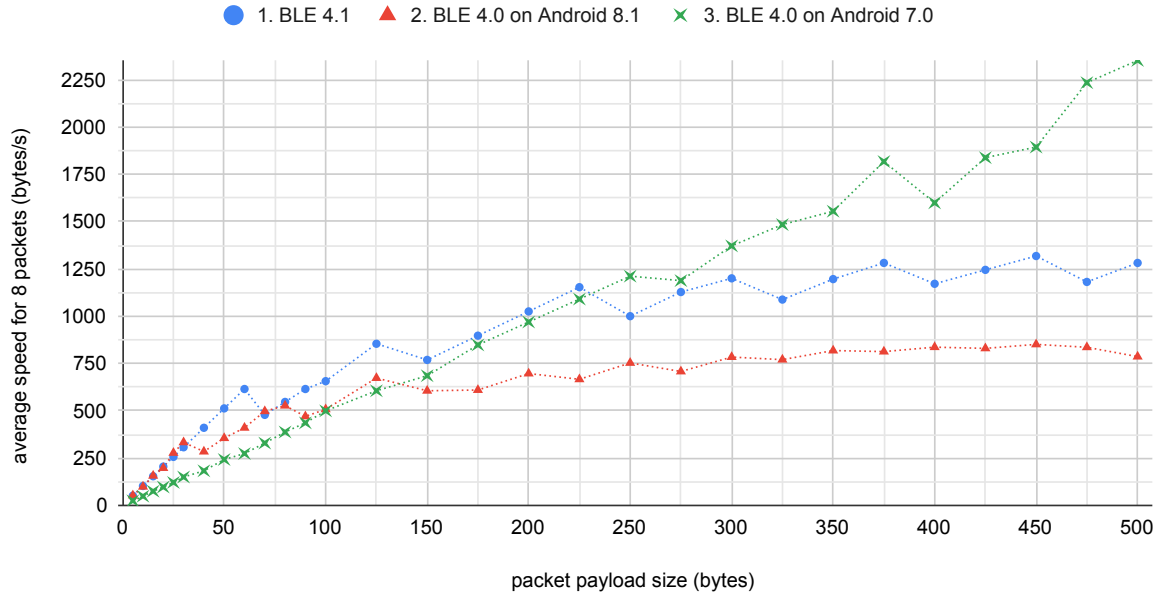
First, we tested the throughput of Crowdnet while using just two ESP32 Nodes, as described before, connected to a single smartphone via BLE, that is, using two local Nodes.

The same test was ran a few times on each of three different smartphones, which run different versions of Android and use different versions of BLE. The reason for choosing these particular devices is simple — they are the only ones the author had access to:

1. an OnePlus One, running Android 6.0.1 and BLE 4.1.
2. an Asus ZC520TL, running Android 8.1 and BLE 4.0;
3. an Asus ZC520TL, running Android 7.0 and BLE 4.0;

Each test used the exact same setup, but a different smartphone. The results are show in Figure 5.1. In each run, the results for each device were consistent, presenting only a small variation between tests on the same device. For simplicity, only one test per device is shown. Each sample (dot, triangle, or cross) in the graph is the average speed of a burst of 8 packets of equal payload size, calculated as per Equation 5.1.

Figure 5.1: Average Crowdnet receiving speed versus payload size for BLE using two Nodes communicating via a smartphone. Each series uses a different smartphone.



[1] `https://developer.android.com/reference/android/bluetooth/BluetoothGatt#CONNECTION_PRIORITY_BALANCED`, accessed on 30th August 2020

Concerning BLE communications, the first thing to note is that, as expected, the throughput increases when using greater payload sizes. This is likely because greater packet sizes help reducing periods of radio silence, which are considered in our average calculation, making more efficient use of the available radio bandwidth.

The maximum measured speed was about 2353 bytes/s while using a payload size of 500 bytes per packet on the Asus ZC520TL running Android 7.0 and BLE 4.0. This corresponds to a speed of about 294 kbps (kilobits per second), which is about 1.3 times faster than the maximum BLE 4.2 application layer throughput of about 221 kbps that was measurement by Dian et al. (2018) when using a peer-to-peer network of only two devices. It is worth noting that, while the connection priority setting on Android influences the underlying BLE connection interval, Android does not allow changing or reading the connection interval value directly. Therefore, we can not be certain that the connection interval we used matches the one used by Dian et al. (2018).

Using the other two smartphones, however, the maximum observed speed is significantly lower, with the next best performing smartphone — the OnePlus One, running Android 6.0.1 and BLE 4.1 — achieving a maximum speed of about 1319 bytes/s (165 kbps) while using a payload size of 450 bytes per packet. This is contrary to what was expected of these devices: it was expected that a more recent version of BLE would perform better than BLE 4.0, which was not the case.

It is therefore clear that, in practice, the throughput of BLE will depend on the device's manufacturer, operating system version, firmware, and even device drivers. This is even more noticeable in the case of the two Asus ZC520TL that were tested, which, despite using the exact same hardware, achieved a very different throughput. The fact that the ESP32 Nodes support BLE 4.2, while the smartphones used in these tests use BLE 4.0 or 4.1, may also play a role in the obtained results.

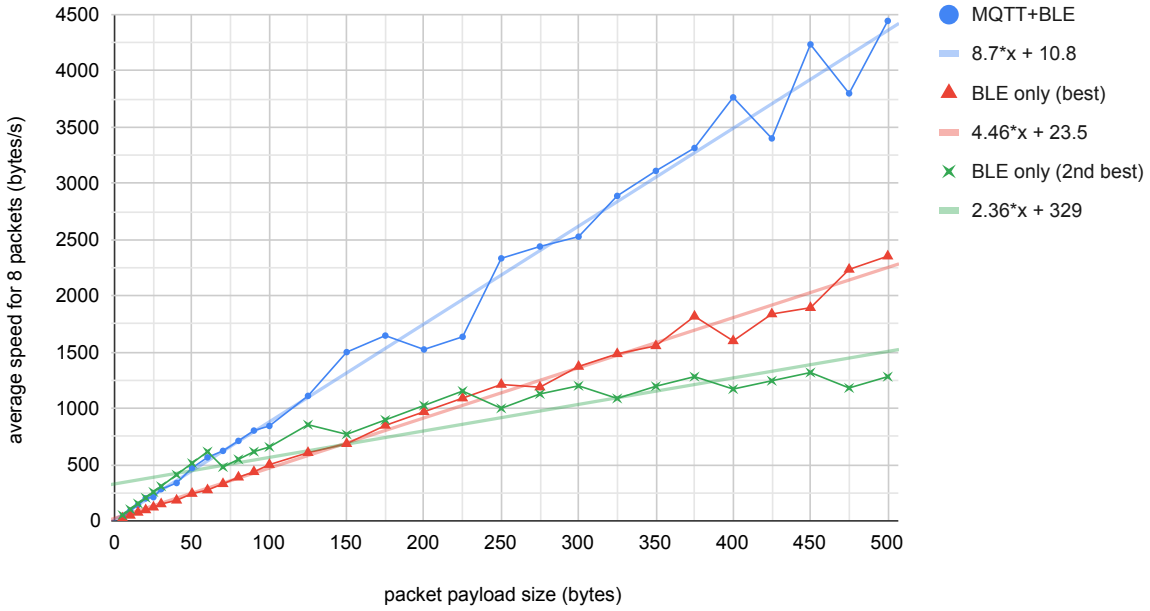
Despite the achieved speeds being slower than what was expected of BLE communications in some cases, these are still sufficient for the use cases for which Crowdnet is intended, and will most likely vary depending on the smartphones that will be used. Furthermore, these can be increased by using `CONNECTION_PRIORITY_HIGH`, at the cost of higher energy use.

5.1.2 BLE and MQTT

After testing how Crowdnet performs using just BLE, we tested how it performs when using MQTT in conjunction with BLE, and compared it with the results from the best of the three smartphones in Figure 5.1. This is so we can see how using MQTT for communications between remote Nodes affects the way Crowdnet performs when compared to using only BLE in local Nodes. For this test, the best two performing devices from the previous test were used, each connected to its own ESP32: the Asus ZC520TL running Android 7.0 and BLE 4.0, and the OnePlus One.

The results are shown in Figure 5.2. Each sample (dot, triangle or cross) is the average speed of a burst of 8 packets of equal payload size, calculated as per Equation 5.1. The line in red corresponds to the speed over BLE that previously measured by connecting both Nodes via BLE to the smartphone

Figure 5.2: Average Crowdnet receiving speed versus payload size for BLE and MQTT



that better performed in the BLE-only test — the Asus ZC520TL running Android 7.0 and BLE 4.0. Then, shown in blue, is the speed over BLE+MQTT as measured by connecting each Node to a different smartphone in different rooms. Smartphones were connected via Wi-Fi, using the same AP, to an MQTT Broker running on the same LAN. These two setups correspond, respectively, to use cases A) and B) that were previously depicted in Figure 3.2. A linear regression of the samples of each test is also shown.

As expected, when using MQTT+BLE, the maximum speed of 4.4 kilobytes per second that was measured is about 2–3 times greater than when using only BLE, even though the two smartphones showed different BLE throughputs in the previous tests. This may be explained by the following:

- When two Nodes are connected to the same smartphone via BLE, a shared medium (the air) is used for transmission. This means that only one of the three devices is able to transmit at a time, assuming the smartphone uses the same BLE channel to communicate with both Nodes. In particular, the smartphone is unable to forward packets to the receiver Node while the sender Node is transmitting. This alone cuts the BLE throughput by half, plus the processing overhead of the smartphone.
- On the other hand, when using MQTT with Nodes in a different room, out of range of each other:
  - the first smartphone can transmit via MQTT while the first Node is sending the next packet;
  - the second smartphone can transmit to the receiver Node, via BLE, while simultaneously receiving the next packet via MQTT;
  - therefore, in this case, the two Nodes are able to respectively transmit and receive at the same time;

- Furthermore, the smartphones are able to reach the MQTT Broker via Wi-Fi, which is able to achieve a greater throughput than BLE, as explained in Section 2.1.4.

5.2 Communication latency

This section describes various tests that were performed in order to compare the latency of Crowdnet while communicating via BLE using different payload sizes.

In order to accurately measure time intervals, an accurate and synchronized clock is needed. Since it is not trivial to synchronize the clocks of multiple Nodes, only one Node was used in these tests.

A single ESP32 Node was programmed to send packets destined to itself via Crowdnet. The Node was connected to a single smartphone via BLE, which receives the packets and sends them back to the Node. The latency values are calculated by the Node as follows:

1. the Node stores the current timestamp ($start$) as the number of milliseconds since system reset;
2. the packet is sent via BLE with the destination address of the same Node;
3. the Node waits until the packet is received back; during this time:
   (a) the packet is handed over to the ESP32 BLE stack, which
   (b) sends the packet via BLE to the smartphone;
   (c) the smartphone receives and processes the packet;
   (d) the packet is sent back to the Node by the smartphone;
4. when the packet is received, the Node records the current timestamp (end) and a latency value for this packet is then given by subtracting end from $start$;
5. this procedure is then repeated until the requested number of packets was sent and received;

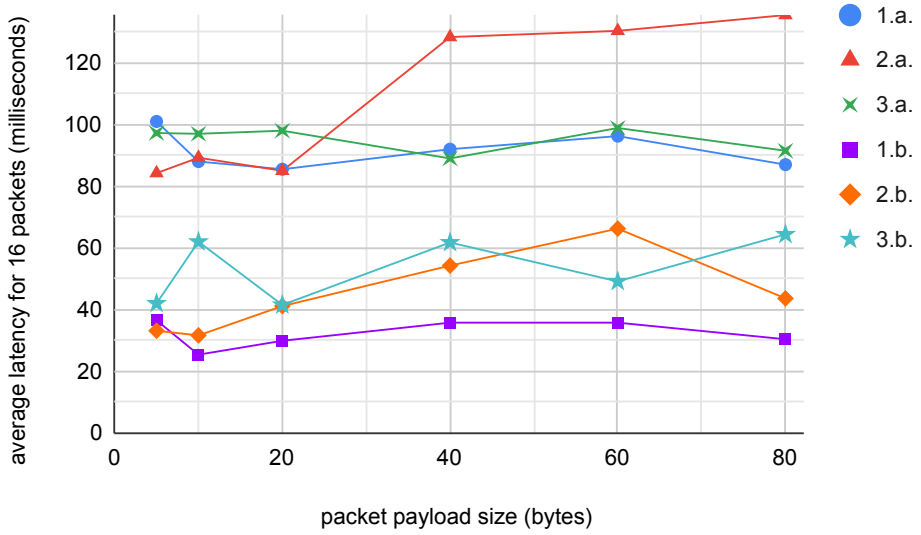
Therefore, the latency value for each packet includes the total air time of the packet for two round trips, the processing time on the smartphone, and the processing time on the Arduino Node.

For each payload size that was tested, 16 packets were sent. Then, an average latency value was calculated by averaging the 16 samples. Each test was repeated, respectively, on each of the three smartphones used to evaluate the throughput of BLE in Section 5.1.1, which are numbered from 1 to 3. For each smartphone, the test was repeated using different connection priority values on Android, though it is not clear what exact connection interval values are used by these priorities:

a. `CONNECTION_PRIORITY_BALANCED`, which uses the connection parameters recommended by the Bluetooth SIG;
b. `CONNECTION_PRIORITY_HIGH`, which requests a low latency connection.

The results are shown in Figure 5.3. The first thing we can see is that, as expected, the latency is reduced to about half when using a high connection priority on all smartphones that were tested.

Figure 5.3: Average Crowdnet round-trip latency versus payload size for BLE
1–3 are different smartphones; `a.` uses the default connection priority while `b.` uses high priority



Also, since the minimum connection interval allowed by BLE is 7.5 milliseconds, we can expect that the measured latency values will be above that. Indeed, the minimum observed latency value was 25 ms when using a payload size of 10 bytes on the OnePlus One (1.b.).

Using small payload sizes, the results between each smartphone are similar. However, just like in the BLE throughput tests in Section 5.1.1, the results start to show significant variations between devices when larger payload sizes are used.

From these tests we can conclude that, in practice, the expected latency between Nodes using BLE will be in the range of 25 to about 136 milliseconds, depending on the smartphone, the connection priority, and the payload size that is used. It is also worth noting that these values are well above what is expected of an I^{2}C bus. To further compare Crowdnet to I^{2}C, let us consider that, for example, on the Arduino Uno the I^{2}C interface supports data transfers up to 400 kHz. I^{2}C packets include 9 bits of metadata, start and stop conditions, and application data, which includes an acknowledge bit after every byte. Therefore, on the Arduino Uno, the maximum theoretical throughput we could get, for example using 10 bytes of application data, is 316 kbps, with a round-trip latency of about 0.5 milliseconds, excluding packet processing time. Crowdnet, on the other hand, using the same payload size, only achieved a throughput of 96 bytes/s (768 bps) when using MQTT+BLE, and a much higher round-trip latency of 25 milliseconds.

Despite this, the performance offered by Crowdnet is still suitable for the use cases for which Crowdnet is intended, and the conveniences offered by the system when compared to I^{2}C are a trade-off that users must consider. For example, the traffic lights project discussed in Section 1.1 requires very little bandwidth of only a few bytes per second, and only needs to send a few messages every few seconds, or in response to an external event such as a pedestrian crossing button being pressed. Therefore,

latencies of a few dozen milliseconds are acceptable, and will not impact the performance of the overall project, nor will introduce a noticeable delay in the response time of the traffic lights or other sensors operated by humans.

6 Conclusion

This work introduced a middleware for communication between Arduino boards by using the BLE and Internet connectivity of smartphones. Its architecture is able to support communication between both local and remote Nodes, allowing students to work in and out of the laboratory, as well as from their homes. The API offered by the Crowdnet Arduino library is similar to that of I^2C, making it simple to be used by most users that are familiar with the Arduino environment. It was implemented and evaluated using the ESP32 SoC, the implemented Android smartphone application, and the Mosquitto MQTT Broker.

Evaluation of our implementation showed that the Crowdnet system was able to achieve reasonable throughput speeds and latency, even though these results vary considerably depending on payload size, the smartphone's manufacturer, operating system version, firmware, and even device drivers. Nonetheless, the achieved throughput is adequate for the use cases for which Crowdnet is intended, and using MQTT in conjunction with BLE provides greater throughputs than just BLE.

6.1 Future Work

Future improvements to the Crowdnet system could include porting the Arduino library to support additional BLE shields on other Arduino boards, and adding more features to the Android application, such as the ability to send sensor or input data to the Arduino, as explained below.

6.1.1 Using Crowdnet on other Arduino boards

As explained in Section 4.1, the Crowdnet Arduino library was implemented to run on an ESP32 board.

However, it would be desirable to be able to use the Arduino Uno boards that already exist in the laboratory with Crowdnet. For this to be possible, there are least two possible approaches:

1. use the ESP32 as a BLE module, which the Arduino Uno interfaces with using a serial interface such as the Serial Peripheral Interface Bus (SPI) or I^2C;
2. acquire a separate BLE shield for Arduino.

Using the first approach, the Crowdnet library could still run on the ESP32, requiring no changes to the BLE-specific API calls of ESP32. In this case, an additional sketch or library that translates Crowdnet or I^2C operations would need to be developed and would run on both the ESP32 and the Arduino Uno, communicating, for example, via SPI.

As for the second approach, the Crowdnet library would run on the Arduino Uno. For this reason, the BLE API calls that are specific to the ESP32 library would need to be ported to use the API offered by the desired BLE shield.

Note that, in both cases, the same protocol that we implemented on top of BLE, as described in Section 4.2, can be kept as-is, and the Android application and server would require no changes.

6.1.2 Interaction between the smartphone and Arduino

Since Crowdnet uses a BLE interface to connect an Arduino board to a smartphone, in a laboratory setting, this connection could be used for other applications, such having the Arduino interact with user interfaces running on the smartphone. For example, other applications running on Android could interact with the Crowdnet application to send arbitrary data such as sensor readings or user input to the Arduino. Another possibility is to add such functionality directly to the Crowdnet application. This is similar to what is done by the *1sheeld* project[1], which emulates various Arduino shields in a single Android application that connects to the Arduino via a single shield, also using BLE.

Communication from the Arduino to the smartphone is also possible. Therefore, another possible application is to use Crowdnet to gather data from many Crowdnet-enabled sensors using Arduino boards. These sensors can, for example, be installed inside the laboratory and the Crowdnet architecture would allow students to conveniently interact with them locally or remotely.

[1] `https://1sheeld.com/`, accessed on 9th September 2020

Bibliography

Adelantado, F., X. Vilajosana, P. Tuset-Peiro, B. Martinez, J. Melia-Segui, and T. Watteyne (2017, September). Understanding the Limits of LoRaWAN. *IEEE Communications Magazine 55*(9), 34–40.

Cunha, A. R. (November 2017). Software for Embedded Systems Laboratory Guide. Lisbon. Instituto Superior Técnico.

Dian, F. J., A. Yousefi, and S. Lim (2018). A practical study on Bluetooth Low Energy (BLE) throughput. In *2018 IEEE 9th Annu. Inf. Technol. Electron. Mob. Commun. Conf. (IEMCON), Inf. Technol. Electron. Mob. Commun. Conf. (IEMCON), 2018 IEEE 9th Annu.*, pp. 768–771.

Gomez, C., J. Oller, and J. Paradells (2012). Overview and evaluation of bluetooth low energy: An emerging low-power wireless technology. *Sensors (Switzerland) 12*(9), 11734–11753.

Lee, J.-s., Y.-w. Su, and C.-c. Shen (2007). A Comparative Study of Wireless Protocols: Bluetooth, UWB, ZigBee, and Wi-Fi. In *IECON 2007 - 33rd Annu. Conf. IEEE Ind. Electron. Soc.*, pp. 46–51. IEEE.

Leens, F. (2009, 02). An introduction to I2C and SPI protocols. *Instrumentation & Measurement Magazine, IEEE 12*(1), 8–13.

Light, R. A. (2017). Mosquitto: server and client implementation of the MQTT protocol. *Journal of Open Source Software 2*(13), 265.

Maier, A., A. Sharp, and Y. Vagapov (2017, 09). Comparative analysis and practical implementation of the ESP32 microcontroller module for the Internet of Things. In *2017 Internet Technologies and Applications (ITA)*, pp. 143–148. IEEE.

Rahiem, M. D. H. (2020, June). The Emergency Remote Learning Experience of University Students in Indonesia amidst the COVID-19 Crisis. *International Journal of Learning, Teaching and Educational Research 19*(6), 1–26.

Wagner, M., B. Kuch, C. Cabrera, P. Enoksson, and A. Sieber (2012). Android based Body Area Network for the evaluation of medical parameters. In *10th Int. Work. Intell. Solut. Embed. Syst.*, pp. 33–38. IEEE.

Yassein, M. B., M. Q. Shatnawi, S. Aljwarneh, and R. Al-Hatmi (2017, May). Internet of Things: Survey and open issues of MQTT protocol. In *2017 International Conference on Engineering MIS (ICEMIS)*, pp. 1–6. IEEE.

Yokotani, T. and Y. Sasaki (2016, 09). Comparison with HTTP and MQTT on required network resources for IoT. In *2016 International Conference on Control, Electronics, Renewable Energy and Communications (ICCEREC)*, pp. 1–6.