Abstract

The Internet has found its place in almost every area of our day to day lives and it looks set to continue with the development of the Internet of Things (IoT). The IoT will bring online low-power sensors and everyday appliances, allowing for networks these objects, and enabling them to interact with existing Internet infrastructure. The IoT could revolutionize lives, with potential use-cases in everything from Remote Patient Monitoring, to Manufacturing, to Autonomous Vehicles. However, the progress of the IoT is hindered by a lack of standardization. The wireless technologies used to create these networks of 'things' existed before the premise of an IoT began and are not optimised for such networking. This can be seen in the absence of Internet Protocol compatibility in the most popular of these technologies, Bluetooth. This report explores the feasibility of creating a generic architecture to circumvent these shortcomings.

Contents

Chapter 1 Introduction

1.1 Context and Motivation

Over the past four decades, the Internet has expanded to have a place in most aspects of everyday life. This is expansion looks to continue with the creation of the Internet of Things (IoT), which aims to bring sensors and appliances online. The IoT will pave the way for systems that interact with data from their physical environment such as smart homes, and smart factories. For the IoT to work, these 'Things' must be made accessible. Gartner Inc. forecasted that 8.4 billion connected 'Things' would be active in 2017, rising to 20.4 billion in 2020 [1]. The primary means of communication for these devices is over short-range, low-power wireless technologies to systems in close proximity. The restrictive range of these technologies, and the lack of IP compatibility for some, leads to a huge volume of valuable data being under-utilised.

However, these wireless technologies have seen widespread adoption within consumer electronics. In 2018, 100% of smartphones, tablets, and laptops shipped will be Bluetooth enabled [2]. This enables the use of the everyday smartphone, tablet, or laptop as a gateway to interface between these 'Things' and the Internet. One field that would greatly benefit from this is Remote Patient Monitoring (RPM).

RPM is an extremely effective technique employed by healthcare systems to relieve pressure on their physical systems. Studies on the effectiveness of RPM found that patients, that had suffered heart failure, benefitting from telemonitoring in a transitional care model saw a 50% reduction in 30-day readmissions [3]. RPM is also extremely beneficial from a business perspective. The market of RPM is far more profitable than the traditional form of patient monitoring and has grown 54.9% between 2013 and 2018 [4]. IBM forecast the global revenues of mobile health to be $21.5 billion in 2018 [4]. For medical professionals, RPM offers a reduction in paperwork, and an increase in time with high priority patients.

1.2 Research Objectives

The aim of this report is to explore the feasibility of translating short range technologies into RESTful protocols in a generic fashion with a focus on a healthcare use case. This would enable remote access to data typically restricted by short-range technologies. An application leveraging the proposed solution would allow the use of the average smartphone, tablet, or laptop as a gateway to access local sensor networks.

One of the requirements for the design is to ensure it is a generic solution. While the scope of the desired prototype is only to integrate one wireless technology, the ultimate aim is to create a generic solution for the problem space.

1.3 Technical Approach

The system has several requirements. The gateway must be connected to the internet to receive and interpret the client's requests. The gateway must also be able to communicate with the local 'things' (in this case, sensors) over a short-range, low-power wireless technology. Similarly, the client must also be connected to the internet to receive requests and forward them to the gateway. The local 'things' must be available to communicate over a short-range, low-power technology, however the system ought to still function without any 'things' in close proximity.

The protocol used for the communication to the gateway should have low overheads to facilitate the use of a constrained device as the gateway, and to cope with lossy networks. It would also be beneficial if the protocol could easily translate to HTTP to allow the client be interfaced by a HTTP client.

1.4 Overview of Report

Chapter 2 will provide a discussion on the technologies considered for the prototype implementation. The author will then present the chosen technology and provide a more detailed technical background.

Chapter 3 outlines the high-level design for the proposed system. The author will discuss each component of the design in detail, as well as the proposed algorithms at the centre of the system.

Chapter 4 will walk through the specifics of the prototype implemented by the author. Each component of the system will be discussed at a lower level, looking at the design decisions made in implementing the system. The author will also discuss the issues that were encountered during the implementation process.

Chapter 5 will offer an evaluation of the implementation qualified by the findings of the author's testing.

Finally, chapter 6 will reflect on the report, offering a conclusion and proposals for further work on the topic.

Chapter 2 Background

This chapter will provide a summary of the background research conducted ahead of designing the system and implementing a prototype.

2.1 Technology

The system proposed in this report requires a low-power wireless technology, and a suitable IoT protocol. This section will present a discussion of the candidate technologies, concluding with a decision and in-depth discussion of the selected technologies.

2.1.1 Wireless Technologies

At lowest level of the architecture, the system is required to communicate with constrained sensor networks over a low energy wireless protocol. There are various technologies which would be appropriate for the proposed use case, such as Zigbee, Z-Wave, ANT, 6LoWPAN, and Bluetooth Low Energy.

Zigbee is an IEEE 802.15.4-based specification for low rate wireless personal access networks (WPAN) which offers reliable, multi-hop mesh networking. It is designed to be used in battery-powered applications where low data rate, low cost, and long battery life are the primary concerns. Zigbee operates within the 868 MHz, 915 MHz, and 2.4 GHz frequency bands with a maximum data rate of 250 K bits per second [5].

Z-Wave is a wireless protocol designed to reliably transmit short messages over a mesh network, operating in the 900 MHz Industrial Scientific Medical (ISM) bands, as well as the 2.4 GHz band [6], over a range of 30 meters [7]. Z-Wave can transmit data at 9.6 and 40 K bits per second [6].

ANT is an ultra-low-power, short range protocol designed for sensor networks or similar applications. It is capable of point-to-point, star, tree, and mesh network topologies. ANT operates in the 2.4 GHz band and has a maximum throughput of 1 Mbps [8], and has been designed to work for 30 meters line of sight [9].

6LoWPAN is a protocol defined to enable IPv6 packets be carried on low power wireless networks, specifically IEEE 802.15.4. 6LoWPAN was designed under the belief that Internet Protocol could and should be applied even to the smallest devices and offers an IP based solution without any proprietary technology. It is available on three frequencies: 868 MHz, 902-928 MHz and 2.4 GHz [10], and offers data rates from 20 to 250 kbps [11] with a physical range of up to 1.5km [12].

Bluetooth Low Energy (BLE) is a low-power alternative to the standard Bluetooth protocol, designed by the Bluetooth Special Interest Group (SIG), for short range communications. It was introduced as

part of the Bluetooth 4.0 specifications in 2010 as a specialized protocol for low power, low bandwidth and low-cost systems. BLE operates in the 2.4 GHz ISM band [13], with an approximate throughput of 125 kbps and a reliable range of 30 meters [14].

Each of the discussed wireless technologies would be a valid choice for the sensor communication level. However, for the scope of this project, only one wireless technology is to be implemented. The main requirements for protocol are low energy consumption, and high adoption. BLE has the lowest energy consumption of the discussed protocols [15]. BLE is also the most widely adopted technology discussed. 100% of the 2.3 billion phones, tablets, and laptops expected to be shipped in 2018 will be Bluetooth enabled [2]. This enables the use of everyday devices such as smartphones within the system as a gateway.

2.1.2 IoT Protocols

The communication between the client and the gateway in the system is done over an IoT application protocol to facilitate the use of a constrained device as the gateway. There are four suitable application protocols for the system use case. They are Constrained Application Protocol (CoAP), Message Queuing Telemetry Transport (MQTT), Extensible Messaging and Presence Protocol (XMPP), and Advanced Message Queueing Protocol (AMQP) [16].

CoAP was created by the IETF Constrained RESTful Environments (CoRE) working group as an application layer protocol for IoT applications. CoAP was designed as a lightweight, RESTful protocol to work with very limited resources. One core method for reducing the complexity of CoAP is sending messages over UDP, instead of TCP [17]. Each UDP package sent over CoAP uses a four-byte binary header, followed by a sequence of one-byte option headers. An average CoAP header consists of 10 to 20 bytes [17], compared to a HTTP header which ranges from approximately 200 bytes to 2KB [18]. CoAP's base specification defines a familiar four request methods: GET, PUT, POST, DELETE. [19], and later specifications added the OBSERVE request which allows a client to 'observe' resources [20].

MQTT is a lightweight IoT publish-subscribe protocol that runs over TCP. MQTT is designed to provide one-to-many messaging, with three qualities of service for message delivery: 'at most once', 'at least once', and 'exactly once'. MQTT has two core specifications: MQTT, and MQTT-S which defines a UDP mapping. MQTT does not support the labelling of messages with metadata, so subscribing devices are assumed to have a priori knowledge of how to interpret the incoming data.

XMPP is an open-source protocol designed for near real-time instant messaging which has been extended to enable publish-subscribe models. XMPP's core specification is outlined in RFC 6120 published by the IETF. XMPP traditionally runs over TCP, transporting XML data which is divided into three components: message, presence, and iq (info/query). These sections outline the source and destination of the message as well as the message content, show and notify users of updates, and pair senders and receivers respectively.

AMQP is a robust application layer protocol designed to support multiple messaging applications and communication patterns. AMQP provides the same three qualities of service as MQTT and operates over a reliable transport layer such as TCP. AMQP supports both request-response and publish-subscribe architectures. An AMQP frame has an 8-byte header, the first four of which are the size of the frame. The next two bytes outline the Data Offset (DOFF), and message type, while the final two bytes of the header are type dependent.

Each of the proposed protocols would be appropriate for the suggested use case. However, the scope of the project is exploratory, so only the most appropriate protocol is to be implemented. Of the discussed protocols, CoAP is the most robust. The addition of the OBSERVE specification enables the addition of a publish-subscribe architecture in tandem with a request-response model. While this is also possible using AMQP, the ease of translation from HTTP to CoAP is a clear advantage over AMQP as it facilitates the provision of a graphical user interface (GUI) over a HTTP client.

2.2 Bluetooth Low Energy

As previously discussed, BLE is the low energy evolution of standard Bluetooth. BLE is designed for very low power operations and consumes between 1-50% the energy of traditional Bluetooth [21]. BLE is a high potential technology within the area of the IoT. This section will offer an overview of the BLE technology.

2.2.1 Bluetooth Stack

As shown in figure 2.1, a complete single-mode BLE device is divided up into three components: controller, host, and application. Each of these blocks of the protocol stack are split into several layers to provide the necessary BLE functionality.
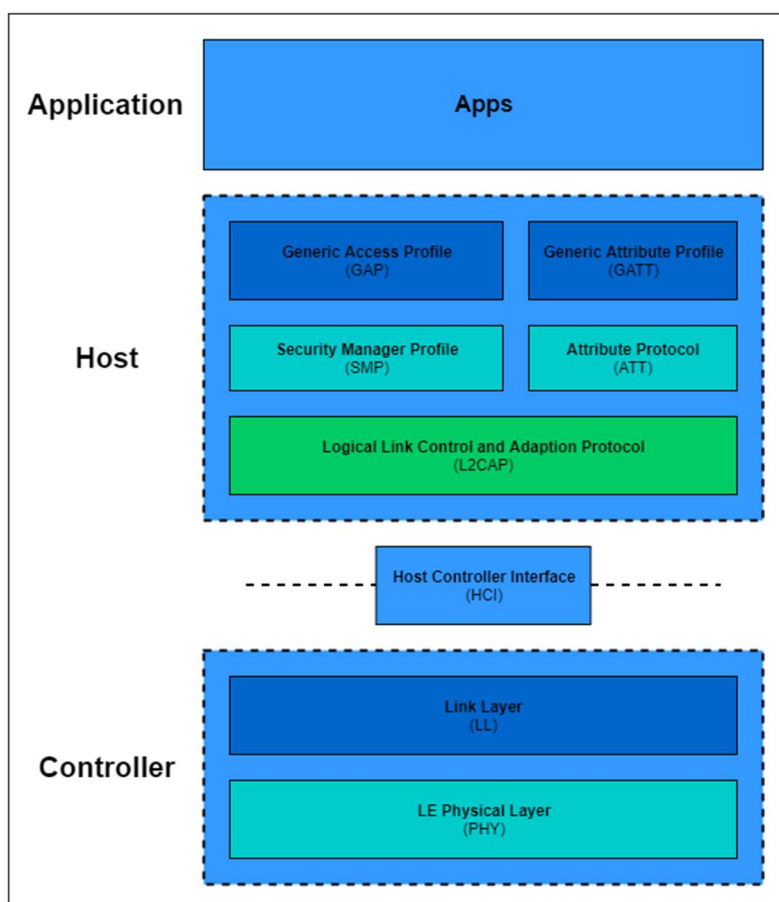
Figure 2.1: The BLE Protocol Stack

The Application layer contains the application logic, interface, and data processing required to satisfy the given use case. As such, the content of the application is almost entirely context dependent.

The controller contains the following layers:

- Physical Layer (PHY) – the physical layer contains the analog communications circuitry which modulates and demodulates signals, transforming them into digital symbols. As discussed in Section 2.1.1, the BLE radio uses the 2.4GHz ISM band to communicate. This frequency band is divided into 40 channels, 37 of which are used for connection events and the remaining 3 are used for advertising. The advertising process will be discussed further in Section 2.2.2. The BLE radio employs frequency hopping spread spectrum to minimize the amount of radio interference, and Gaussian Frequency Shift Keying (GFSK) to encode the bitstream. The modulation rate for BLE is 1 Mbit/s.

- Link Layer (LL) – the link layer directly interfaces with the PHY and is typically comprised of custom hardware and software. The link layer is responsible for meeting the timing requirements defined by the specification. This is the only real-time constrained layer in the entire BLE stack and, as a result, its complexity is hidden behind a standard interface – the

Host Controller Interface. The link layer hardware implements the expensive but automatable functionality such as Random number generation and AES encryption, while the software controls the state of the radio.

The Host contains the following layers:

- Logical Link Control and Adaption Protocol (L2CAP) – The L2CAP is responsible for taking multiple protocols from the upper layers of the stack and combining them to produce a standard BLE packet format. L2CAP also provides fragmentation and recombination of packets that exceed the 27-byte payload limit.
- Security Manager Profile (SMP) – The SMP is both a protocol and a series of algorithms which are used to generate and exchange security keys which enable secure communication over an encrypted link, remote device identification, and the ability to hide the public Bluetooth address. The SMP defines two roles: the initiator, or GAP central, and the responder, or GAP peripheral.

The Host also provides the Generic Access Profile (GAP), the Attribute Protocol (ATT), and the Generic Attribute Profile (GATT). These will be discussed in Sections 2.2.3, 2.2.4, and 2.2.5 respectively.

2.2.2 Generic Access Profile

The generic access profile (GAP) outlines the procedures for connections and advertising in Bluetooth, as well as defining the various device roles. The GAP describes four roles each with specific behaviours:

- Broadcaster – a device in the broadcaster role can only broadcast data via advertisements and cannot establish a connection with other devices.
- Observer – the observer role is designed to receive data from the broadcaster.
- Central – a device in the central role is in charge of initiating and managing connections with other devices.
- Peripheral – a device in the peripheral role is designed to use a single connection to a device in the central role.

The central and peripheral roles dictate that the device's controller support the master and slave roles respectively. A device may only adopt a single role at a time, however it is possible for several roles to be supported.

2.2.3 Advertising

Operating within the 2.4GHz ISM band, BLE has 40 1MHz-wide channels, with 2MHz between each one. Of these 40 channels, 3 are reserved for advertising [22]. These channels are located at the start, middle and end of the spectrum. This reduces the likelihood of interference from other 2.4GHz technologies such as Wi-Fi. When in advertisement mode, a peripheral sends out advertising packets at a fixed interval with a pseudo-random offset. The fixed interval is set between 20ms and 10.24 seconds, at intervals of 0.625ms [23]. This interval is then offset by a pseudo-random value between 0 and 10ms to reduce interference between BLE devices [22]. When in advertising mode, the peripheral cycles through the channels.
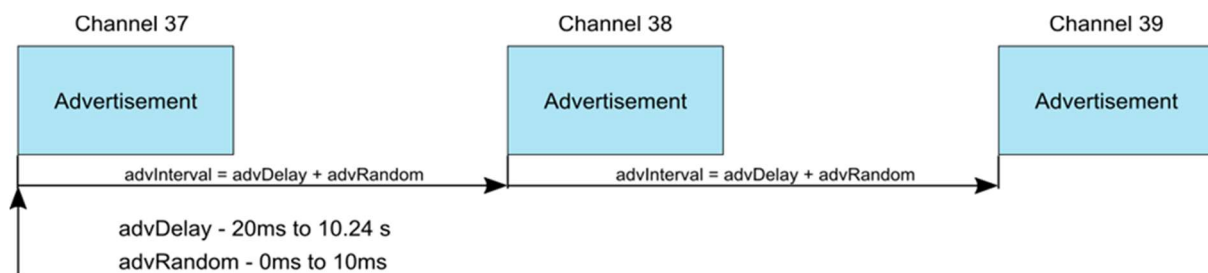

Figure 2.2: Periodic Advertising [22]

The advertising packets are used by peripherals to broadcast their presence to nearby centrals.

2.2.4 Attribute Protocol

The Attribute Protocol (ATT) is a client/server protocol which defines two roles: a client and a server. A BLE device may be a client, server, or both. The client requests data from the server and the server sends data to the client. The ATT enables the server to expose a set of attributes, and their associated values to a client. The attributes may then be discovered, read, and written by the client, and indicated, and notified by the server.

The ATT is a strictly sequenced protocol. If a client's request is pending, no more requests may be sent until the response is received and processed. [This is discussed further in section x.y.]

2.2.5 Generic Attribute Profile

The Generic Attribute Profile (GATT) provides a service framework on top of the ATT. The GATT defines the procedures and formats of services and their characteristics. The GATT profile is designed for use by an application to allow communications between a client and server. The server contains attributes, and the GATT profile defines how the ATT is used in order to discover, read,

write, and obtain indications, and notifications of these attributes. The GATT profile also outlines how to use the ATT in configuring the broadcast of these attributes.

Devices that implement the GATT profile have two defined roles:

- Client – The client initiates commands and requests to the server and can receive responses, indications, and notifications sent by the server.
- Server – The server receives requests and commands, and can send the client responses, indications, and notifications.

A device may act as both a client and a server at the same time.
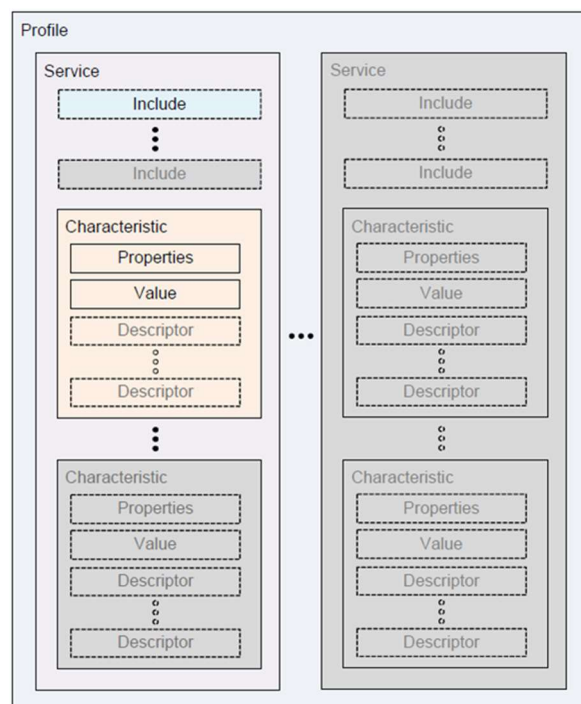
Figure 2.3: The GATT profile hierarchy [24]

At the top level of the hierarchy is a profile. The profile contains one or more services which are necessary to fulfil a use case. Each service is comprised of multiple characteristics, or references to other services. Characteristics consist of a type, value, and a set of properties which detail the interactions supported by the characteristic. Characteristics may also include descriptors which provide extra information. The GATT groups these services to reflect a single aspect of the behaviour of a device. For example, a heart rate monitor has a heart rate service which contains the heart rate measurement, and body sensor location as characteristics.

2.2.6 Universally Unique Identifier

A universally unique identifier (UUID) is a 128-bit number which has a high probability of being globally unique. The UUID is not unique to BLE. The UUID specification is from ISO/IEC 9834-8:2005, and they are used in several protocols. However, as the BLE payload is only 27 bytes, BLE specifies two other forms of UUID: a 16-bit and a 32-bit UUID. These formats are only allowed to be used for UUIDs defined in the Bluetooth specification. The Bluetooth Special Interest Group provides shortened UUIDs for all of the types, services, and profiles that it defines and specifies.

2.2.7 Attributes

Attributes are the smallest data entity defined by both GATT and ATT. Attributes are addressable pieces of information that contain user data. As GATT and ATT only work on attributes, all information in BLE must be organised in this form. Conceptually, attributes are located on the server, and may be accessed for reading, or for writing, by the client.

Every attribute contains information about itself as well as the user data. The attribute fields are:

- *Handle:* The handle is a 16-bit identifier for each attribute on a GATT server. The handle is the address for the attribute and is a constant.
- *Type:* The type is a UUID. It can be a 16-, 32-, or 128-bit UUID, and is used to determine the type of data stored in the attribute.
- *Permissions:* The permissions define the legal ATT operations for an attribute. ATT and GATT define the following permissions:
  - *Access Permissions:* None; Readable; Writable; Readable and Writable
  - *Encryption:* No encryption required; Unauthorized encryption required; Authenticated encryption required
  - *Authorization:* No authorization required; Authorization required
- *Value:* The actual data stored in the attribute. The type of this data is unrestricted, and it has a maximum length of 512 bytes.

2.3 Constrained Application Protocol

The Constrained Application Protocol (CoAP) was designed to be a specialized web transfer protocol for use with constrained nodes operating within constrained networks. CoAP is easily translated into HTTP due to the similarity in their interaction model, which will be discussed in Section 2.3.3. CoAP was employed in the prototype for communication from the client to the gateway. This section offers a brief discussion of CoAP.

2.3.1 Message Structure

CoAP messages are encoded in a binary format and transported over UDP. Each message occupies the data section of one UDP datagram. The message begins with a fixed-length 4-bte header, followed by a Token value between 0 and 8 bytes in length. After the Token value is the sequence of CoAP options in the format of Type-Length-Value. The final element of the message is the payload which takes up the rest of the datagram.
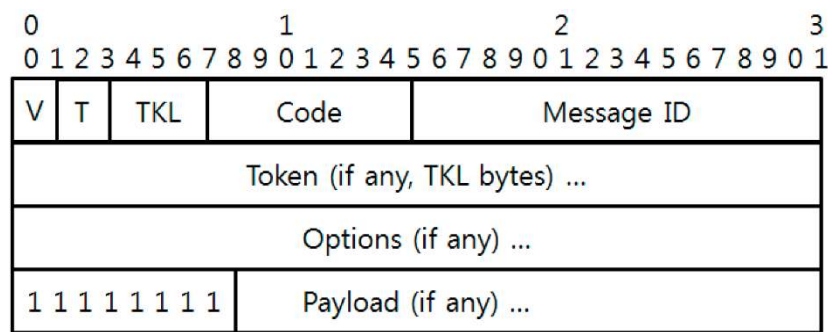

Figure 2.4: CoAP Message Structure [25]

CoAP uses the following header fields:

- Version (V): 2-bit unsigned integer, indicating the CoAP version number.
- Type (T): 2-bit unsigned integer, indicating if the message is Confirmable (0), Non-confirmable (1), Acknowledgement (2), or Reset (3).
- Token Length (TKL): 4-bit unsigned integer, indicating the length of the Token field (0-8 bytes)
- Code: 8-bit unsigned integer divided into a 3-bit class (the 3 most significant bits), and a 5-bit detail (the remaining bits). The code is documented in the form "c.dd" where "c" is a digit from 0 to 7 representing the class, and "dd" are two digits from 0 to 31 representing the detail. There are four primary class values: request (0), successful response (2), client error response (4), and server error response (5). 0.00 is the special case for an empty message.
- Message ID: 16-bit unsigned integer in big-endian order. This is used to detect message duplication and to match ACK and RST messages to their respective CON and NON messages.

The Token value follows the header. The Token is used to correlate requests and responses. The Token is followed by zero or more options. The options may be followed by either the end of the message or the payload.

2.3.2 Messaging Model

While CoAP provides a similar request/response pattern to that of HTTP, it does so asynchronously over UDP. CoAP does this using a layer of messages which provide optional reliability. There are four types of messages in CoAP:

- Confirmable (CON)
- Non-Confirmable (NON)
- Acknowledgement (ACK)
- Reset (RST)

A CON message is used for reliable messaging. When a client sends a server a CON message, it will retransmit until it receives an ACK message with the same ID from the server.
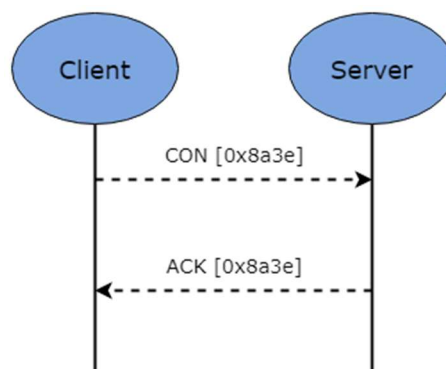
Figure 2.5: Reliable message transport

CoAP uses a timeout to control message retransmission, calculated according to the formula:

$$TIMEOUT\_VAL * RANDOM\_FACTOR$$

Where TIMEOUT_VAL is defaulted to 2 seconds, and RANDOM_FACTOR is defaulted to 1.5 [26]. If a server is unable to process the message it responds with an RST message instead of an ACK message.

A NON message is used for unreliable messaging. A NON message does not need to be ACKed. However, it must contain a message ID to detect retransmission.
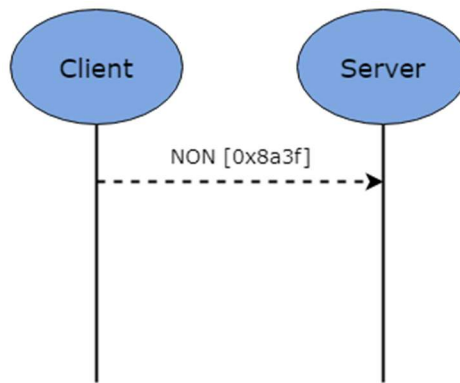
Figure 2.6: Unreliable message transport

If the server fails to process the NON message, it responds with an RST message.

2.3.3 Request/Response Model

The CoAP request/response semantics are carried in CoAP messages. The messages include a Method Code when carrying a request, and a Response Code when carrying a response. Information such as the URI, and Payload are carried in CoAP options, and a Token, or 'request ID', is used to match requests and responses independent of the underlying messages.

Requests are executed using CON or NON messages, and, if the requested data is immediately available, the response piggybacks on the resulting ACK message. The ACK message includes a response code. Figure X.Y shows two client-server exchanges, one successful and one failed due to a 4.04 (Not Found) error.
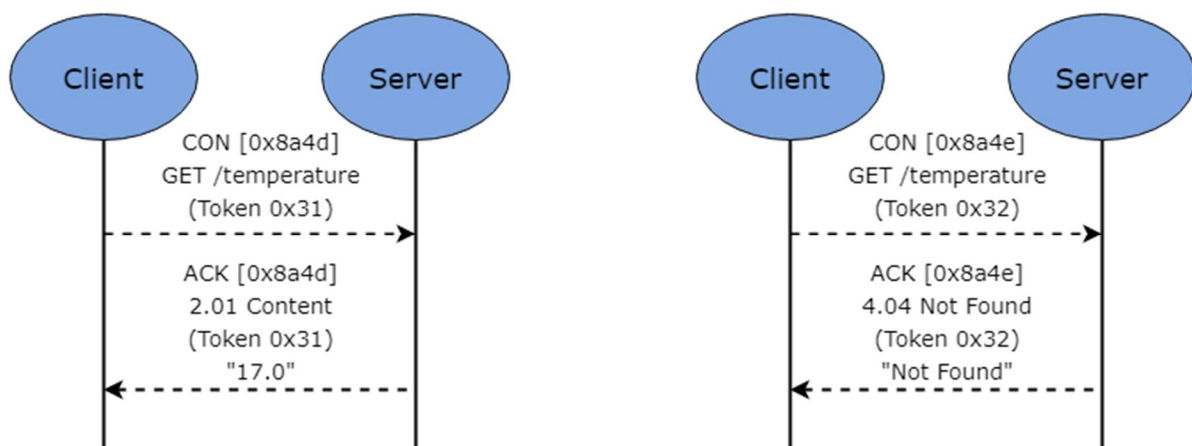
Figure 2.7: CoAP Requests with Piggybacked Responses – One successful, One failed

If the information requested is not immediately available, the server responds with an empty ACK message. When the information becomes available, the server then sends it as a CON message

containing the Token value from the original request, as shown in Figure 2.8. This is known as 'separate response'.
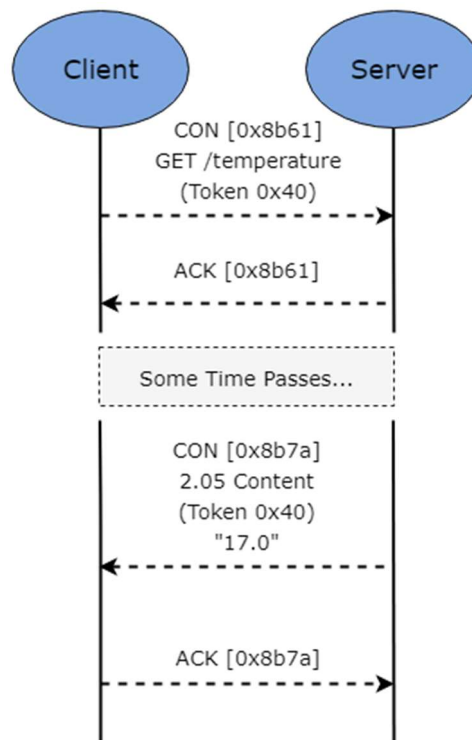
Figure 2.8: Separate Response

If a request is made using a NON message, then the server may send the response using a new NON message. However, the server may also use a CON message.

CoAP provides the four standard methods of HTTP (GET, PUT, POST, and DELETE) as well as OBSERVE. The OBSERVE method allows a client to receive notifications on the state of a resource on a CoAP server as it changes. The OBSERVE method is reflected in a CoAP message as an option.
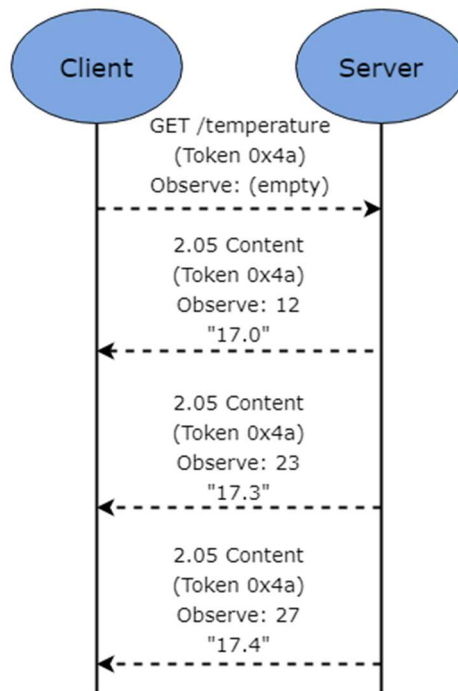
Figure 2.9: Observing a Resource

As shown in Figure 2.9 the server uses the token of the initial request to semantically connect each notification that it sends the client.

2.4 WebSocket Protocol

The WebSocket Protocol (WSP) was standardized by the IETF in 2011 to enable two-way communication between a client and server. Previously, applications requiring bidirectional client-server communication implemented polling over HTTP which leads to great overheads. WSP offers an alternative, by allowing a single TCP connection to be used for traffic in both directions. WSP was designed to replace existing bidirectional HTTP technologies, so it operates over HTTP ports 80 and 443. This section will provide an overview of the workings of WSP.

2.4.1 Opening Handshake

The first stage in WSP is the handshake. The client must send the server the opening handshake to initiate the connection. The server uses information obtained from the client's handshake to generate its own handshake. The client handshake contains the following fields:

- An HTTP version 1.1 or higher GET request, which includes a request-URI.
- A *Host* header field.
- An *Upgrade* header field containing the value "websocket".
- A *Connection* header field containing the value "Upgrade".

- A *Sec-WebSocket-Key* header field containing a base64-encoded value, which decodes to a 16-byte value.

- A *Sec-WebSocket-Version* header field with a value of 13.

- An optional *Origin* header field.

- An optional *Sec-WebSocket-Protocol* header field containing a list of values specifying the protocols that the client would like to speak in order of preference.

- An optional *Sec-WebSocket-Extensions* header field with a list of values indicating the client's desired extensions.

Once the client establishes this connection to the server, the server must complete the following steps to accept the connection:

- For secure connections, perform a TLS handshake over the connection. Should this handshake fail, close the connection.

- Optionally, the server may perform further client authentication.

- Optionally, the server may also redirect the client using a 3xx status code.

Upon completion of these steps, the server must establish the following information:

- *Origin*: The *Origin* field in the client's opening handshake provides this.

- *Key*: The value from the *Sec-WebSocket-Key* header field of the client's opening handshake is used to create the server's handshake to indicate that the connection has been established.

- *Version*: The *Sec-WebSocket-Version* header field in the client's opening handshake is compared to the versions of the WebSocket protocol understood by the server. If the server does not understand the requested version, it responds with an appropriate HTTP error code.

- *Resource Name*: This value is the identifier for the service provided by the server. If the server provides multiple services, then the value is derived from the *Request-URI* sent by the client.

- *Subprotocol*: The subprotocol value may be either a single value representing the subprotocol the server is prepared to use or null. This value must be derived from the value provided in the client's opening handshake's *Sec-WebSocket-Protocol.*

- *Extensions:* A list of zero or more values representing the protocol-level extensions that the server is prepared to use.

Once this information is established, the server replies to the client's opening handshake with a valid HTTP response. The response uses the RFC2616 specified 101 response code indicating the changing

of protocols. This response completes the client-server handshake and establishes the WebSocket connection.

2.4.2 Data Framing

In WSP, the sent data is formatted into frames. Each frame that the client sends must be masked, and the server must close the connection upon receiving an unmasked frame. The server does not mask any of the frame that it sends to the client. The client must close the connection upon receiving a masked frame.

The base framing protocol defines a frame with an *opcode,* a *payload length*, and designated locations for *extension data* and *application data*. *Extension data* and *application data* form the payload data.
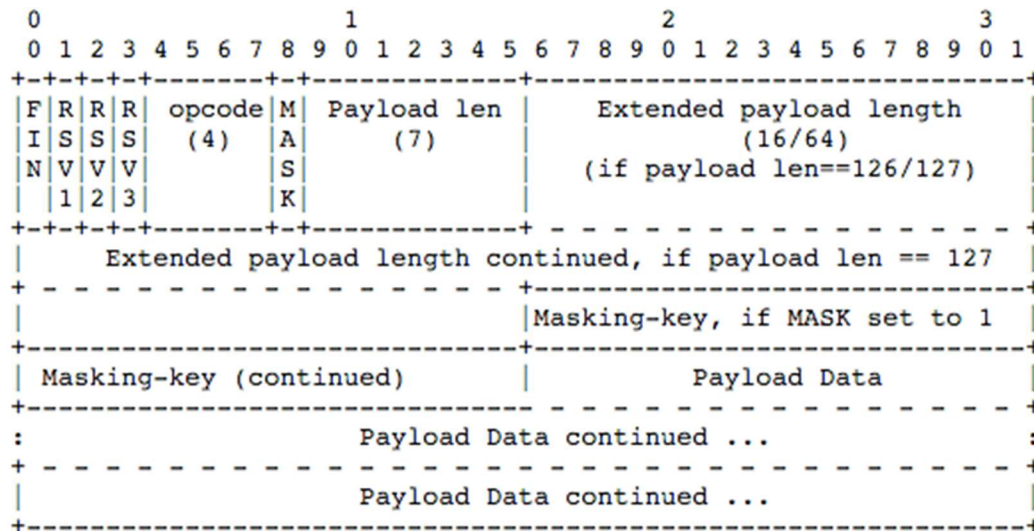
```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-------+-+-------------+-------------------------------+
|F|R|R|R| opcode|M| Payload len |    Extended payload length    |
|I|S|S|S|  (4)  |A|     (7)     |             (16/64)           |
|N|V|V|V|       |S|             |   (if payload len==126/127)   |
| |1|2|3|       |K|             |                               |
+-+-+-+-+-------+-+-------------+ - - - - - - - - - - - - - - - +
|     Extended payload length continued, if payload len == 127  |
+ - - - - - - - - - - - - - - - +-------------------------------+
|                               |Masking-key, if MASK set to 1  |
+-------------------------------+-------------------------------+
| Masking-key (continued)       |          Payload Data         |
+-------------------------------- - - - - - - - - - - - - - - - +
:                     Payload Data continued ...                :
+ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - +
|                     Payload Data continued ...                |
+---------------------------------------------------------------+
```

Figure 2.10: Base Data Framing Protocol [27]

The data frame fields are as follows:

- *Fin:* 1 bit. This indicates if the frame is the final fragment of the message.
- *RSV1, RSV2, RSV3:* 1 bit each. Each bit must be set to 0 unless an agreed upon extension provides a meaning for non-zero values.
- *Opcode:* 4 bits. This defines the interpretation of the *Payload Data*. The possible values and their meaning are provided in Appendix 1.
- *Mask:* indicates whether the P*ayload Data* is masked or not. If the value is 1, the masking key is also contained in the frame (in *Masking-key*).
- *Payload Length:* 7 bits, 7+16 bits, or 7+64 bits. This encodes the length of the payload.
- *Masking-key:* 0 or 4 bytes. If the *Mask* bit is set, this contains a 4-byte masking key.

- *Payload data:* x+y bytes. The *Payload Data* is the *Extension Data* and the *Application Data*.

- *Extension data:* x bytes. This is 0 bytes unless an extension has been agreed upon. If an extension has been agreed upon, the extension specifies the length.

- *Application data:* Application data occupies the remainder of the frame.

Any messages sent between the client and server are translated into this frame format. WSP uses fragmentation to enable the sending of messages of unknown length. To do this, servers or intermediaries choose a reasonable buffer size, and when the buffer is full, its contents is written to the network in the specified format.

2.4.3 Closing Handshake

To close the connection cleanly, the WSP specifications outline a closing handshake. The closing handshake is the desired means of closing a connection, however, in extreme situations, an endpoint may close by any means. As discussed in section x.y, the opcode 0x8 is used to indicate a Close control frame. When an endpoint has both sent and received a Close control frame, the endpoint should then close the underlying TCP connection.

The endpoint may also specify a closure reason using a pre-defined status code. The defined status codes and their meanings are provided in Appendix 2.

Chapter 3 Design

This section will offer a high-level discussion of the proposed design, looking at the core components and the central algorithms within the system.

3.1 Design Requirements

Before looking at the system design, the requirements for the system must be specified. The primary requirements of the system are for it to be generic and scalable. This will force the system design to abstract away from specific protocols and technologies and focus purely on the central concept of deploying a process to translate short-range wireless technologies into RESTful protocols.

The secondary requirements of the system are to ensure that the system is consistent and offers the user transparency (i.e. the state of the system is always visible).

3.2 The Proposed System

The diagram below shows the generic architecture employed in this project. The sections that follow will discuss each component of the architecture individually in further detail.
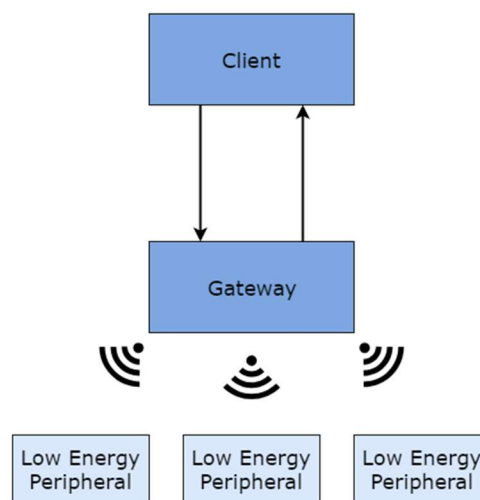
Figure 3.1: Generic System Architecture

3.2.1 Client

The role of the client is to interface with the gateway. The client may be fulfilled by several actors such as a cloud system aggregating data, or a web-client. The primary function of the client is to proxy Bluetooth requests, translating them from HTTP to CoAP to communicate with the gateway. The design of the system ensures that a client can be connected to multiple gateways concurrently.

3.2.2 Gateway

The role of the gateway in the system is to handle CoAP requests and, where appropriate, translate them into Bluetooth interactions. As BLE is not IP compatible, the system required a URI scheme which could translate the hierarchical structure of BLE into a RESTful style. The developed URI scheme is discussed in Section 3.3. A graphical representation of the scheme and the related BLE components is provided in Appendix 3. To dynamically create URIs as devices moved in and out of range of the proxy, the author developed an algorithm, referred to as the Dynamic Naming System, which is discussed in Section 3.4.

3.3 URI Scheme

For the interactions with the web-client to be translated into CoAP, a URI scheme had to be specified. The URI scheme follows the RFC7252 CoAP specifications [26] in implementing the '/.well-known/' path prefix from RFC5785 [28] to identify well-known locations in the namespace of the host. In the case of the prototype, the '/.well-known/' path retrieves all of the observed peripherals.

The URI scheme for device interactions begin with the device id. The endpoint of '/:deviceID' executes a Bluetooth connection request on that device. The endpoint of a device id followed by 'exp' (i.e. '/:deviceID/exp') results in an exploration of that device. An exploration finds all of the available services on the device. The service level interactions are requested by following the device id with a service id, '/:deviceID/:serviceID'. When followed by 'getChars', the characteristics of that service are retrieved. Characteristic interactions, such as read and subscribe, are available at '/:deviceID/:serviceID/:characteristicID/read' and '/:deviceID/:serviceID/:characteristicID/sub' respectively.

The goal of the URI scheme design is to mirror the semantics outlined in the BLE GATT. This was achieved by reflecting the descending hierarchical state traversal observed when going from device to service to characteristic in the URI scheme from left to right. Alongside this hierarchical mapping, several endpoints were created to reflect specific Bluetooth interactions, such as read and subscribe. A diagrammatic representation of the URI scheme alongside the BLE hierarchy can be seen in Appendix 1.

3.4 Dynamic Naming System

The dynamic naming system allows the gateway to create and validate URIs, or paths, relating to the observed Bluetooth low energy peripherals in its environment. This circumvents the issue of Bluetooth's IP incompatibility. The author presents the three-part algorithm in the sections below. The algorithm is presented in two parts: dynamic path generation, and path (or URI) interpretation.

3.4.1 Dynamic Path Generation

The first aspect of the algorithm involves generating URI paths according to the scheme presented in Section 3.3. This enables the gateway to offer the client dynamic endpoints generated in relation to the peripherals within range of the gateway. The pseudocode of the steps taken in this process are provided below.

Dynamic Path Generation – Pseudocode
FOR every peripheral observed
enumerate and store the peripheral
IF peripheral advertises services
FOR every service advertised
get characteristics for advertised service
store services and their characteristics in the peripheral

As the Bluetooth advertising packets are detected by the gateway's Bluetooth central, they are enumerated. The enumeration process serves to provide an id when handling requests. If the advertising packet contains information about services provided by the peripheral, the characteristics of each service are retrieved. Finally, the services and their characteristics are stored in a JavaScript Object Notation (JSON) hierarchy which can easily map to the client's RESTful requests.

3.4.2 Path Interpretation

The second aspect of the algorithm is the process by which the gateway handles incoming requests from the client. The gateway must take the request and verify it against the paths that it has generated before it can perform the desired action. The pseudocode of the steps involved in this process are provided below.

Path Interpretation - Pseudocode
get requested path
interpret requested path
IF requested peripheral exists **AND** contains requested service
IF requested service contains the requested characteristic
perform requested Bluetooth interaction on characteristic

As requests come in, they are parsed according to the system's URI structure presented in Section 3.3. The URI's start with a peripheral id. This id is the number assigned during enumeration.

3.4.3 Path Maintenance

The final part of the algorithm is the process by which the gateway maintains its endpoints, or paths. Unlike traditional RESTful endpoints, the resources of the proposed system are subject to expiry. In the case of RPM, a patient wearing a heart rate monitor may walk out range of the gateway. In such cases, the system requires a mechanism for reflecting this.

The solution to this issue is provided below in pseudocode. The gateway maintains a list of peripherals along with the last time that an advertising packet was received. This timestamp is used by the system to remove expired endpoints according to a predetermined threshold.

Periodic List Refresh - Pseudocode
FOR every peripheral observed
IF THRESHOLD_VALUE has elapsed since observation
remove peripheral

This reduces the risk of unavailable peripherals being displayed to the user. The current value of the threshold is 22000ms. This value ensures that a peripheral which is advertising with the maximum interval, 10.24s, would have to be absent for two advertisements. However further testing could be done to determine an optimal value.

Chapter 4 Implementation

To test the feasibility of the proposed system, the author developed several prototypes following an iterative development process.

4.1 Iterative Designs

The author followed an iterative development framework in implementing the system. Throughout this process, several takes on the high-level design were considered before arriving at the final implementation. This section will discuss some of these iterations.

4.1.1 MQTT-CoAP Client

The first iteration of the design saw the client utilising two separate protocols depending on the type of interaction. The CoAP client was to handle the 'instantaneous' requests (i.e. requests which result in a single response), while the MQTT client would handle the 'persistent' requests (i.e. requests which result in multiple responses). This design is shown in Figure 4.1.
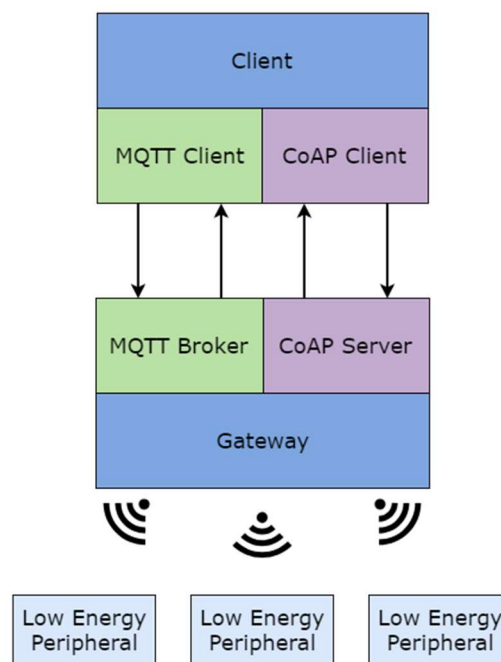

Figure 4.1: First Iteration of the Design

This design was semantically sound: a publish-subscribe protocol (MQTT) to transmit the recurring subscription data, and a request-response protocol (CoAP) to obtain the instantaneous data. However, it had several drawbacks. The introduction of MQTT brought with it several other technologies. The MQTT broker required a MongoDB backend, and a second URI scheme to assign to topics.

While this implementation was attempted, the additional technologies proved to over-complicate the system. As a result, the debug process was significantly slowed. The second iteration sought to simplify the system.

4.1.2 CoAP Client with Polling

The second iteration removed MQTT from the system entirely. On review of the background research conducted, an alternative for the publish-subscribe model of MQTT was found in CoAP's OBSERVE method. The revised design is shown in Figure 4.2.
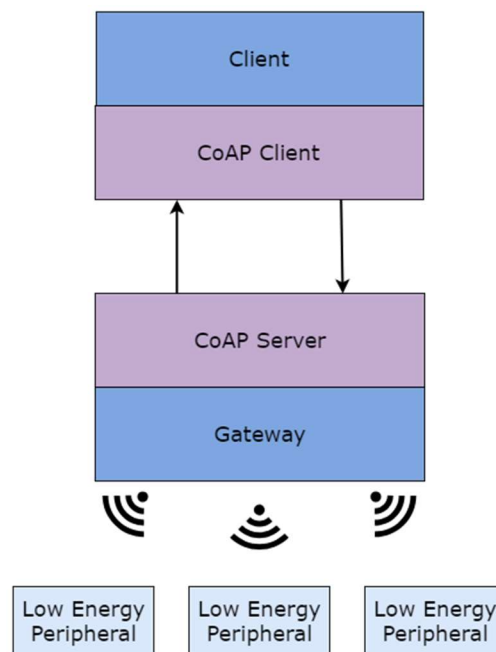

Figure 4.2: Second Iteration of the Design

The second iteration used CoAP for all of the Client-Gateway interactions. The 'instantaneous' requests were made using CoAP *GET* requests, and the 'persistent' requests were made using CoAP *OBSERVE* requests. Utilising the CoAP *OBSERVE* method simplified the implementation, however one issue remained. Subscribing using the *OBSERVE* method facilitated a publish-subscribe model of sorts between the Client and the Gateway. However, as the requirements for the implementation included a GUI served over a HTTP web-client, the subscription data needed to be pushed to the user. While this iteration was able to fulfil most of the requirements, it was unable to push data from the client to the user. The final iteration improved upon this.

4.2 Implemented System Overview

This section will provide an overview of the prototype implemented. The core components of the system are as shown in Figure 4.2. This iteration has a slightly different client however. The client is able to push values to the user in the case of a subscription. This is discussed in Section 4.2.1.

As shown in Figure 4.1, the primary components of the system are the Client and Gateway (including the Bluetooth Central). The Client behaves as the CoAP client that interacts with the Gateway's CoAP server. In the prototype implemented the requests made by the client are made by the user through a GUI. The user interacts with the GUI over HTTP, which is translated into CoAP and forwarded to the Gateway. The Gateway receives these CoAP requests and, where appropriate, executes the relevant BLE interactions. The following sections will discuss each of these components in detail.

4.2.1 Client

The Client in the high-level design is employed to translate HTTP requests to CoAP requests which are then forwarded onto the Gateway. In a real-world deployment of the system, this role may be fulfilled by a centralised cloud aggregation system. In the implemented prototype however, HTTP requests are received from a GUI served to the user over HTTP. The user's interactions with the GUI generate the HTTP requests which are translated and forwarded. The Client design is shown in Figure 4.3.
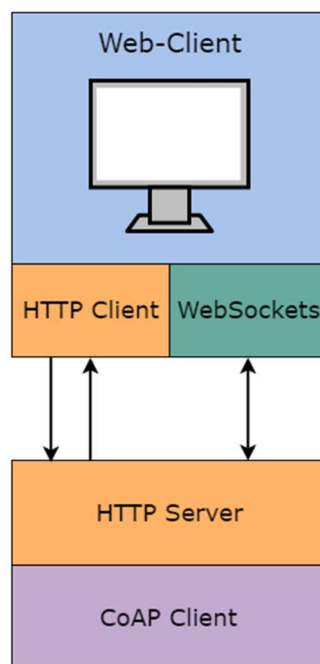

Figure 4.3: Detailed View of the Client Design

The 'instantaneous' interactions are sent over HTTP. When a user requests a subscription to a peripheral however, the request to the server is made using WebSockets. This allows the Client server to push data out to the user. The protocol stack for the Client server is shown in Figure 4.4.
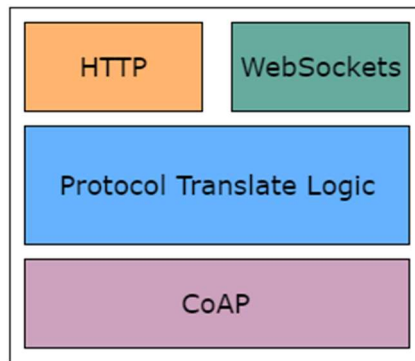
Figure 4.4: Client Protocol Stack

The Client receives requests from the user through HTTP or WebSockets. These requests are translated into CoAP and forwarded to the Gateway. When the response is received, it is translated from CoAP and back into its protocol of origin.

The server for the Client was written in NodeJS, using the expressJS and node-coap libraries. The ExpressJS library is used to create an express app which provides routing, and middleware. The node-coap library is used to make CoAP requests to the user specified gateway. The client also uses websockets to facilitate a bidirectional data stream for subscriptions. The static pages served to the user use the HTML 5 websockets API through inline javascript, while the server uses the node websocket-stream library. The subscription data is graphed using the ChartJS library.

4.2.2 Gateway

The Gateway in the high-level design is responsible for handling CoAP requests and interacting over BLE. The prototyped Gateway fulfils these responsibilities and is capable of dynamically creating endpoints depending on its environment as discussed in Section 3.4. The protocol stack for the Gateway is shown in Figure 4.5.
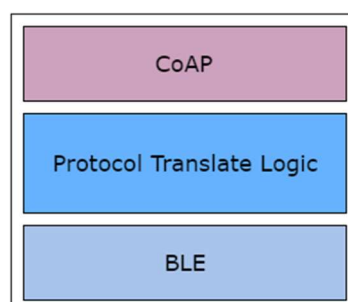

Figure 4.5: The Gateway Protocol Stack

As shown in the protocol stack, the Gateway receives requests over CoAP, and translates them into BLE interactions. The translation process is quite different to that of the Client however. Due to BLE's

lack of IP compatibility, the translation from CoAP to BLE is not straightforward. The algorithm discussed in Section 3.4 is at the crux of the proposed solution. Once the server starts, it begins to look for BLE advertising packets. Each observed packet is enumerated and stored. The server begins to use the observed packets, and their newfound ID to construct paths. The constructed paths are stored alongside the device to which they refer. This means of storage, along with the URI scheme discussed in Section 3.3, ensure the verification process is extremely quick.

The incoming CoAP requests are verified against the knowledge base of paths. If the path is found to be valid, then the requested BLE interaction is executed and the result forms the body of the CoAP response. However, if the path is invalid, the server will respond with an appropriate error code.

The Gateway was implemented in NodeJS, using the node-coap library for the CoAP server, and noble for the BLE interactions. The asyncjs library was also used to provide asynchronous versions of the server logic. The Bluetooth requests were made by an ASUS Bluetooth adapter with the WinUSB driver as stipulated by the noble library.

4.3 Difficulties Encountered

During the development of the prototype, the author encountered several issues. These issues will be discussed in this section, along with issues that the author predicts may be encountered by the project in the future.

4.3.1 Single Persistent Connection

In developing the prototype, several issues arose when routinely connecting and disconnecting with peripherals to facilitate multiple concurrent device interactions. These issues resulted in slower connection times, as well as an increase in library-level errors. The final iteration of the prototype avoids this issue using a narrowed scope. When the user selects a peripheral through the web-client, the gateway maintains a connection with this single peripheral. While this removes the potential for concurrently observing multiple peripherals, it enabled more efficient and consistent Bluetooth interactions.

4.3.2 Temperamental Libraries

Many of the problems encountered were due to existing issues in the noble library. The issues have been reported multiple times on the github of the library, however they are without solution. The issues include falsely reporting existing connections between the central and peripherals, and connection requests to observed peripherals resulting in an 'unknown peripheral' message. These

issues significantly hindered the consistency of the system, and future work may be done on a more consistent BLE API.

4.3.3 BLE Sequencing

As discussed in Section 2.2.4, ATT is the BLE client-server protocol, and it is strictly sequenced. This has not led to any issues during the development of the prototype, however it may cause difficulties in the future. Any BLE requests made by the client must be completed before another request will be processed by the BLE device. This will most likely require a client-side request queuing system. In such a case it will be necessary for the client to include a CoAP server. This would allow the gateway to respond to the request with an empty acknowledgement upon receipt of the request, before sending the actual response as a later message. This would behave the same to a separate response as discussed in Section 2.3.3.

Chapter 5 Evaluation

This chapter offers a reflection on the system design and the implemented prototype in light of the requirements outlined in Section 3.1.

5.1 Primary Requirements

The primary requirements for the system were to provide a scalable and generic solution. This is delivered by the system. The algorithms detailed in Section 3.4 succeed in allowing the system to remain responsive to its environment. The Dynamic Name Spacing offers the end-user an insight into the environment of the Gateway at a delay of the predetermined threshold value (in the case of the prototype 22000ms). This algorithm is scalable, and, by proxy, so is the system. While the algorithm deployed is a BLE specific implementation, the algorithm itself has no such restriction. To expand the algorithm into the space of alternate wireless technologies would require two changes:

- A separate module would be needed to discover devices over this wireless technology. This module would perform the first part of the algorithm, as discussed in Section 3.4.1.
- An agreed upon name-spacing structure. The prototype employs a name spacing structure of '/:deviceID/:service/:characteristic' (a graphical representation of this structure can be found in Appendix 3). As the data representation reflected in this structure is specific to BLE, a new structure would be required for a new technology.

Aside from these changes, the body of the algorithm remains unchanged. On this basis, it is reasonable to conclude that the deployed system offers a generic and scalable solution.

5.2 Secondary Requirements

The secondary requirements for the system were for the system to operate in a consistent and transparent manner. The performance of the system was poor when assessed under these criteria. The issues discussed in Section 4.3 are the primary factors. The issues on with the noble library severely hindered the system's consistency, and transparency. While the system may succeed in detecting the local devices and displaying them to the user, issues with the noble library may prevent the user from being able to interact with the devices.

Chapter 6 Conclusion

This section will provide a summary of the report, future directions, and concluding with a review of the overall outcome.

6.1 Report Summary

The report provided a formalisation of the research, development, and review journey undertaken by the author to answer the research question:

"Is it feasible to translate short-range technologies into

a RESTful protocol in a generic fashion?"

The first chapter introduced the problem, its context, and the motivations for addressing it. The author documented their initial thoughts on the solution requirements, and their aims for the proposed system. The preliminary research conducted was presented, as were the decisions it informed. From this research, the author found BLE to be the most appropriate wireless technology due to its widespread penetration and its energy efficiency. CoAP was chosen as the IoT protocol due to its wide set of features, its similarity to HTTP, and its compact message format. The author followed these decisions with an in-depth discussion of the employed technologies, providing the reader with a sufficient base to understand the report that followed.

Following this discussion, the high-level design for the proposed system was detailed. The author discussed the core components, and the ways in which the design provided a generic solution. The algorithms at the core of the design were also subject to high-level discussion. The analysis of the system components provides the reader with the required information before the discussion of the implementation.

The implementation chapter offered an analysis of the specifics involved in putting the design into practice. The earliest iterations of the prototype were presented along with their shortcomings. The author then outlined the various programming languages, and libraries utilised during the final prototype development, as well as an overview of the prototype technology stack. Following this, the issues encountered were discussed, along with the employed circumventions.

Finally, the developed prototype was evaluated. The prototype was found to have fulfilled the outlined scope of the project.

6.2 Future Directions

The prototype implemented proved the feasibility of the proposed system, however many expansions could be made. The research conducted would benefit from the integration of alternative wireless technologies. While the prototype successfully offers a generic solution for BLE to CoAP translation, the data representation differs across various wireless technologies, and further work could be done on an even more general dynamic naming system.

Further work could also be done on bypassing the limitations highlighted by the author in Section 4.3. The prototype's gateway is restricted to a single connection at the BLE level. While this is acceptable for the scope of the prototype, it would not be appropriate for use in a real-world setting. Similarly, the implementation of a cloud aggregation system as a client would certainly serve to create a more robust system.

Future study may be done into a case-by-case threshold value for the path maintenance algorithm presented in Section 3.4.3. This would allow a specific timeout for each peripheral calculated using the peripheral's fixed advertising interval. The formula for such a calculation may be:

$$T = N*(I + 10)$$

Where:

- *T* is the threshold value (ms)
- *N* is the leniency constant (the number of intervals to wait before removing a path)
- *I* is the fixed advertising interval (ms)
- 10 is the constant maximum random offset from the Bluetooth specifications (ms).

This extension to the algorithm would greatly further the scalability of the design while also providing a more transparent system.

6.3 Outcome

The work presented displays a generic approach to translate short-range wireless technology into a style that is addressable over a RESTful protocol. By meeting the requirements set out in Section x.y.z, the prototype showed that the design presented is a scalable and generic solution to communicate with short-range technologies. The system designed, and then implemented, allows for the communication between a client and low-energy peripherals via a gateway using an IP designed for constrained networks. The prototype implemented the client-gateway communication using CoAP, and the gateway-peripheral communication using BLE, however the high-level design

has no restriction to these technologies. As such, the design is scalable, and generic, facilitating the seamless integration of alternative communication protocols.

In conclusion, the design proposed in this report succeeds in providing a scalable and generic solution to RESTful communication with short-range technologies through a gateway. The Dynamic Naming System presented in Section 3.4 enables the ad-hoc creation of RESTful endpoints to reflect the environment of the gateway. This is in keeping with the constraints of the system as the aim for the gateway is give remote access to devices within its environment over one or more short-range technologies. The endpoint maintenance discussed in section 3.4.3 also enhances the blanket nature of the solution by giving it the ability to reflect the changing environment of the gateway. The improvements posed in Section 6.2 show that there is the potential for expansion on the discussed system.

On a personal note, I was pleased with how the development went. The project gave me a great insight into the rapidly emerging area of the IoT. I gained valuable experience in developing my own solutions to real-world problems. Overall, my experience of the project was extremely positive, and I would like to continue my work on it.

Appendices

Appendix 1

The potential values for the opcode field in the WSP data frame are as follows:

- 0x0: indicates a continuation frame
- 0x1: indicates a text frame
- 0x2: indicates a binary frame
- 0x3 to 0x7 are reserved values
- 0x8: indicates a connection close
- 0x9: indicates a ping
- 0xA: indicates a pong
- 0xB to 0xF are reserved values

Appendix 2

The potential status codes to be used in a WSP closing handshake are as follows:

- 1000: a normal closure.
- 1001: used when a server is going down, or a browser has navigated away from a page.
- 1002: indicates a closure due to a protocol error.
- 1003: indicates a closure due to an endpoint receiving a data type it cannot accept.
- 1007: indicates a closure due to the data in a received frame being inconsistent with the message type.
- 1008: indicates a closure due to an endpoint receiving a message that terminates its policy. This status code is also used as a generic code.
- 1009: indicates a closure due to an endpoint receiving a message that is too large to process.
- 1010: indicates that an endpoint is terminating the connection due to a lack of desired extensions in the handshake.
- 1011: indicates that a server is terminating the connection due to an error which prevented it from fulfilling the request.

Appendix 3

A graphical representation of URI scheme alongside corresponding BLE components.
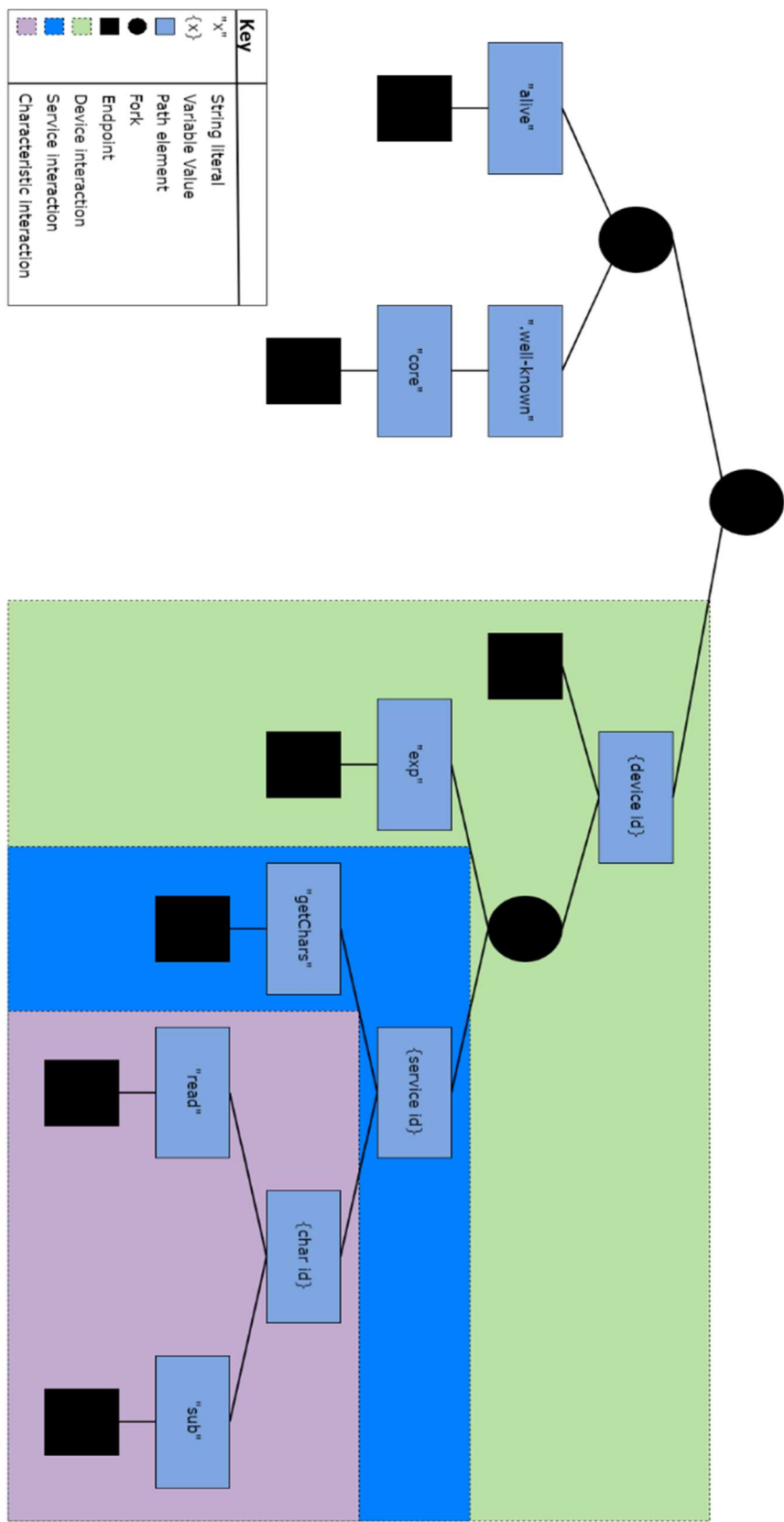
Bibliography

[1] Gartner Inc., "Gartner Says 8.4 Billion Connected "Things" Will Be in Use in 2017, Up 31 Percent From 2016," Gartner Inc., 7 February 2017. [Online]. Available: https://www.gartner.com/newsroom/id/3598917. [Accessed 10 February 2018].

[2] Bluetooth SIG, "Phone, Tablet & PC | Bluetooth Technology Website," Bluetooth SIG, Inc., 10 January 2018. [Online]. Available: https://www.bluetooth.com/markets/phone-pc. [Accessed 22 February 2018].

[3] C. Davis, M. Bender, T. Smith and J. Broad, "Feasibility and Acute Care Utilization Outcomes of a Post-Acute Transitional Telemonitoring Program for Underserved Chronic Disease Patients," September 2015. [Online]. Available: http://www.cchpca.org/sites/default/files/uploader/Feasibility%20and%20Acute%20Care%20Utilization%20Outcomes%20of%20a%20Post-Acute%20Transitional%20Telemonitoring%20Program%20for%20Underserved%20Chronic%20Disease%20Patients.pdf. [Accessed 1 April 2018].

[4] Deloitte, "Deloitte - Connected Health," 22 April 2015. [Online]. Available: https://www2.deloitte.com/content/dam/Deloitte/uk/Documents/life-sciences-health-care/deloitte-uk-connected-health.pdf. [Accessed 1 April 2018].

[5] S. Farahani, "ZigBee Basics," in *ZigBee Wireless Networks and Transceivers*, ELSEVIER, 2008, p. 1.

[6] J. Paradells and C. Gomez, "Wireless home automation networks: A survey of architectures and technologies," *IEEE Communications Magazine,* vol. 48, no. 6, pp. 92-101, 2010.

[7] L. Frenzel, "What's the difference between ZigBee and Z-Wave?," Electronic Design, 29 March 2012. [Online]. Available: http://www.electronicdesign.com/communications/what-s-difference-between-zigbee-and-z-wave. [Accessed 13 February 2018].

[8] T. Aasebø, "Wireless Technologies," [Online]. Available: http://cwi.unik.no/images/8/84/Wireless_technologies.pdf. [Accessed 20 February 2018].

[9] Dynastream Innovations Inc., "Tech FAQ - THIS IS ANT," Dynastream Innovations Inc., 26 July 2014. [Online]. Available: https://www.thisisant.com/developer/resources/tech-faq/category/10/. [Accessed 20 February 2018].

[10] Z. Shelby and C. Bormann, "Link Layers for 6LoWPAN," in *6LoWPAN The Wireless Embedded Internet*, Wiley, 2009, pp. 19-20.

[11] N. Kushalnagar, G. Montenegro and C. Schumacher, "IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs): Overview, Assumptions, Problem Statement, and Goals," August 2007. [Online]. Available: https://www.rfc-editor.org/rfc/pdfrfc/rfc4919.txt.pdf. [Accessed 20 February 2018].

[12] ZMDI, "Range - 6LoWPAN," Tecams, 3 August 2014. [Online]. Available: http://www.6lowpan.at/range.html. [Accessed 20 February 2018].

[13] C. Gomez, J. Oller and J. Paradells, "Overview and Evaluation of Bluetooth Low Energy: An Emerging Low-Power Wireless Technology," *Sensors,* vol. 12, no. 9, pp. 11734-11753, 2012.

[14] K. Townsend, C. Cufi, Akiba and R. Davidson, "Introduction," in *Getting Started With Bluetooth Low Energy*, O'Reilly, 2014, pp. 7-8.

[15] A. Dementyev, S. Hodges, S. Taylor and J. Smith, "Power Consumption Analysis of Bluetooth Low Energy, ZigBee and ANT Sensor Nodes in a Cyclic Sleep Scenario," in *2013 IEEE International Wireless Symposium*, Beijing, 2013.

[16] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari and M. Ayyash, "IEEE Communications Surveys & Tutorials," *Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications,* vol. 17, no. 4, pp. 2347-2376, 2015.

[17] C. Bormann, A. P. Castellani and Z. Shelby, "CoAP: An Application Protocol for Billions of Tiny Internet Nodes," *IEEE Internet Computing,* vol. 16, no. 2, pp. 62-67, 2012.

[18] The Chromium Projects, "SPDY: An experimental protocol for a faster web," 15 November 2009. [Online]. Available: http://dev.chromium.org/spdy/spdy-whitepaper. [Accessed 2 March 2018].

[19] Z. Shelby, Sensinode, K. Hartke, C. Bormann and B. Frank, "draft-ietf-core-coap-09 - Constrained Application Protocol (CoAP)," CoRE Working Group, 12 March 2012. [Online]. Available: https://tools.ietf.org/html/draft-ietf-core-coap-09. [Accessed 2 March 2018].

[20] K. Hartke, "RFC 7641 - Observing Resources in the Constrained Application Protocol (CoAP)," September 2015. [Online]. Available: https://datatracker.ietf.org/doc/rfc7641/. [Accessed 2 March 2018].

[21] Bluetooth SIG, "Radio Versions | Bluetooth Technology Website," Bluetooth SIG, 26 January 2018. [Online]. Available: https://www.bluetooth.com/bluetooth-technology/radio-versions. [Accessed 5 March 2018].

[22] Argenox Technologies, "A BLE Advertising Primer · Argenox Technologies," Argenox Technologies, 15 May 2015. [Online]. Available: http://www.argenox.com/bluetooth-low-energy-ble-v4-0-development/library/a-ble-advertising-primer/. [Accessed 8 March 2018].

[23] Bluetooth SIG, "Specifications of th Bluetooth System, v5.0," 6 December 2016. [Online]. Available: https://www.bluetooth.com/specifications/bluetooth-core-specification. [Accessed 10 March 2018].

[24] Bluetooth SIG, "GATT Overview | Bluetooth Technology Website," Bluetooth SIG, 16 December 2015. [Online]. Available: https://www.bluetooth.com/specifications/gatt/generic-attributes-overview. [Accessed 15 March 2018].

[25] MDPI, "energies-10-00393," 9 October 2017. [Online]. Available: http://www.mdpi.com/energies/energies-10-00393/article_deploy/html/images/energies-10-00393-g003.png. [Accessed 20 March 2018].

[26] Z. Shelby, K. Hartke and C. Bormann, "RFC7252 - The Constrained Application Protocol (CoAP)," June 2014. [Online]. Available: https://tools.ietf.org/html/rfc7252. [Accessed 13 March 2018].

[27] I. Fette and A. Melnikov, "RFC 6455 - The WebSocket Protocol," December 2011. [Online]. Available: https://tools.ietf.org/html/rfc6455. [Accessed 20 April 2018].

[28] M. Nottingham and E. Hammer-Lahav, "RFC5785 - Defining Well-Known Uniform Resource Identifiers (URIs)," April 2010. [Online]. Available: https://tools.ietf.org/html/rfc5785. [Accessed 13 March 2018].

[29] J. Nieminen, C. Gomez, M. Isomaki, T. Savolainen, B. Patil, Z. Shelby, M. Xi and J. Oller, "Networking Solutions for Connecting Bluetooth Low Energy Enabled Machines to the Internet of Things," *IEEE Network,* pp. 83-90, 2014.