Intelligent Sewer Blockage Detection System using Internet of Things

Benjamin Mark Buurman

Bachelor of Information Technology & Systems

A thesis submitted in total fulfilment of the requirements for the degree of

Master of Computing (Research)

School of Science, Engineering & Information Technology

Federation University Australia

Northways Road, Churchill, Victoria 3842 Australia

April 2019

Statement of Authorship

I hereby declare that I am the sole author of this master thesis, and I have not used any sources other than those listed in the bibliography and identified as references. I further declare that I have not submitted this thesis at any other institution in order to obtain a degree.

Bairnsdale, Victoria, Australia
20/04/2019

Declarations

I declare that:

- I have read the copyright legislation of the university and understand the provisions therein.
- My thesis does not contain material infringing the copyright of other persons.
- I have stated clearly and fully in my thesis the extent of any collaboration with others. To the best of my knowledge and belief, the thesis contains no material previously published by any other person except where due acknowledgement has been made.
- My thesis submitted contains no material which has been accepted for an award of any other degree or diploma at any university.
- My thesis has identified work of others relied upon by providing appropriate acknowledgement, citation, and reference in the text and in the bibliography.

I grant the University the right to display or copy any or all of the thesis, in all forms of media, for use within the University, and to make available the thesis to other persons or organisations being either educational or industrial, for reference purposes or for other legitimate educational purposes.

Bairnsdale, Victoria, Australia
20/04/2019

Dedicated to Stephanie Kate Buurman

Acknowledgements

Ben Buurman was supported by an Australian Government Research Training Program (RTP) Fee-Offset Scholarship through Federation University Australia.

I would like to begin with acknowledging my supervisors, Professor Joarder Kamruzzaman and Dr Gour Karmakar. Without their continued support, guidance, and encouragement, completing this thesis would not have been possible. Both are very intelligent, generous, and insightful people who I consider myself fortunate to have worked with over the years. Not only have I enjoyed the opportunity, but I have learned much from them – not just in academics, but in life.

The preliminary concept behind this project was developed in consultation with staff at East Gippsland Water Corporation (EGW) – namely, Mr. Steven Mowat (Innovation Officer and Field Technician) and Iain McDougall (Information Technology Manager). Both Mr. Mowat and Mr. McDougall have proven themselves as technical visionaries during their career, and Mr. McDougall supported the project in-kind until leaving EGW to pursue another opportunity. While the project is no longer being undertaken at EGW, I owe much to these two persons both as colleagues and friends. I would also like to acknowledge Mr. Terry Upton and Mr. Mathew Scott at EGW for their support during my work on this project.

On a related note, I would like to thank my current manager Luke Potter for his wonderful understanding, flexibility, and encouragement while I completed this thesis.

This would also never have been possible without my family – my father Andrew, mother Sharon, youngest sister Emily, and middle sister Stephanie who watches over us from heaven. For the entire time this project was underway, my family never stopped supporting me financially and morally. Without my family being as supportive, caring, and compassionate as they have been my whole life, I would never be where I am today.

I would also like to thank my girlfriend, Emily McLennan, who has been absolutely wonderful with her support and love. Every day she encourages me to be the best person I can and gives me a happiness and energy that motivate me to do well. Finally, I'd like to thank the following people. In your own ways, you have all helped me along this journey – Jack Lindsay, Jaymie Dawes, Peter Giacobbe, Guy Bransgrove, Alex Kyriazis, Kate McSweeney, Karina Burley, Andrew Nyhuis, Anna Dyer, Vikki White, and Tristan McQuitty.

Abstract

Despite being a common issue in both developed and developing countries, wastewater blockages have severe potential consequences. Blockages can be located at sewer mains or individual properties and can also be classified as *partial* or *full*. Full blockages completely obstruct a wastewater asset, and partial blockages will often develop into full blockages if left unattended.

Currently, blockages are managed by routine manual inspections to wastewater assets on a round-robin schedule. This is highly inefficient and costly, as blockages that form between these inspections and progress to effluent breaches will go undetected. In this thesis we present an Internet of Things (IoT) solution capable of simultaneously monitoring an entire wastewater infrastructure for blockages while still remaining inexpensive, reliable, and practical. Wireless motes use float switch sensors to detect blockages and transmit this to a central system using either LoRa or Wi-Fi communications. Making both LoRa and Wi-Fi available ensures the system can be adapted in any situation across a variety of geographic and economic restrictions.

The central system determines whether a surcharge is caused by a blockage or simply the result of regular activity not requiring intervention. Detection of false positives is critical, as deployment of field technicians is an expensive process that moves resources from other skilled work. If a surcharge is determined to be caused by a blockage, the central system will classify it as full or partial before estimating the property or length of main between properties it is located at. Following this, relevant parties will be notified so field technicians can be deployed to resolve the blockage.

We performed both practical laboratory testing and simulation modelling on our proposed system, and confirmed it is indeed capable of detecting, classifying, and locating blockages across a wide urban area. Our choice of hardware, software and network equipment ensures that the proposed IoT-based solution is inexpensive, workable, and easily deployable.

List of Abbreviations

Please note that this list does not include abbreviations that are common general knowledge (such as CPU), or cases where the abbreviation is the best-known identifier. This is common in network protocols, for example HTTP or TCP. People often are aware of the acronym more so than the full term.

IoT	Internet of Things
LTE	Long-Term Evolution (Cellular Standard)
GSM	Global System for Mobile Communications
LPWAN	Low Powered Wide-Area Network
WSN	Wireless Sensor Network
ICT	Information Communication Technologies
CCTV	Closed-Circuit Television
NS	LoRa Network Server
AS	LoRa Application Server
ADR	Adaptive Data Rate (Networking Service)
TX	Transmission (Serial Communications)
RX	Receipt (Serial Communications)
CRC	Cyclic Redundancy Check
MAC	Media Access Control
SNR	Signal-to-Noise Ratio
LWT	Last Will and Testament (Networking Service)
MQTT	Message Queueing Telemetry Transport
MQTT-SN	MQTT for Sensor Networks
MQTT-SN GW	MQTT-SN protocol Gateway

RSSI	Received Signal Strength Indicator
DCC	Dublin City Council
AQI	Air Quality Index
GPS	Global Positioning System
KDD	Knowledge Discovery Process
EGW	East Gippsland Water Corporation
DC	Direct Current
LED	Light-Emitting Diode
CSS	Combined Sewer System
GPIO	General-Purpose Input and Output
IRQ	Interrupt Request
Vcc	Voltage at the Common Collector
GND	Ground (Electronics)
SCADA	Supervisory Control and Data Acquisition
PDU	Protocol Data Unit
LIFO	Last-In-First-Out
API	Application Programming Interface
REST or RESTful	Representational State Transfer
RPZ	Raspberry Pi Zero
VM	Virtual Machine

Table of Contents

List of Figures

List of Tables

List of Technical Content

Pseudocode

Publications from this Work

1. Ben Buurman, Joarder Kamruzzaman, and Gour Karmakar, "Low-Power Wide-Area Networks: Design Goals, Architecture, Suitability to Use Cases and Research Challenges," submitted to IEEE Access and received Revise & Resubmit decision. Revision in progress and will be submitted soon.

2. Ben Buurman, Joarder Kamruzzaman, and Gour Karmakar, "Intelligent Sewer Blockage Detection System using Internet of Things". (in preparation)

Chapter 1 Introduction

1.1 Background

Since the dawn of the 21st century, the *Internet of Things* (IoT) has evolved from a supply-chain management platform [26] to a paradigm whose impact is compared to the introduction of the public Internet [35] [41]. The Internet has traditionally been perceived as human-centric, where human users request and are served remotely located resources including HTML documents and multimedia. IoT challenges this by giving ubiquitous devices Internet connectivity, allowing them to communicate with other devices, web servers, resources, and services. Estimates state that several billion IoT devices will be connected to the Internet by 2020 [24], with some predicting a number as large as 50 billion [26] and research challenges preparing for trillions of devices in the future [28].These numbers outnumber the amount of living humans, meaning the IoT could redefine the Internet as machine-centric in addition to human-centric.

IoT has largely involved giving Internet connectivity to everyday devices including household appliances, wearables, industrial machinery, fixtures such as light as taps, and healthcare devices. Connectivity will be natively granted to new devices during manufacturing by integrating microcontrollers, sensors, actuators, and network transceivers. For older objects, connectivity can be emulated by attaching IoT-capable sensors and/or actuators which interact with the objects. Discussion in recent years has raised the concept of a *Smart Earth*, discussed at levels including the President of the United States [35]. *Smart Earth* involves giving Internet connectivity to roads, urban infrastructure, utilities, and vehicles, alongside natural features such as waterways, forests, and animals. Stankovic [28] proposed the IoT will eventually become a 'layer' of sensing and actuation overlaid on the world, with all humans and devices constantly interacting with one another.

In summary, IoT can be defined by adding Internet connectivity to ubiquitous, non-human objects and devices with sensors and actuators. Sensors will gather contextual information for these 'things', while actuators will carry out some real-world function.

Many IoT implementations require many inexpensive devices deployed sparsely over long distances. Current wireless networks such as Wi-Fi, Bluetooth, LTE and GSM fail to meet these requirements as they often sacrifice range, cost-efficiency, or scalability to meet other requirements such as data rate. Limitations of current wireless networks led to the development of *Low Powered Wide Area Networks* (LPWANs) for deploying IoT systems. LPWANs have relatively low throughput but offer long communications range, inexpensive hardware, high scalability, and low power consumption [30-32] [42-43]. These attributes have made LPWANs useful for implementing a wide variety of IoT systems, with *smart cities* often discussed as a potential application.

Smart Cities are an increasingly popular initiative worldwide, however there is no single definition of the term. Zanella et al. [39] provide a good definition - using ICT to better utilise public resources and improve public services, while reducing the costs of administration. This same study proposes an *Urban IoT* to realise smart city goals through collecting data on urban infrastructure and using this to optimise service delivery by remote actuation.

Urban IoT can also improve delivery of *utilities* such as power, water, wastewater, and gas. Water and wastewater are arguably the two most essential, as humans cannot live without potable water, and wastewater management significantly reduces health hazards. Examples of using *urban IoT* to effectively improve water treatment is shown in [34], while [44-45] show examples of enhancing wastewater treatment and disposal.

Traditional wireless technologies have insufficient range to deploy a network practically and cost-effectively. This is because water and wastewater assets are often deployed over long distances, deployed underground, or obscured by concrete. Water and wastewater infrastructure are often composed of a very large number of assets such as pipes, with many located in harsh or hard-to-reach places. An urban IoT to monitor this infrastructure must be capable of handling many devices, able to communicate over long distances, and prove tolerant to obstructions. As a consequence of requiring many devices, urban IoT for this infrastructure must be inexpensive to remain practical for limited budgets. Practicality also requires a long battery life, as changing batteries is expensive and often difficult in hard-to-reach places. Considering the above factors, LPWANs provide the perfect implementation for an urban IoT in wastewater or water systems.

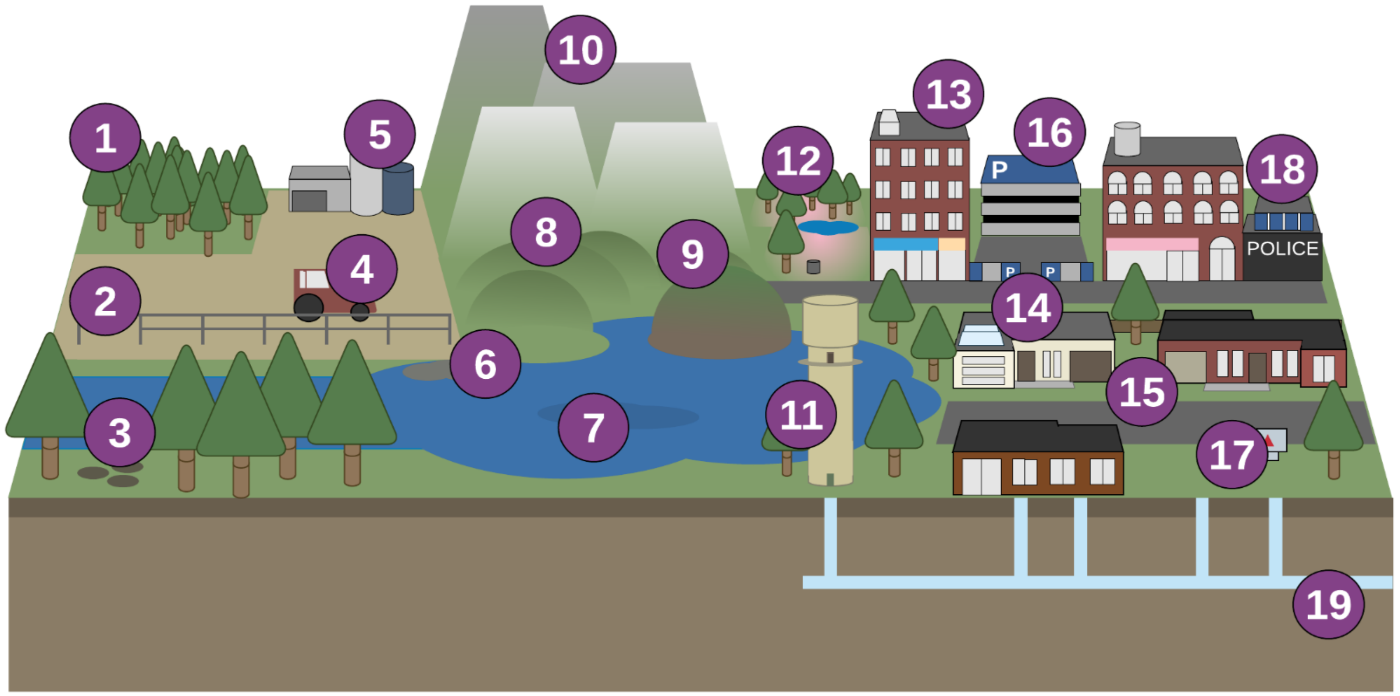


Figure 1-1 – An illustration of the Smart Earth concept, where data is collected from a wide variety of natural and human-made features to provide an 'overlay' of data collection. (1), (3), (7), (8), (9), and (10) are examples of this taking place in different natural environments and ecosystems including forests, plains, hills, mountains, animal burrows, and waterways. (12), (13), (14), (16) and (18) represent IoT enabled buildings or urban facilities capable of data collection and potentially actuation. This includes smart homes, commercial buildings, parks and recreational facilities, parking, and traffic management, and even emergency services. Similarly, (2), (4), and (5) show how data can be collected from farms and agricultural equipment. Finally, (11), (17), and (19) demonstrate an IoT-enabled smart water infrastructure. Water is not alone in this, as a smart earth could also include power, gas, garbage, or telecommunications infrastructure.

An urban IoT implemented with LPWAN technology could theoretically be used for monitoring wastewater systems and detecting blockages. Low-power, low-cost wireless sensors could be deployed throughout a wastewater system and used to read contextual variables which indicate blockage. The low cost and easy maintainability of LPWANs makes a potential network practical for deployment in developing countries, where improving wastewater systems could significantly reduce fatalities and environmental contamination alongside improving quality of life. In first-world countries, an inexpensive system with clear benefits provides significant motivation for governments to begin introducing smart city programs. Implementing such a system would also increase maintenance efficiency and reduce time of service delivery, providing additional benefits to both governments and customers.

1.2 Wastewater Systems

Effluent is a term used for liquid waste produced by human habitation. Homes and small businesses produce moderate amounts of effluent through fixtures such as sinks, toilets, showers, and baths, alongside appliances such as dishwashers and washing machines. Commercial facilities and urban infrastructure also produce a larger volume of effluent with a higher probability of containing hazardous materials. Effluent is mostly water; however, its other components range from human and animal waste to cleaning products and industrial chemicals. These present significant risks to both humans and the environment, making safe containment and disposal of effluent a high-priority issue for governments.

The aforementioned disposal and containment of effluent is implemented by urban infrastructure collectively termed wastewater systems. Sources of effluent at each property are connected to underground pipes named property connections. These carry effluent to underground pipes named sewer mains, with each sewer main connected to many property connections. Sewer mains end by feeding into other and often larger sewer mains, forming a network of pipes that eventually ends at a specified treatment plant or storage facility. Figure 1-2 illustrates the above system. To reach this intended destination, effluent must always flow through pipes in a specific direction. The direction which wastewater flows is named downstream, while the opposite direction named upstream.

An additional pipe named an *inspection shaft* is attached to each property connection between the actual property and sewer main. These rise vertically from the property connection and terminate at ground level, allowing observation of the inner property connection. Watertight caps are installed on these shafts to prevent foreign material entering the system or effluent leaking out. Field technicians use inspection shafts to search for *surcharges*, which is the industry term for an increase in effluent level. As we discuss below, surcharges can be used to determine both the type and location of a potential blockage.

Blockages can occur at both sewer mains and property connections when normal effluent flow is obstructed. This relative location is used to classify blockages, alongside whether they are *full* or *partial.* Full blockages completely occupy all available space in a pipe, and consequentially completely stop effluent flow from that location. In contrast, partial blockages do not occupy all available space, and provide one or more 'gaps' for effluent to flow through.

Both types of blockage result in surcharge at assets located upstream, with *full* blockages causing a constant rise and *partial blockages* causing a fluctuation. Property connection blockages will only cause surcharges in that individual property connection, while main blockages result in surcharges in all upstream connections. From a diagnostic perspective, a property connection blockage will result in that connection's inspection shaft surcharging. Conversely, a main blockage will cause observable surcharges in all upstream inspection shafts.

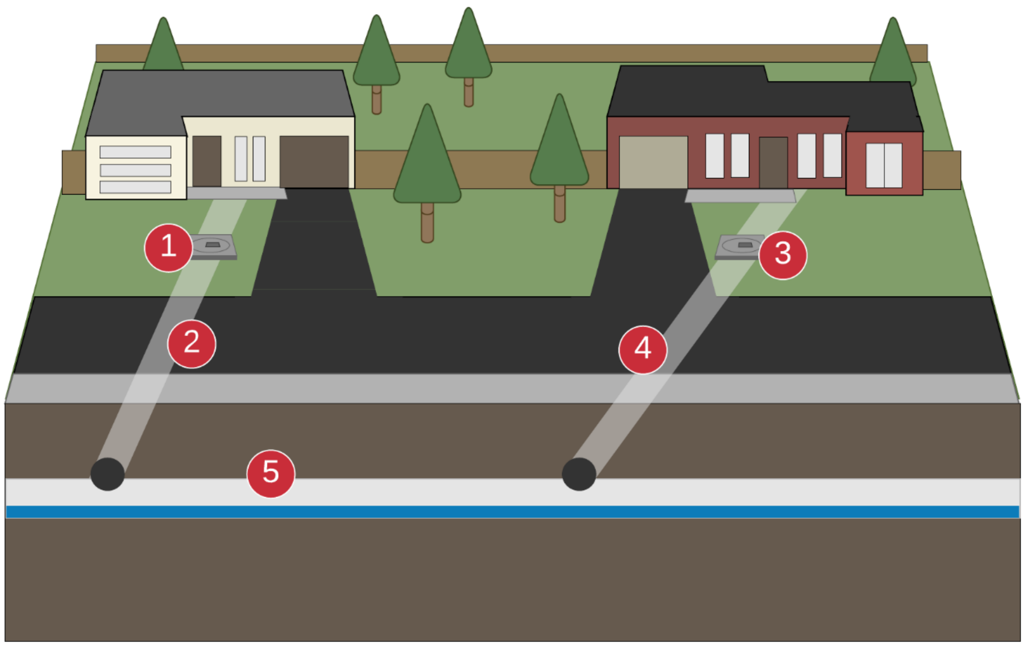


Figure 1-2 – Two properties connecting to their closest sewer main. 1 is the lid covering the first property's *inspection shaft*, and 2 is that property's *property connection*. 3 is the second property's inspection shaft lid, and 4 the second property connection. These serve the same purpose in both properties, and the connections carry effluent to the sewer main which is represented by 5.

1.3 Research Motivations

This section provides motivations for undertaking the research in this thesis by detailing the critical importance of detecting wastewater blockages and shortfalls of current systems. This research is largely motivated by the limitations of current systems, and endeavours to develop a new solution addressing these limitations using smart sensors and IoT. Following this, other motivations arising from the system's intended deployment are discussed.

1.3.1 Monitoring of Wastewater Blockages

Left unattended, blockages will inevitably cause surcharges, and eventually effluent will breach through inspection shafts or property fixtures. These breaches are not only distressing for residents but pose significant risks to public health and the environment. Pathogens in effluent can cause a myriad of disabling or fatal health conditions, which alarmingly do not even require direct contact with the effluent itself. Australia's Department of Health [8] stated that exposure can occur *indirectly* if animals or insects come into contact with the breach, and then with humans.

Utility providers are responsible for resolving blockages and decontaminating any areas exposed to effluent if a breach occurs. Not only is decontamination costly and time-consuming, but breaches can cause significant damage to a utility provider's reputation and open them to potential legal and regulatory action.

The current most common method of blockage monitoring is *Closed Circuit TV* (CCTV) inspection. This involves attaching a video camera to a long, retractable tube and pushing it through a length of sewer main. Video recorded inside the sewer is displayed on a monitor to field technicians in real-time, who inspect the footage for blockages or faults. Inspections are performed on every length of main on a round-robin basis, and ad-hoc on specific lengths if a blockage is suspected. These video recordings are often archived for further inspection and future reference.

Perhaps the biggest limitation of CCTV inspection is that it can only be performed on a round-robin basis. A combination of limited resources and large number of wastewater assets result in a potentially large time between inspections for a given asset, often measured in months but sometimes even years. Round-robin CCTV inspection is also only capable of detecting blockages that have formed and become noticeable in the time between each inspection. Logically, we can assume the probability of detecting a blockage has a negative relationship with the length of time between inspections. Many blockages consequentially go undetected until they cause an effluent breach, defeating the purpose of inspection.

Acoustic blockage detection is an increasingly popular alternative to CCTV inspection, utilising sonar technology to detect blockages or other abnormalities in a pipe. To utilise acoustic detection, technicians lower a special probe into a pipe and activate its sonar component. This is much quicker to perform than CCTV inspection, decreasing the length of time between inspections for a given asset. However, acoustic detection also shares its biggest limitation with CCTV technology; staff must physically visit each main on a round-robin basis. There will still be significant lengths of time between a given asset's subsequent inspections, and blockages will form during this time while remaining undetected.

Addressing the fundamental limitations with CCTV and acoustic detection requires a method for simultaneously monitoring an entire wastewater system without regular human intervention. Instead of blockages only being detected during sparsely spread routine inspections, blockages should be detectable as soon as they begin. Early detection will ensure blockages are detected before customers even notice any signs, along with preventing exposure to dangerous effluent. Field technicians will also spend much less time performing manual sewer inspections and can instead be assigned to tasks requiring more human ingenuity.

Building on Section 1.1's concepts of *Smart Cities* and *Urban IoT,* this research will utilise a successful union of hardware and software to develop a system addressing the limitations of current blockage detection techniques. By using a low-powered microcontroller, sensor and wireless transceiver, a device as shown in Figure 1-3 can be developed to detect sewer blockages. This device's sensor must be capable of detecting surcharges at the point of deployment, as surcharge indicates a downstream blockage. If the sensor detects a surcharge, a notification can be processed by the microcontroller and sent to a central system using the wireless transceiver.

Software deployed on a central system will be responsible for receiving surcharge alerts from devices and determining their location, before notifying appropriate staff. Field technicians will be deployed to the blockage's location and able to resolve the issue before it prevents a health risk. The central system should also be capable of determining whether a blockage is full or partial, and whether it is located at a property connection or main.

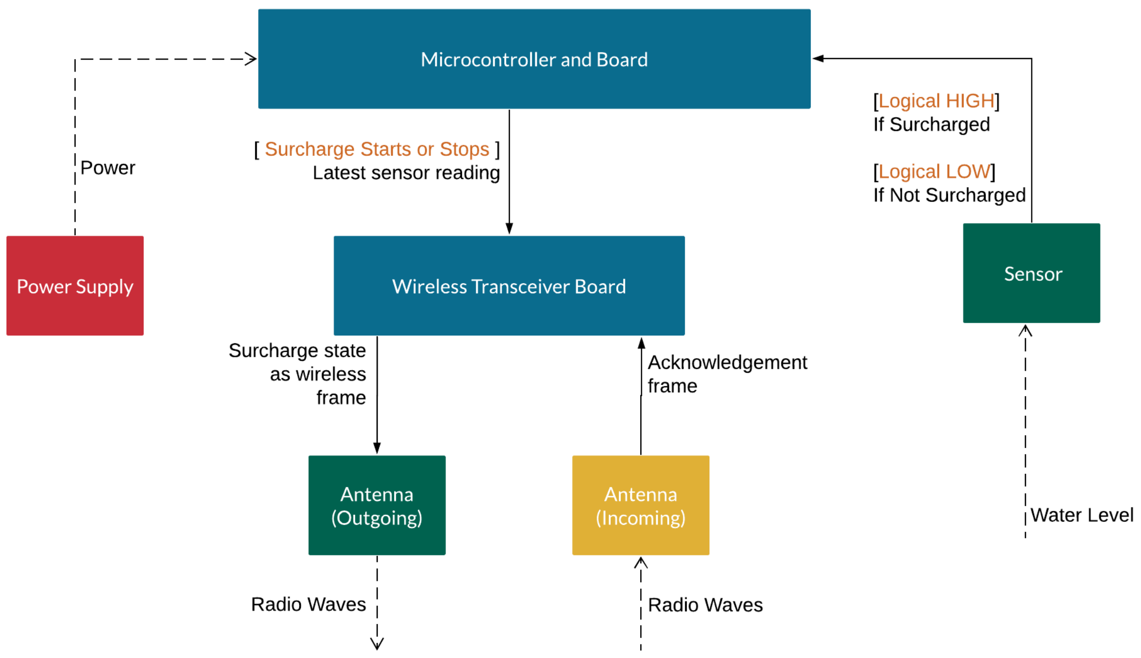


Figure 1-3 – An abstract, theoretical model of a hardware device which could detect sewer blockages. This would collect *water level*, as it is the most common and distinctly changing factor when blockages occur – essentially, this will be monitoring for surcharges.

This device contains a sensor to detect surcharges, a microcontroller for processing input, a transceiver for sending that input to a central system, and a power supply. The antenna is used for physically conveying the message.

1.3.2 Cost and Complexity of Current Solutions

Regardless of effectiveness, any wastewater monitoring system will be completely impractical if implementation and operation are too expensive. Utilities providers often utilise public funds and are fiscally conservative, resulting in hesitation to implement a new system where cost outweighs potential benefit or profit. Despite significant improvements to current methods such as CCTV or acoustic inspection, the system we will develop still performs the same overall function of blockage detection. Ultimately, management decisions at utilities providers will depend on the previously alluded comparison of cost and benefit. As a result, we must also conduct research on how to keep development, deployment, and maintenance costs as low as possible.

To provide maximum benefit our delivered system must be deployed throughout an entire wastewater infrastructure. Any areas where the system is not deployed will effectively be unmonitored and blockages will remain undetected. As the area serviced by wastewater infrastructure grows, two requirements will emerge; i - increased number of connected properties will require a larger number of devices, and ii - larger overall distance will require a greater wireless range. These two requirements do not necessarily grow in a linear fashion, with their individual rates of growth dependent on the area's population density. Both the per-unit cost of devices and cost of wireless network deployment must be kept low enough for utilities providers to realistically implement.

Looking elsewhere, sewer inspection methods in developing countries are often crude and dangerous, putting the persons responsible under significant risk. An example of this is 'manual scavenging', where people enter sewers without protective gear and remove blockages or perform repairs by hand. Our research can provide the greatest possible benefit for these countries, even potentially saving lives. This increases the motivation for developing very inexpensive techniques, as developing countries understandably have lower budgets for wastewater development. Any system we develop must not only be practical for wide-scale deployment in 'typical' environments, but also those with limited funds.

Complexity also presents another barrier to wide-scale adoption of the system. Regardless of cost-effectiveness, any system will provide little practical benefit if technicians are unable to operate it. Complex systems require more time and funds be spent on training, implementation, and troubleshooting – tipping the cost-benefit ratio further out of the system's favour. Furthermore, onerous, and overly complex systems will be seen by utility providers as providing less benefit. Utility providers are often change-adverse, and any new systems must prove beneficial relatively quickly and allow for easy transition.

1.3.3 False Blockage Alarms

As our research scope does not include developing an actuation method for resolving blockages, any blockages will be resolved by notifying the utility provider who then deploy field technicians to the correct asset. Deployment not only requires field technicians to immediately cease their current task but incurs costs to the employing utility provider in person-hours, transport, and asset lifetime. While the benefits of deployment far outweigh the costs if a blockage is occurring, this will not be the case if there is no actual blockage. Therefore, to ensure any system we develop is both financially viable and attractive to utilities providers, we must ensure that most blockage notifications are genuine.

False Alarms can be defined as any occasion where our system detects a surcharge, but no blockage is occurring. A system that simply detects surcharges with no additional logic will produce a myriad of false alarms, making it unviable for the reasons previously stated. There are two broad categories of events that can cause false alarms; actual surcharges caused by something other than a blockage, and surcharges being detected when none are occurring.

The first category of events is the most common, occurring on a near-daily basis when an appliance or fixture ejects a large amount of effluent in a very short period of time. This commonly occurs with dishwashers, washing machines, fixtures built before water-saving technology and industrial appliances. Utilities providers have no prerogative to respond to these ejections, as they have no negative consequences and are a normal part of wastewater infrastructure. Surcharges caused by these ejections are typically short-lived and prone to fluctuations much more rapid than partial blockages. Our research must determine a method for measuring and correctly processing surcharge durations to construct a model differentiating these events from blockages.

The second category of events occur when the float switch or mote detect a surcharge, but none are present. This is in contrast to the first category where a surcharge is present, but not caused by blockages. Faulty devices and sensors are the most obvious cause of this event, however quirks inherent in all sensors or electronics can also be responsible – examples of this include pin floating and switch bouncing. To prevent these from creating false alarms, our research must identify each of these quirks and develop methods for managing them. Additional effort must be placed into selecting quality equipment within the very limited price range and developing robust hardware and software that is naturally more resistant to faults.

1.4 Research Objectives

To address the motivations outlined in Section 1.4, the following objectives should be carried out;

1. Design, develop and test an inexpensive, energy-efficient device capable of detecting effluent surcharges and notifying a central system.
2. Design an IoT-based central system capable of processing readings from a large number of connected IoT-based devices, processing readings and notifying relevant parties. This system must also be capable of differentiating actual blockages from false alarms.

3. Evaluate the overall system and its functionality using a combination of a real-world simulation environment and virtual model.

1.5 Research Contribution

This research delivers an IoT-based system consisting of both hardware and software to fulfil the objectives outlined in Section 1.4. The system delivered will be capable of monitoring an entire wastewater system for blockages, determining blockage type and location, and notifying relevant staff of blockages. To fulfil objective 3, the system will be capable of differentiating blockages from false alarms such as *Rainfall-Dependent Infiltration and Inflow* (RDII).

Monitoring an entire wastewater system requires deploying a large number of devices across the system's potentially vast infrastructure. Manufacturing such a volume of devices is an unrealistic goal for this research, however we will provide sufficient information for other organisations to manufacture large-scale solutions. This research will also build a smaller-scale prototype of the system consisting of a single end device and central system for proof-of-concept and testing.

To deliver this system, we provide the following;

1. Research across various disciplines ranging from computer science to public health and urban infrastructure.
2. A detailed design for producing a low-cost, energy-efficient device capable of detecting effluent surcharges and communicating wirelessly with a central system.
3. A detailed design for an IoT-based central system capable of processing inputs from all devices in deliverable 2, differentiating genuine blockages from false alarms, and classifying and locating blockages.
4. A prototype system capable of demonstrating an implementation of key requirements and the aforementioned design. This will include both a mote and central server software, alongside a virtual communications network.
5. Testing and evaluation of the prototype system developed in deliverable 4.

Challenges for future research and starting points for future developments with the system will also be provided to allow continued evolution of the concept and suitability in a changing environment.

1.6 Thesis Structure

Chapter 2 starts by evaluating past research to perform a comprehensive multidisciplinary literature review. Applications of computer science to wastewater management and smart utilities, IoT systems, wireless communication technologies, and relevant hardware and software engineering are reviewed. In addition, other domains such as civil engineering and environmental science are reviewed for relevant wastewater management projects.

The system's design is provided in Chapter 3, first by developing a detailed overall model of the delivered system to separate it into three components; *motes, central system*, and the interface connecting them. Following this, the remainder of the chapter is divided into detailed designs for each component. Each component's detailed design utilises both our own research and that of past literature provided in Chapter 2.

Mote design first evaluates factors and potential issues that must be considered during the design process. Considering these issues, we follow this with actual designs for both the mote's hardware components and physical sensor used to detect surcharges. This is then expanded by selecting the wireless network technologies and protocols used by the mote and determining how these are integrated into the mote. Finally, we design the firmware that will run on these motes to utilise the hardware, sensor, and wireless platform.

Next, we design the overall communications network that links mote and central system. With goals of long range, low cost, and energy-efficiency, we utilise the research in Chapter 2 along with our own ingenuity to produce a robust and efficient network architecture.

Chapter 3 concludes with a design for the central system's software, which is responsible for processing all blockage notifications alongside notifying appropriate parties. In our research's context, *processing* refers to identifying false alarms, classifying, and locating a blockage. This design includes specifications for complex and parallel-processing algorithms used to classify and locate multiple surcharge messages simultaneously.

While Chapter 3 delivers the system's design, Chapter 4 demonstrates an implementation of this design through a prototype mote and central server. We first utilise schematics, photographs, and actual firmware code to detail how the mote was developed, before detailing architecture and specifications for the network and its hosts. Finally, we describe the central server's software in detail, presenting both its architecture and practical deployment on the aforementioned network.

Finally, Chapter 5 summarises our research, how this project met and surpassed the challenges, and other achievements made during the study. This is followed by analysing the experimental results and their implications, before suggesting what should be done in future for future improvement of the developed system.

Chapter 2 Literature Review

With the scope of our research planned, it is appropriate we start by reviewing previous relevant literature. We will begin by utilising past works to provide an overall definition for the *Internet of Things* (IoT) before discussing protocols and services with potential implications for our work. Following this, we will review the concept of *smart cities* by providing a robust definition of the concept and a series of examples where it has been used. This look at smart cities will be concluded by examining different architectures and protocols that have been identified, giving us a practical guideline for deployment the system developed in this research. Finally, we provide a more detailed analysis of wastewater infrastructure and implications of blockages, before finishing with an examination of other attempts at detecting them with an IoT system. Our review will show that while past research has made worthwhile contributions, nothing is yet capable of detecting sewer blockages across an urban area in a practical and cost-effective manner. Combined with the implication of blockages, this provides ample literature for our research to be completed.

This section is broken further into sub-sections, each associated with one of the points above.

2.1 Internet of Things (IoT)

Given that we intend to deliver our solution as an IoT based system, it seems appropriate to first discuss the IoT as a concept and provide a solid definition of what the term actually means. As a consequence of its somewhat rapid ascension into the mainstream corporate and commercial lexicon, there are a myriad of definitions and 'buzzwords' obscuring its true purpose.

All prior literature converges on defining the IoT as a concept connecting 'everyday' physical objects to the Internet and allowing them to communicate with each other and more 'standard' Internet services. For example, a smart home's front door could command lights to turn on when a person opens the door. Simultaneously, that person would be capable of remotely controlling and monitoring the lights through a more traditional web interface. Al-Sawari et al. [126] further elaborate by claiming the IoT will involve the Internet's transition from exclusively *human-to-human* communications to also include *human-to-thing* and *thing-to-*

thing communications. In this context, a *thing* is any non-human object that connects to the Internet.

The IoT was first introduced as a concept in 1999 by a researcher named Kevin Ashton, who intended to *bridge* real-world objects with the Internet [125] [127]. According to Miraz et al. [124], the IoT emerged as a distinct entity in 2008 when more inanimate objects were connected to the Internet than human users. Since this inception, the goal of building the IoT has always been unifying everything in the world with a standard infrastructure, allowing contextual data about the physical world to be collected from an unprecedented number of sources.

Nagakannan et al. [125] state that the introduction of IoT should make human lives simpler and more comfortable, which alone is sufficient motivation for investment and development. Routh and Pal [127] make a similar statement, claiming IoT is solving current everyday challenges and will continue to solve new challenges. Miraz et al. [124] paraphrase futurist Ray Hammond, who predicted that the linking of computer networks would increase the spread and dissemination of information on an unprecedented basis. While we have seen Hammond's prediction eventuate with the more familiar human-to-human Internet, applying these concepts to the IoT sees the unprecedented spread of information regarding the physical world we interact with every day. This provides significant motivation to commercial entities, with modern organisations often seeing data as their greatest asset – a fact easily proven by the increase in data scientist jobs.

Al-Sawari et al. [126] and Miraz et al. [124] both state *things* connecting to the IoT have several unique requirements, unique consequences of the system's massive scale and potentially scattered or remote deployment. We also propose these requirements stem from the actual nature of *things* – if Internet capabilities are added to everyday objects, any extra components must be as subtle and inconspicuous as possible. The unique requirements of IoT systems have been the subject of several papers, and it would be possible to fill this entire thesis with a study on them. However, for the sake of brevity, we will briefly state them as follows. IoT systems require very low power consumption, and therefore also require minimal processing power and storage volume alongside very energy-efficient wireless communications schemes. The potentially massive number of devices also requires very scalable systems with an equally massive number of potential addresses, or at least an intuitive method for handling a smaller address space [119]. Miraz et al. confirm this by affirming that IPv6's introduction was crucial to identifying "*billions of sensors*".

Al Sawari et al. [126] classify IoT communication protocols as either *Short Range* or *Low Powered Wide-Area Networks* (LPWANs). Naturally, this implies a taxonomy based on communications range, and while *Short-Range* networks have a reach measured in metres, LPWANs are typically measured in kilometres [31]. Comparing the specifications Raza et al. [43] provides for each LPWAN with those provided in [126] for short range protocols reveals a few interesting relationships. Short range networks generally provide a higher data rate than LPWANs and provide *mesh* or multi-hop topologies. In contrast, almost every LPWAN is restricted to the star topology, although Finnegan and Brown [67] note that LoRa can potentially support mesh networking.

Examples of short-range networks provided by [126] include the IP-based 6LoWPAN, ZigBee, Bluetooth, Radio Frequency Identification (RFID), Near Field Communication (NFC) and Z-Wave. They also provide SigFox and cellular networks as examples of LPWANs, however traditional LTE or GSM cellular networks do not fit this category due to high power consumption. More valid examples of LPWANs are provided in [43] and [67] including SigFox, LoRa, Weightless, NB-Fi, RPMA, and Telensa. While LTE and GSM are not strictly LPWANs, LPWAN adaptations of these standards with lower power consumption do exist such as the low-powered LTE-M, NB-IoT, and EC-GSM.

Mohanty et al. [121] architecturally break the IoT into four constituent components. This includes the *things* themselves, a local area network (LAN) that many things connect to, the public Internet, and the cloud. A *base station* device manages multiple things to form a LAN, and all traffic from these things must pass through it. One traffic reaches the Internet, it can utilise cloud computing as required.

Throughout the rest of this research, we will refer to Internet-enabled things as ***Motes***. This is a prominent term among the business world and general population, and for our purposes refers to a combination of physical *thing*, power supply, microcontroller, and wireless transceiver.

2.2 Low Powered Wireless Area Networks

Studies [42] [43] [67] comparing LPWAN protocols show that all commercially available networks utilise the *star* or *star-of-stars* network topology. *Star-of-stars* topology (illustrated in Figure 2-1) is very similar to standard star topology, with the defining exception that a mote can connect to multiple base stations. Motes will broadcast outgoing messages, and any base station in range will accept and process the reading. This potentially increases system complexity, as multiple base stations accepting a single message will naturally result in duplicates. Duplication can be handled by additional processing, however as always this results in further resource consumption.

Despite increasing the complexity of message processing, star-of-star topology allows networks to continue partial operation when base stations are offline, increasing both resilience and reliability. Increased resilience makes a star-of-star network much better suited to potentially sparse, signal-hostile deployment environments of LPWANs and IoT systems.

A wide variety of LPWAN platforms are available, with both standardisation and communication between multiple platforms in its relative infancy. Additionally, the availability of platforms differs across regions based on financial, political, and social factors. This raises several problems when attempting to develop a system that will be deployed worldwide, especially when considering that developing countries will reap the most benefit. As a consequence of reduced investment capacity and profitability for network operators, developing countries are likely to have a smaller range of LPWAN platforms available.

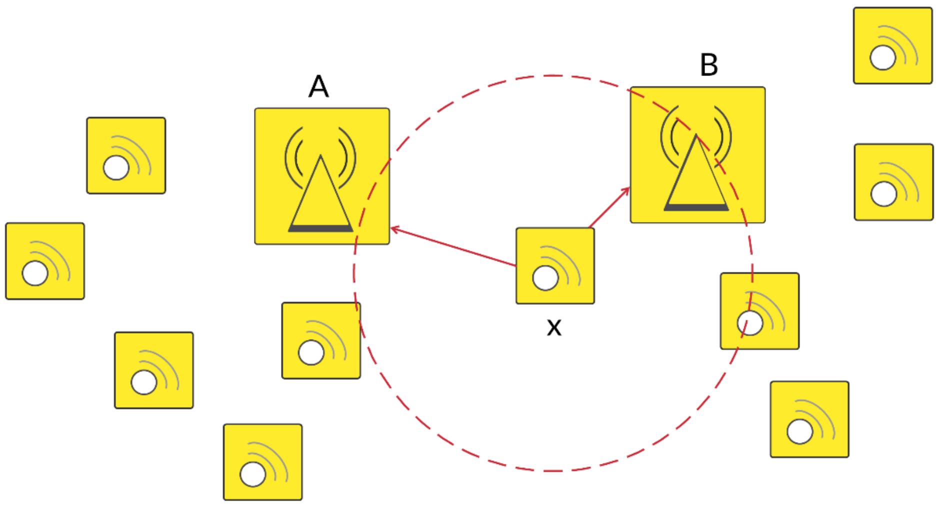


Figure 2-1: A simple depiction of the *Star-of-Stars* topology. The mote labelled *x* is in communications range of both base stations – A and B. As a result, it is capable of sending and receiving traffic from both. Algorithms must be implemented on either end to process and discard duplicates, with the highest signal strength packet often being retained.

After reviewing previous literature regarding different LPWAN platforms, we have determined that LoRa is the most suitable for any system we develop. LoRa provides a communication range of 5km in urban and 15km in rural environments [42] [67], which is sufficient for network deployment over a large area. In addition, LoRa base stations are theoretically capable of individually serving over 1,000,000 devices [67].

However, one of LoRa's biggest advantages is the ability to deploy private networks. While many organisations offer LoRa networks as a service, any organisation worldwide is capable of buying an inexpensive LoRa gateway and deploying their own network. Location no longer restricts the availability of LPWAN platforms, as by deploying private networks organisations can make LoRa available wherever they wish.

2.3 LoRa

The term 'LoRa' actually refers to two different protocols [68]. *LoRa* is the physical-layer modulation scheme used to transport data over an air interface, while *LoRaWAN* is the medium access control (MAC) protocol allowing multiple motes to send LoRa-modulated messages to a single gateway and vice-versa. As expected from the above definitions, LoRaWAN specifies the architecture of a LoRa network. LoRa itself is simply the means by which motes and gateways communicate in this architecture.

LoRaWAN specifies a network architecture by classifying all devices (named *LoRa Nodes* or *End Points*) based on how often they can receive incoming messages [72] as illustrated in Figure 2-2. This can be thought of as a form of *duty cycling*. Gateways are aware of each mote's scheduled receive windows, and store-and-forward messages bound for any connected motes according to this schedule [68].

According to technical specifications on LoRa, Gateways route mote data to a single available *Network Server* (NS) using a higher capacity backhaul. LoRaWAN's specification states that the NS receives and decodes LoRaWAN packets, before routing them to the correct *Application Servers*.

Application Servers (AS) are simply other servers that communicate with the NS over its available interfaces. Our system will utilise a single AS for all intelligent 'backend' processing and handling of surcharge data. This includes storage and retrieval, determining a surcharge's cause, locating a surcharge, handling alarms, and providing an API for other corporate systems.

Data sent from an external system to a mote must first be sent to the relevant AS, before being forwarded to the correct NS which will route it to that mote's gateway. In this situation, the NS is responsible for encoding the application data in the LoRa format and performing any other processing. For both uplink and downlink communications, the LoRaWAN specification also delegates responsibility for network security and administration, *Adaptive Data Rate* (ADR) mechanisms, and discarding of duplicate packets to the NS.

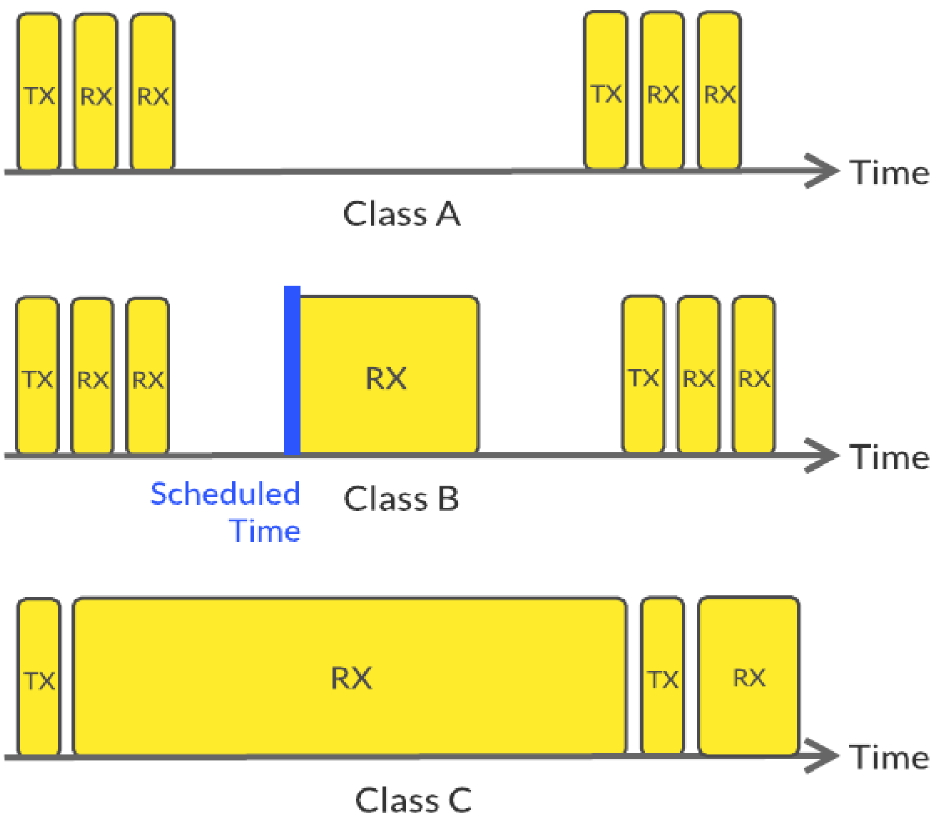


Figure 2-2: LoRaWAN device classes. The term *receive window* refers to a period of time during which a device is listening for incoming messages and is therefore capable of receiving them. **TX** denotes a period during which a device is transmitting (sending) a message, and **RX** denotes a receive window. It is important for system developers to understand that devices will always consume significantly more power while a receive window is open.

2.3.1 LoRa Frame Format

Augustin et al. [68] outline the frame format for LoRa's physical layer. LoRa PHY frames begin with a preamble between 10.25 and 65,539.25 symbols long, with this length configurable by network developers. Preamble length is determined by spreading rate, with higher spreading rates requiring longer preambles. Preambles mostly consist of constant upchirps, starting at the lowest frequency available for the selected bandwidth and ending at the highest. The last two of these upchirps modulate a variable named the *sync word*, performing a form of multiplexing by uniquely identifying the current LoRa network among those using the same frequency bands. Following these upchirps, 2.25 symbols of downchirp denote the end of the preamble.

Like with many network frame formats, LoRa preambles are followed by a 4-byte *header* [76]. Interestingly, unlike many other network protocols, inclusion of this header is entirely optional. Headers contain the payload's size in bytes, the utilised code rate, whether a message *Cyclic Redundancy Check* (CRC) is present at the end of the frame, and a header CRC. The payload's length in bytes is stored with a single byte, limiting each frame's payload to 256 bytes in length.

When encapsulating these PHY frames for MAC-level communications with LoRaWAN, the aforementioned PHY frames are encrypted with AES-128 and several leading fields are added. The first of these is *FPort*, which is a one-byte port multiplexing field. Following this is a group of fields named *FHDR*, which contains the following;

DevAddr – A 32-bit device address. 7 bits identify the network itself, while 25 bits uniquely identify the mote on the network.

ADR – A one-bit 'flag' denoting whether or not LoRa's ADR mechanism is being utilised.

ADRAckReq – A one-bit flag presumably used by the ADR mechanism for message acknowledgement.

ACK – A one-bit value acknowledging the last received frame.

FPending/RFU – A one-bit value called *FPending* in uplink messages and *RFU* in downlink messages. FPending flags whether or not a frame is pending from the network server, instructing the end device to send more frames to keep receive windows open. The purpose of RFU is not stated.

FOptsLen – A 4-bit number stating many additional MAC-layer commands (if any) have been added to this frame.

FCnt – A 16-bit frame counter. This potentially allows 65,536 frames to be transmitted while maintain sequence.

After *FCnt*, any MAC-layer commands will be added to the frame. All MAC commands are comprised of an 8-bit command ID, optionally followed by any bytes making up that command's parameters. Command parameters can be up to 32 bits in length. If no MAC commands are utilised, this space will still be filled with six bits.

The above sequence is then further encapsulated between other fields. Before this sequence are the following three fields;

MType – A 3-bit code representing the message type. This specifies whether the message is uplink or downlink, alongside whether an acknowledgement is required.

RFU – A 3-bit code which has not been elaborated on.

Major – A 2-bit code representing the version of the LoRaWAN protocol utilised by this frame. At the time of writing, the only permitted value is 0. This will only allow 4 versions as it is a 2-bit number.

The MAC sequence previously discussed follows the *Major* field and is then trailed by a 32-bit checksum field named *Message Integrity Code* (MIC). MIC is calculated using the *MType, RFU, Major, DevAddr, ADR, ADRAckReq, ACK, FPending/RFU, FOptsLen* and *FPort* fields alongside the encrypted PHY payload.

Considering protocol field values, Augustin et al. provide a formula for calculating the symbols required to send a payload, which is as follows. *PL* is the payload size in bytes, *SF* is the spreading factor, *CRC* value is 16 if enabled, *H* is 20 if the PHY frame header is enabled, *DE* is 2 if data rate optimisation is enabled and *CR* denotes code rate.

$$Symbols = 8 + \max\left(\left\lceil \frac{8PL - 4SF + 8 + CRC + H}{4 * (SF - DE)} \right\rceil \frac{4}{CR}, 0\right)$$

Communication range depends on *link budget*, the size of which varies according to bandwidth, coding scheme, transmission power, carrier frequency, and spreading factor [67]. LoRa can also theoretically utilise any bandwidth between 7.8 and 500 kHz, however Finnegan and Brown [67] note that only 125, 250 and 500 kHz are used in practise. Higher bandwidths result in a higher data rate and greater resistance to interference, but conversely lower communications range. The 125 kHz bandwidth appears to be the most commonly utilised in existing literature.

In both [69] and [75], Lavric and Popa provide examples of SNR limit, bitrate, and time-on-air measurements for each spreading factor at a 125 kHz bandwidth. These are detailed in Table 2-1 and observation shows clear trends. Higher spreading factors result in lower bit and symbol rates along with higher time-on-air and power consumption, however, also increase SNR limit – resulting in longer range and improved noise tolerance.

Research by Augustin et al. [68] measured the percentage of packets successfully received over a 2800m distance obstructed by urban environments. No packets were received with a spreading factor of 7, but a spreading factor of 12 saw 80% of packets received. As spreading factor was increased, the number of obstacles became increasingly irrelevant as obstacle penetration improved. However, this came at a cost – increasing spreading factor also lowered bit rate. Alongside reduced bit rate, Finnegan and Brown [67] demonstrated that increased spreading factors reduces the number of messages that can practically be sent per day (Table 2-2).

Table 2-1: Effects of spreading factor – 125 kHz bandwidth ([69], [75]).

Spreading factor	**Symbols/second**	**SNR limit**	**Time on air for 10-byte packet**	**Bitrate (baud)**
7	976	-7.5	56	5459
8	488	-10	103	3125
9	244	-12.5	205	1758
10	122	-15	371	977
11	61	-17.5	741	537
12	30	-20	1483	293

Table 2-2: Practical number of messages sent per day with each spreading factor. This assumes that the coding rate is 4/5, payload is 20 bytes and bandwidth is 125 kHz ([67]).

Spreading Factor	**Messages Per Day (Packets with 20-byte Payload)**
7	417
8	224
9	121
10	66
11	30
12	16

2.4 The MQTT and MQTT-SN Protocols

Several studies recommend using the *Message Queueing Telemetry Transport* (MQTT) application-layer protocol for IoT networks [77] [78], and upon observation their reasoning becomes obvious. MQTT is very simple and flexible, and its *publish-subscribe* architecture is perfectly suited to transparent communications between heterogenous systems.

MQTT requires a server named a *broker* to operate, which facilitates the aforementioned publish-subscribe communication between a number of clients [82]. Clients subscribe to a range of t*opics* by sending a *subscribe* request to the broker and can conversely *unsubscribe* as needed with another request. Clients also 'publish' data by assigning it a *topic* and sending it to the broker. When published data arrives at the broker, it will be sent it to all clients that have subscribed to its topic.

In addition, MQTT employs a mechanism named *Last Will and Testament* (LWT). LWT allows clients to specify a message and topic, and if they unexpectedly disconnect from the broker this message will be sent to all clients subscribed to that topic. We have given this mechanism special mention because of its potential advantages to an IoT system. As shown later in this chapter many IoT systems are of *critical* importance, with examples including infrastructure, medical, and security systems. Many IoT systems will also deploy motes in remote or hard-to-reach places, making on-site repairs impractical and costly.

If LWT is cleverly used, the self-healing network capabilities often restricted to short-range mesh networks can be extended to LPWANs. Even if this cannot be achieved, LWT can easily be used to notify network monitoring software or system administrators, expediating rapid response and quick repairs. Self-healing is particularly advantageous for the aforementioned types of IoT system. Critical systems can reduce downtime, while massive or remote systems can reduce the need for on-site deployment. Even if self-healing is not possible, rapid notification and following deployment will make significant reductions to downtime.

2.4.1 MQTT-SN

MQTT-SN is a version of MQTT developed especially for IoT systems utilising constrained networks and devices [83]. Fuqaha et al. [78] recommend using the connectionless UDP transport-layer protocol, as it exhibits far better performance with low-bandwidth and unreliable networks than the more ubiquitous TCP protocol [84]. UDP also provides its own adaptation of TCP's *Transport Layer Security* (TLS) mechanism named *Datagram Transport Layer Security* (DTLS). [81] As the name suggests, DTLS is designed for *datagrams* as used in connectionless systems such as UDP. Several distinctions between MQTT and MQTT-SN can be observed and are discussed below.

Most notably, MQTT-SN does not send a topic's human-readable name with *publish* and *subscribe* messages. Instead, a client sends the human-readable topic name to the broker once, which responds with a 2-byte ID it has mapped to that name. Other clients can then request this ID by sending the name to the broker. Interestingly, this does not always result in improved efficiency. If the topic name is under 2 bytes in length, mapping it to an ID will make no difference to message size and additional overhead will be required when requesting these IDs. In these cases, the process can be skipped, and the original topic name can be used instead.

MQTT-SN clients connect to an *MQTT-SN Gateway* (MQTT-SN GW) which translates its packets into the standard MQTT format before relaying them to the broker. If the MQTT-SN GW and client are not on the same network, an MQTT-SN *forwarder* can be attached to the client's network and held responsible for encapsulating packets and relaying them to the gateway over a backhaul interface. MQTT-SN GWs can also send MQTT-SN packets to the forwarder, which will decapsulate them and pass them to the client over that same backhaul.

Two types of configuration are possible for MQTT-SN GWs; *transparent* and *aggregating* gateways. Transparent gateways open and maintain an individual MQTT connection to the broker for each connecting client, while aggregating gateways open a single MQTT connection to the broker and multiplex all client communications. While transparent gateways perform direct translation between MQTT-SN and MQTT for each client, aggregating gateways intuitively determine which information is essential and only send that.

Transparent gateways are easier to configure, however aggregating gateways are much better suited to a large number of clients. Aggregating gateways prevent the broker from maintaining a potentially enormous number of connections and ensure any connection limits are not reached. Figure 2-3 provides a graphical overview of the MQTT-SN architecture.

2.4.2 MQTT-SN Packet Structure

All MQTT-SN packets can be broken into two sections [82];

- A *Message Header* between 2 and 4 bytes
- A *Message Variable Part* of variable length

The message header has a fixed format, while the message variable part's length depends on the message's type and any payload. This length is specified in the message header field. Message header can be further broken into two fields; *Length* and *MsgType*.

Length specifies the packet's total length and can actually be between 1 and 3 bytes. If the field's first byte is equal to 1 (00000001), then the remaining two bytes are the packet's total length. If the first byte has a different value, that value is the packet's length. Notably, packets under 256 bytes long are only permitted to use a single byte for storing length. Given that two bytes are available for message length, this gives a practical maximum length of 65,536 bytes or 65.536 kilobytes.

MsgType is a single byte and states the message's type, which outlines the message's intent and stage it fulfils in the MQTT-SN process – examples of this include CONNECT, PUBLISH and SUBSCRIBE. As can be seen from the presence of PUBLISH and SUBSCRIBE, these are vital to the core process of MQTT-SN. Table 2-4 provides more information on some of these messages.

A number of fields are available for the *Message Variable Part*, and each different type of message will contain a different combination of these fields. Data other than these fields is not permitted in the message variable part, so it will always be some combination of the values shown in Table 2-3. Some messages also contain a single byte named *flags* which consists of many 1 or 2-bit values. These provide valuable information about the message being sent and are detailed in Table 2-4.

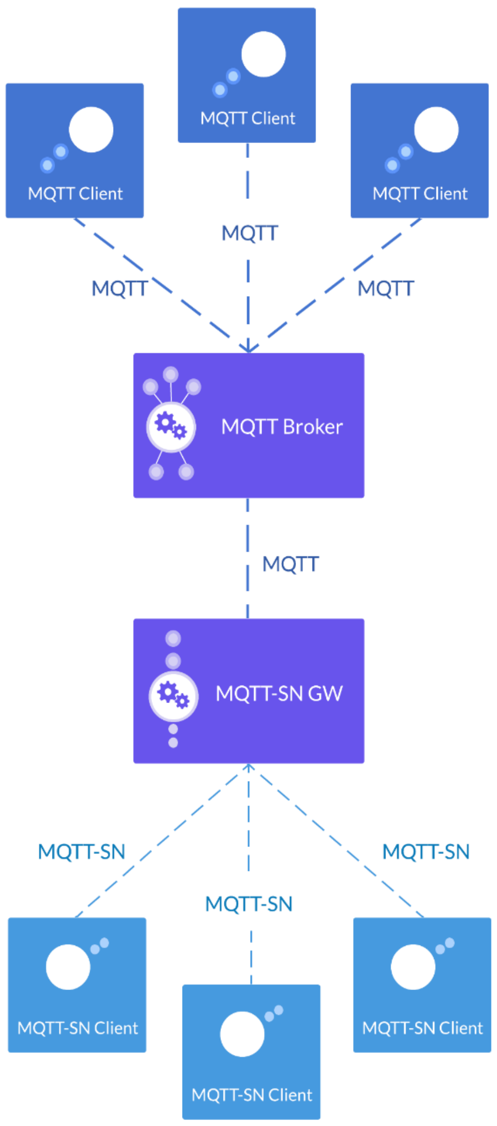


Figure 2-3: A graphical overview of MQTT-SN network architecture as described in this section. MQTT-SN clients send MQTT-SN messages to an MQTT-SN GW that translates them to standard MQTT. These are then forwarded to the *broker*, which facilitates publish/subscribe MQTT communications between clients. MQTT messages bound for MQTT-SN clients must also pass through the GW to be translated to MQTT-SN.

Table 2-3: *Message Variable Part* fields for MQTT-SN packets. Some fields have been omitted as their discussion is not essential to our research.

Field Name	Length (Bytes)	Purpose
ClientId	3	Uniquely identifies an MQTT-SN client
Data	Variable	Contains the message payload.
GwId	1	Uniquely identifies an MQTT-SN GW.
MsgId	2	Uniquely identifies a message. A message's ID will be shared with any acknowledgement, allowing the two to be matched.
TopicID	2	The 2-byte ID assigned to a specific topic.
TopicName	Variable	Contains the human-readable topic name.
WillMsg	Variable	Contains the will message to be sent upon unexpected disconnection.
WillTopic	Variable	Contains the topic name of the LWT message.
ReturnCode	1	If this message is being sent in response to another, this field contains the response status – or, the sender's reply to the original sender. The following values are permitted: • 00000000 – Accepted • 00000001 – Rejected due to congestion • 00000010 – Rejected due to invalid topic ID • 00000011 – Rejected due to message not being supported All other values are reserved.

Table 2-4: Bits constituting the MQTT-SN *flags* byte. Some have been omitted as discussing them is not essential to our research.

Name	Length (Bits)	Purpose
DUP	1	Whether this message is a retransmission of an earlier one. This is only relevant for PUBLISH messages.
Retain	1	When publishing a message, this flag states whether the message's value should be retained. When a message value is retained, the specified topic will stay at this value until otherwise specified. This means that any new subscribers to that topic will be able to view this current value. When not retained, the topic's value will clear once it has been sent to all subscribers. This means any new subscribers will not be able to see this value.
Will	1	Used when a client connects to a broker and indicates whether the client will utilise the LWT functionality.
CleanSession	1	Used when a client connects to a broker. If true, the broker will not store any information on the client such as subscribed topics or unpublished messages. This can be set to true if the client is only going to publish messages and has no intention of subscribing to any topics.
TopicIdType	2	If this message contains a topic ID, this states the type of ID used. 00000000 indicates a standard topic ID, while 00000001 indicates a pre-defined topic ID. As previously mentioned, topic names under 2 bytes can disregard the entire process of mapping to an ID. In this case, a value of 00000010 indicates a short topic name is stored in the topic ID field.

Table 2-5: Some of the types of MQTT-SN message.

Name	**Purpose**
CONNECT	Establishes a connection between client and gateway.
WILLTOPICREQ	If a client connects to an MQTT-SN GW and states it will utilise the LWT mechanism, the gateway will send this message to the client requesting the LWT topic.
WILLTOPIC	Sent by a client in response to a WILLTOPICREQ message
WILLMSGREQ	Like WILLTOPICREQ, this is sent from gateway to client if the client is using the LWT mechanism.
WILLMSG	Like WILLTOPIC, this is a client response to a gateway's WILLMSGREQ containing the LWT message.
PUBLISH	Publishes data under a given topic, with both clients and gateways being permitted to send it.
REGISTER	Uniquely, the REGISTER message has two purposes. When a gateway assigns a topic a unique ID, it will send a REGISTER message to a client informing them of this unique ID. If clients know a topic's name but not the unique ID assigned by the gateway, they can send a REGISTER message to the gateway to request it. If this is sent from a client, *Topic ID* will be zero.
SUBSCRIBE	Sent by a client to subscribe to a specified topic.
DISCONNECT	Sent by a client to disconnect from the gateway and close the active connection. These messages can optionally contain a *Duration* field, used by clients which intend to enter *sleep mode*. Gateways can also send DISCONNECT messages to client if they are experiencing errors processing a client's message. This will instruct the client to re-establish the connection to the gateway, ideally resolving any errors.

2.5 LoRa and MQTT

While the LoRaWAN specification is informative and suited to a wide variety of deployments, it fails to account for an MQTT-SN deployment and required components such as gateways and brokers. To integrate MQTT functionality with LoRa, our system must deviate slightly from the specification's standard architecture. By gateways sending packets directly to the MQTT-SN GW or MQTT broker, the vital *network server* component of LoRa processing appears to be skipped. Network servers are often described as core components of any LoRa network, where gateways communicate directly with network servers and exchange their respective UDP/IP packets.

Thankfully, previous systems have provided examples of how LoRa can integrate with an MQTT-based system. The *Things Network,* a large corporate provider of LoRa networks as a service, deploy their network servers as an MQTT client [85]. Network servers subscribe to MQTT topics related to LoRa traffic, alongside publishing LoRa-specific information which gateways subscribe to. Using this architecture, the MQTT broker acts as the intermediary between network server and gateway. Control messages to dictate network function are simply *published* to the MQTT broker by the network server and subscribing gateways can subscribe to the relevant topics to retrieve these controls. Network servers can also subscribe to topics containing key network management parameters, which will be published by individual gateways.

Penkov et al. also proposed a LoRa/MQTT system architecture for industrial networks that utilised a network server. In the paper presenting their architecture [89], each gateway connected directly to a network server which passed data to the relevant user application. However, this paper did not specify how MQTT was integrated into this architecture, and where the broker would fit in the data transfer process.

Other studies have seemingly ignored the requirement of network servers, using the MQTT broker as the sole intermediary between gateway and application server. Spinsante et al. [86] developed a network architecture for building automation systems using MQTT and LoRa, with no mention of a network server in the research paper presenting the architecture. LoRa gateways and the application server both communicated solely with the MQTT broker, which acted as an *intermediary* between the two. In all cases, the *Received Signal Strength Indicator* (RSSI) was far above receiver sensitivity, leading to adequate and reliable performance.

In [87], Wu et al. presented an IoT system based on LoRa and MQTT for managing elderly patients suffering from dementia. Wu et al.'s system involved placing LoRa transceivers in patient footwear, which presents similar challenges to our research involving ground-level communication. Again, this research paper did not mention a LoRa network server, and gateways sent messages directly to the cloud server using MQTT. Kim et al. [88] also proposed a generic MQTT architecture for IoT systems in [88], where LoRa gateways appear to correct directly to the broker.

2.6 The Node.js Framework

Node.js is a server-side JavaScript environment focused on rapid application development, extending the ubiquitous client-side language to server-side programming. A study by Chitra & Satapathy [104] showed that Node.js exhibits far better performance than a traditional web server (IIS) at tasks requiring a large number of I/O operations – or, large numbers of client communications. Given the nature of IoT systems involves a potentially massive number of devices connecting to a single server, this makes Node.js highly suitable for IoT development. Works in [105-108] present different examples of Node.js being used to build a central system for an IoT solution. As a result, we will provide a brief discussion of the framework and its core features.

2.6.1 Node Modules and Packages

Node.js applications and systems are organised into *modules* – JavaScript files exposing classes, functions, or variables to be referenced in other modules or files. Technically, all Node.js executables are built from modules, with even a single script forming its own monolithic module. If only a single script is used, the module will be completely self-contained and have no interaction with other files. Modules ideally have a single clearly defined purpose with little-to-no overlap between them. Exposed artefacts should also provide a higher-level 'entry point' to the module, hiding lower-level implementation from developers.

To expose their intended artefacts, every module creates an object named ***exports*** at compile-time. This object contains a field for each artefact to be exposed, and to add an artefact to this object developers assign it a given field name. An example is shown below for an anonymous function;

```
exports.getDate = function() { return new Date(); };
```

Conversely, to access artefacts exposed by a given module developers utilise a function named **require.** This accepts a module's file path as a parameter, before returning the **exports** object generated by the specified module. When a module is imported through **require**, it will be fully compiled and executed before returning **exports.** Node.js has a series of directories that it will automatically search for modules – if a module is placed in one of these directories, only its name needs to be passed. These directories are stored *relative* to the executing script's own directory – for example, Node will search the executing script's own directory, as well as any sub-directories with certain names.

Node.js modules are often grouped in single entities named *packages*. Packages are imported through the *require* function, and when imported expose all exported objects of constituent modules. As a result, packages are an efficient method for organising modules that fulfil a greater function. Packages also allow modules to be searchable on public repositories so developers can search for a package that fulfils their required purpose. Advantageously, packages enforce dependencies between modules and other packages, increasing the likelihood of successful compilation.

2.6.2 Parallel Programming with Node.js

IoT systems can benefit hugely from processing data arriving from several motes in parallel. This further confirms the suitability of Node.js – not only does it show increased performance when handling multiple communications [104], but several functions for parallel programming are built into the language itself.

2.6.2.1 *Timeouts*

The ***setTimeout*()** function asynchronously waits for a specified period of time before asynchronously executing a given function. Calling this function actually returns an object representing the wait that can be stored as a variable, or alternatively 'cancelled' with the ***clearTimeout*()** method.

Similarly, ***setInterval()*** schedules another function to execute at a specified interval expressed in milliseconds. While the function specified by **setTimeout** will only execute once, functions specified with **setInterval** will continually execute and restart the countdown until cancelled with the **clearInterval()** function.

Both timeout functions accept two parameters – a first-class function to be executed, and a time period for the countdown expressed in milliseconds.

2.6.2.2 *Promises*

Promises are another special object in Node.js, performing an asynchronous task independently of the main flow of execution. Two functions are called inside each promise – ***resolve()*** and ***reject()***. The value returned by the asynchronous task when successful should be passed into *resolve*, whereas any value returned or created by an unsuccessful task should be passed into *reject*.

Two higher-order functions can be invoked on a Promise – ***then()*** and ***error***(). These each accept an additional function as a parameter, with these additional functions having a single parameter of their own. The function passed into ***then*** will have the value previously passed into ***resolve*** as its parameter, while the same applies to ***error*** and ***reject***. These higher-order functions will also not execute until the Promise has finished processing, and either ***resolve*** or ***reject*** values are available. All lines after code after the Promise will execute as normal, and any **then** or **error** processing will occur in parallel when appropriate.

2.7 Smart Cities

In the first chapter of this thesis, we provided Zanella et al. [39]'s definition of smart cities – using ICT to better utilise and improve public services, while reducing costs to city authorities. A very similar definition is provided by Jin et al. [37], who state that a smart city is one which uses ICT to make city services and monitoring more aware, interactive, and efficient. Other papers arrive at similar definitions, including [119-121]. Each of these definitions require clear benefits to residents and city administrators, with residents experiencing better services and administrators experiencing reduced costs. Jin's definition of services and monitoring poses an interesting implication – not only is infrastructure being made 'smarter' and more efficient, but a potentially massive amount of data can be collected.

The nature of IoT makes it highly suitable for implementing smart cities, a statement echoed by researchers in [35-37]. Notably, Mohanty et al. [121] go as far as stating that the IoT is the *backbone* of the smart city concept. Adding *intelligence* to infrastructure and city assets can transform them into smart *things*, allowing them to connect to the Internet. Internet-capable devices can be placed on existing infrastructure to collect physical information through sensors and send commands through electronic interfaces. Newer infrastructure and assets can also have Internet capabilities built-in during manufacture, a concept beginning to appear in modern infrastructure. Revisiting the first chapter again, Zanella et al. proposed a platform named the *Urban IoT* to realise smart city goals through collecting data on urban infrastructure and using this to optimise service delivery by remote actuation.

An urban IoT can also improve delivery of *utilities* such as power, water, wastewater, and gas. We are particularly interested in the applications of urban IoT to wastewater management, as this aligns with the goals of this research. An example of using *urban IoT* to effectively improve water treatment is shown in [34], while [44-45] show examples of enhancing wastewater treatment and disposal. In the following section, we will discuss specific implementations of the smart city paradigm in greater detail.

2.7.1 Examples of Smart Cities

Zanella et al. [39] provided an example of an urban IoT's successful implementation, detailing the smart city project in Padova, Italy. Developed by the University of Padova, this served as both an experiment and demonstration of the potential held by smart cities and urban IoT networks. Motes were deployed throughout Padova and placed on streetlights, each connected to variety of sensors collecting environmental data. This data included carbon dioxide level, air temperature, humidity, noise, and vibration. Perhaps most importantly, these used a light sensor to determine if the streetlight was operational; if the light detected was below a certain threshold, it could be assumed the streetlight was not working. Motes were also placed in a transparent plastic case to protect from the elements; this is a good point to raise, as physical ruggedization is often forgotten in theoretical discussion.

Padova's motes use the 6LoWPAN *multi-hop* short-range protocol to form a mesh network, with routing handled by the *Ipv6 Routing Protocol for Low-Power and Lossy Networks* (RPL). As these are both IP-based protocols, each mote is uniquely identified with an IPv6 address compatible with the public Internet. This mesh network is bridged with the Internet using a *border router* that also acts as a gateway. While each mote is an Internet-connected device, messages exchanged in the 6LoWPAN protocol are incompatible with standard TCP/IP communications. The aforementioned gateway is the single point of contact between this network and the public Internet and is also responsible for translating messages between IP and 6LoWPAN as appropriate. A traditional fibre-optic network is used to *backhaul* this network to the public Internet, with all data destined for the *central system* (backend) required to pass through the gateway to reach this link.

Working with the previously discussed 6LoWPan protocol, Padova's smart city also utilises the *CoAP* protocol for application-layer communications. CoAP is an alternative to HTTP intended for use in constrained environments, encoding messages in a raw binary format instead of HTTP's verbose human-readable text. CoAP is also compatible with conventional HTTP GET, PUT, POST and DELETE messages, while its response codes also map directly to those used by HTTP. While traditional HTTP-based hosts are capable of sending CoAP messages, Zanella et al. recommend using a *cross proxy* to translate between the two formats. The Padova smart city utilises CoAP for application-layer communication between nodes, whereas its backend database uses conventional HTTP. Consequentially, communication between the two layers requires using the cross-proxy as an intermediary.

The network gateway in Padova's smart city acts as a database server, collecting data from all motes and making it available to the public Internet where practical. As this server utilises unconstrained HTTP, a *cross proxy* is utilised for requesting CoAP mote data. When required, the gateway will request information from the cross-proxy. The cross-proxy will retrieve data from the correct mote using the CoAP format, before returning it to the gateway in a HTTP format.

Guibene et al. [32] provided another innovative use of the smart city concept, applying it to monitoring the River Liffey in Dublin, Ireland. This river had previously overflown and flooded an underground car park, and Dublin City Council (DCC) agreed to collaborate with Intel Corporation to develop a solution. A wireless mote was developed and placed in the river inside a waterproofed floating buoy, connected to a variety of sensors collecting different data. Sensors monitoring depth, water temperature, and flow velocity were placed outside the buoy to make physical contact with the river. Conversely, sensors inside the waterproofed buoy measured air temperature, humidity, and barometric pressure. Like the streetlight motes presented in Padova's smart city, this consisted of a single mote collecting information from many sensors.

While the buoy was deployed in the River Liffey, DCC and Intel also placed ultrasonic level sensors reading rainfall gauges across Dublin's city centre. This is exciting, as different sensors spread throughout a city collecting varied information is a perfect demonstration of the smart city concept. Unlike Padova's streetlight motes and the buoy, these seem to have consisted of motes with a single dedicated sensor.

Dublin's smart city utilises very different communication protocols to Padova, resulting in different network topologies. In contrast to Padova's *multi-hop mesh* topology resulting from the 6LoWPAN protocol, Dublin utilises two long-range protocols; LoRa and LTE cellular communications. Two LoRa gateways were deployed in range of the buoy to test the effects of distance, urban obstructions, and line-of-sight. The closest gateway is referred to as *DCC*, located 3km away from the buoy and at 40m higher altitude. Conversely, the other gateway (referred to as *Three Rock*) is 13km from the buoy at an altitude 575m higher. Communications are obstructed for both gateways by thick walls and traditional Irish buildings with stone masonry. Guibene et al. [32] did not report any problems with communication for either of the LoRa gateways, implying that both were able to satisfactorily communicate with the mote. This proves that LoRa is a suitable protocol for building urban IoT and smart city systems. LTE communications are discussed much less; however, it is assumed they communicate with the closest cell tower.

While not discussed for the rain gauges, Guibene et al. also discuss the power supply and duty cycling techniques used by the buoy mote. Three solar panels are mounted on top of the buoy, and these charges two 12V lead-acid batteries. This is an example of the increasing trend of solar power in smart cities and IoT systems; not only is solar power environmentally friendly, but it reduces the frequency of battery replacement. While a central microcontroller CPU is constantly active, all connected sensors mostly operate in very-low-power *sleep mode.* Every 10 minutes, the CPU will wake required sensors, read their provided data, and transmit it using LoRa before re-entering low power mode. A similar process also occurs every 12 hours, however this sends data over the LTE network. This is wise, as LTE transmission consumes significantly more power than LoRa.

In response to severe levels of air pollution and the associated health risks, Duangsuwan et al. [30] developed an urban IoT system capable of measuring air quality and providing results to citizens. It is theorised that not only can city authorities identify the worst polluted areas and plan restorative action, but citizens can also avoid these areas when necessary. While benefits to city administration are less obvious than immediate advantages for residents, decreasing exposure to air pollution will potentially reduce burden on the healthcare system and provide political advantage.

Duangsuwan et al. used five separate sensors to measure air quality – like the previous two smart city platforms discussed, this involves multiple sensors on a single mote collecting different values. These sensors respectively collected data on ozone level, ambient noise, carbon dioxide level, particulate dust matter, and carbon monoxide level. The mote's microcontroller communicated with these sensors to read and process values using both an analogue-to-digital converter (ADC) and the I^2C industrial protocol. Two of these motes were deployed in separate locations throughout Bangkok, with power provided through a 5V/2A power supply.

To communicate with a central system, the system designed in [30] utilised the *NB-IoT* LPWAN protocol. NB-IoT is a cellular LPWAN standard based on LTE, allowing it to be easily implemented with existing LTE infrastructure. This alone makes it very attractive to telecommunications providers (telcos), giving it several advantages such as availability, support, and the fact telcos will be responsible for installation and maintenance. NB-IoT also has a higher data rate than LoRa, however this comes at the cost of higher power consumption and lower communication range [43]. In addition, NB-IoT is a *licensed spectrum* technology, so setting up networks incurs a hefty fee. This fee is usually paid for by telcos, however costs are passed on to customers. No LPWAN platform is necessarily better than the others, with the 'correct' choice often depending on individual circumstances and preferences. As Bangkok has a robust pre-existing LTE infrastructure and installing private equipment could be difficult, it can be seen that NB-IoT is an appropriate choice.

With the aforementioned NB-IoT network, data was collected from each mote and sent to a website on the public Internet where citizens could view results. Each individual sensor reading is combined to produce a single *Air Quality Index* (AQI) reading, however the paper is unclear on whether this occurs at the node or web server.

2.7.2 Smart City Services

While we have discussed individual smart city projects in the previous section, those are only a handful of potential smart city applications. Several of the papers [39] [119-120] we reviewed on the subject of smart cities provide a list of applications that often converges on the same items. Applications with examples provided in the previous section have not been listed here, as there are already detailed descriptions.

A – Structural Health of Buildings

By placing sensors in buildings or public infrastructure such as bridges, IoT systems are capable of determining structural integrity from vibration or deformation data. Related sensors can also monitor environmental conditions and seismograph activity to produce more detailed information, determining environmental impact on structural integrity and monitoring earthquakes.

Zanella et al. propose that routine structural health monitoring should send 1 packet per 10 minutes, with 30 seconds of delay acceptable. However, delays for alarms notifying authorities of imminent collapse should not exceed 10 seconds. As power consumption is not exceedingly high, batteries provide a suitable source.

B – Waste Management

A commonly seen example of *waste management* in smart cities is the deployment of smart garbage containers fitted with weight sensors to determine fullness. Using fullness data, municipal authorities can optimise both truck routes and recycling to minimise unnecessary cost and environmental impact. For example, empty garbage containers can be excluded from truck routes. This should require relatively little battery power, with *energy harvesting* stated as a viable power source.

C – Traffic Management

Currently, traffic management is often conducted with expensive and resource-intensive camera-based systems capturing high-resolution images. Significant improvements to cost and efficiency be achieved by implementing an IoT system that utilises noise and air monitoring sensors. By cross-referencing noise and air-quality data with GPS information, a model of traffic congestion can be produced for a given area. While the system proposed was for a *smart parking* system, Zhou and Li [129]'s method of detecting cars with geomagnetic sensors could also prove invaluable to traffic management.

D – Energy Consumption Monitoring

Integrating IoT nodes into a city's power grid allows citizens to monitor their own power consumption, and city authorities to monitor power use through the entire city. City authorities can utilise this data to optimise power consumption, identify energy-efficient infrastructure, predict future demand, and prioritise supply to different areas.

Zanella et al. [39] also proposed an actuation component where power supply can be controlled at different points on the grid. Benefits become easily apparent in emergency situations or during plant outages, where rolling blackouts or prioritisation of emergency services are an unfortunate necessity. Sensors or actuators built into an urban electricity grid will also not require any external power source.

E – Smart Parking

Unlike many other applications discussed, smart parking is already available in many cities and firmly planted in the public consciousness. Zanella et al. provided two purposes for smart parking; directing motorists to the best available parking spaces and verifying permits. A further example of this was mentioned earlier in [129], where motorists could check for free parking spaces in advance.

F – Actuation and Salubrity of Public Buildings

Salubrity is a rarely used term meaning invigorating and providing comfort, which can be utilised at public buildings by integrating sensor and actuator systems. Public buildings include schools, museums, libraries, council offices and recreation facilities. Sensors and actuators can provide a myriad of services to these buildings, ranging from climate control to chlorine levels in public pools.

G– Public Security

CCTV surveillance cameras are often placed throughout a city to deter and investigate criminal activity, and many recent models include integrated Internet connectivity. Currently this Internet connectivity is used for both viewing footage in real-time and downloading it to an online location. However, as technology advances, several new and innovative methods of using this footage for public security are being developed. Notably, video and audio footage could be streamed to a web service or API that analyses it in real-time for suspicious or criminal behaviour. If this behaviour is flagged, security measures such as shutters could then be activated at nearby properties.

2.7.3 Smart City Design and Architecture

A myriad of *design decisions* must be made across all stages from initial research to practical deployment when developing a smart city. These decisions are evaluated from variety of perspectives, with each a number of empty specifications to be given values. As a system progresses through development, the results of certain design decisions can cause changes to ones made earlier – for example, changing network protocols can require different overall architecture.

While presenting their research for an urban IoT system in the city of Melbourne, Jin et al. [37] outlined several broad perspectives they considered during that system's design; *Network-Centric*, *Cloud-Centric,* and *Data-Centric* perspectives. The *Network-Centric* perspective is concerned with fundamental components of the networks connecting motes, gateways, and central servers. This examines both the flow of data between each network node, and the characteristics of that node which define and regulate the flow of data. Four design decisions fall under the network-centric perspective; how data is collected with sensors, addressing scheme, network protocols, and QoS mechanisms.

The *Cloud-Centric* perspective is more self-explanatory, and only applies to IoT systems utilising a cloud-based central system. Cloud-based systems focus on the interface provided to each mote or subnet allowing access to the cloud system, alongside the actual processing and data storage performed in that cloud system.

While the network-centric perspective examines how data flows between each node, the *Data-Centric* perspective examines the data itself. Another way of phrasing this is that instead of *how* and *where* the data moves, the actual data moving will be evaluated. Ultimately, this perspective is focused on the *knowledge discovery process* (KDD) - *data* is analysed to extract valuable *information*, which is interpreted by humans to become *knowledge*. Three design decisions are considered under this perspective, each mapping to another stage of the KDD; *data collection* is how the raw data is collected, *data processing* is how that data is transformed into information, and *data interpretation* is how it is conveyed as knowledge.

Siegel et al. [27] defined three perspectives that apply across all of those previously discussed; *security, privacy,* and *resource efficiency*. These can be thought of as occupying a different axis to those above and should be considered when making each of those decisions. While Siegel et al. [27] considered *security and privacy* a single perspective, we feel it prudent to draw a distinction. Security focuses on sensor data and actuator commands, while privacy focuses on data related to users or organisations.

2.7.3.1 Network Architecture

Upon commencing this study, we were aware that IoT systems typically had a basic architecture involving a massive amount of ubiquitous and constrained devices (motes) connecting to a central system through an intermediate base station. However, there are many different methods of implementing this architecture, each of which forms its own distinct and more specific architecture. Several of these methods have been detailed in previous literature, and we will review some in this section. A range of taxonomies have been provided for classifying IoT architecture, each based on a different attribute. Jin et al. provided two taxonomies in [37]; whether the architecture is based on previous models or not, and what can be accessed by external systems to what extent.

Conversely, Siegel et al. [27] classify IoT architecture by network topology, while Zanella et al. [39] base their classification on whether *constrained* or *unconstrained* protocols are used. This is highly relevant to their presentation of Padova's smart city, where exclusively *constrained* protocols were used for local area networks and motes. Zanella et al. also emphasise the need for intercommunication and transcoding between these protocols. Table 2-6 compares taxonomies from each of these sources and provides a list of classifications available for each. Many of the architectures detailed are self-explanatory, however some will require further explanation.

In *direct connectivity* networks, motes directly query their peers using a point-to-point protocol such as Bluetooth or ZigBee. This is best suited to very small and non-critical IoT systems, as it does not scale well and has relatively poor security. *Hub Connectivity* is the most commonly encountered architecture in literature and is what most think of when discussing IoT networks. Motes connect to a central hub or gateway, which is capable of connecting to multiple motes, peer gateways, or a central system. All messages to and from devices pass through this hub, which carries out tasks including flow control, under-sampling, and security services. Hubs can further decrease bandwidth requirements by aggregating data from all connected nodes, and only sending the aggregate results. Siegel et al. [27] conclude that hub connectivity is best suited to small or medium-sized networks where payload size is known.

Cloud Connectivity is proposed as a solution for large-scale networks such as smart cities, likely because of the model's infinite scalability. This infinite scalability and the mechanisms allowing it come at a cost, however, and cloud connectivity is also stated to be needlessly expensive for smaller systems. From a technical perspective this is essentially an extension of hub connectivity, where each mote communicates with virtual hardware in a cloud environment that can be scaled up or down as required. The cloud system abstracts devices and is only concerned with each device's data flow and applications of that data, and consequentially developers must ensure that their IoT network is capable with the cloud interface [37] . Developers are permitted to build applications for interfacing with the cloud, while data-mining professionals can build tools for extracting valuable information from collected data.

Another architecture proposed in the literature is *autonomous networks*, which are completely isolated and not connected to public networks such as the Internet. However, some autonomous network gateways can still be accessed over public networks, and in these situations act as an intermediary for mote traffic. Despite not being connected to the public Internet, motes in many autonomous networks utilise the TCP/IP protocol stack with IPv6 because it is scalable, simple, and effective.

Motes or intermediate servers belonging to *ubiquitous networks* are part of the public Internet and can be directly accessed by clients. Intermediate servers are motes often possessing higher processing power and higher-capacity power supplies, implemented by some networks to serve several other motes as data sinks.

Table 2-6: IoT architecture taxonomies in literature.

Source	**Classified Based On:**	**Classifications**
Jin et al. [37]	Whether architecture is built on an existing architecture.	• Evolutionary • Clean Slate
Jin et al. [37]	What is accessible to the public Internet, and to what extent.	• Autonomous • Ubiquitous • Application-Layer Overlay
Siegel et al. [27]	Network Topology.	• Direct Connectivity • Hub Connectivity • Cloud Connectivity
Zanella et al. [39]	Whether *constrained* (resource-conservative) protocols are used.	• Constrained • Unconstrained

Utilising intermediate servers increases scalability and lowers resource demand on nodes, which is useful for serving very constrained motes or massive-scaled networks. Ubiquitous networks are often *hierarchical*, with sub-gateways served by the main gateway forming subnets. These subnets can have varying air interfaces and even network topologies provided their sub-gateways can communicate with the main gateway.

Finally, *application-layer overlay* architecture is a variant of ubiquitous network architecture where intermediate servers named cluster heads are given a special role. Cluster heads process data from connected nodes using techniques such as aggregation and feature extraction, then send results to the main gateway as needed. As processing is often carried out by higher-level software on cluster heads, data should be transported through application-layer protocols such as HTTP or CoAP. Utilising application-layer protocols provides easy access to high-level software through operating system sockets.

Results are only transmitted from cluster heads when scheduled or in response to events such as alarms. In addition, store-and-forward mechanisms can further conserve energy by decreasing the frequency of non-critical alarm transmission. Dividing an IoT network into cluster heads and subnets mitigates a common issue where large bottlenecks form around a single gateway. As the number of cluster heads increases, the bottlenecks present at each decrease in severity. However, introducing too many cluster heads is also disadvantageous. Excessive cluster heads create overly complex routing, increased hardware cost, under-utilisation of resources and further exhaustion of address space.

2.8 Wastewater Blockages

As discussed in the previous chapter, wastewater blockages occur when a solid obstruction inhibits the flow of effluent through a wastewater asset. We will discuss this phenomenon in further detail here; information was obtained through conversations with field technicians at East Gippsland Water corporation (EGW). EGW provide water and wastewater services to the *East Gippsland* region of Victoria, Australia. East Gippsland covers an area of 21,000 kilometres however only has a population of 45,000 people. This is a very low population density, and much of East Gippsland is covered by old growth forest and national parks. As of April 2019, EGW provide water services to 26,450 customers and wastewater services to 22,491 [130]. These services operate in isolated systems collecting from different rivers across East Gippsland's sparsely spread population centres.

Blockages usually form over time from the accumulation of foreign objects introduced to the asset. These objects can enter the asset through intended means when people use fixtures to inappropriately dispose of objects. Several incidents were observed at EGW where sanitary products, disposable wipes, nappies and even items of clothing were flushed down toilets alongside isolated incidents where children flushed toys. These objects collected until they reached a sufficient size and formed a blockage. Most blockages at EGW were large collections of the aforementioned objects, however at times large flushed objects such as toys singlehandedly caused blockages. Animal fats are also responsible for many blockages, often introduced to effluent through food manufacturing and domestic cooking. Fat will stick to infrastructure and collect as lipids attract each other, leading to blockages called *fatbergs*.

Conversely, foreign objects can also enter the wastewater system through unintended means such as breaks in a pipe or maintenance shafts left open. Perhaps the most notable example of this was when tree roots grow towards a pipe and eventually puncture it. Wastewater is highly nutritious to plant life, and roots that have entered a pipe will grow inside it relatively quickly. The rapid growth of roots inside a pipe will eventually grow thicker, collect other solid material, and obstruct it to create a blockage. This is such a common occurrence that it was the most common cause of blockages at EGW. EGW also identified rocks, landfill, and tree detritus as potential causes of blockage that enter the system through broken assets. Figure 2-4 shows the gradual build-up of blockages.

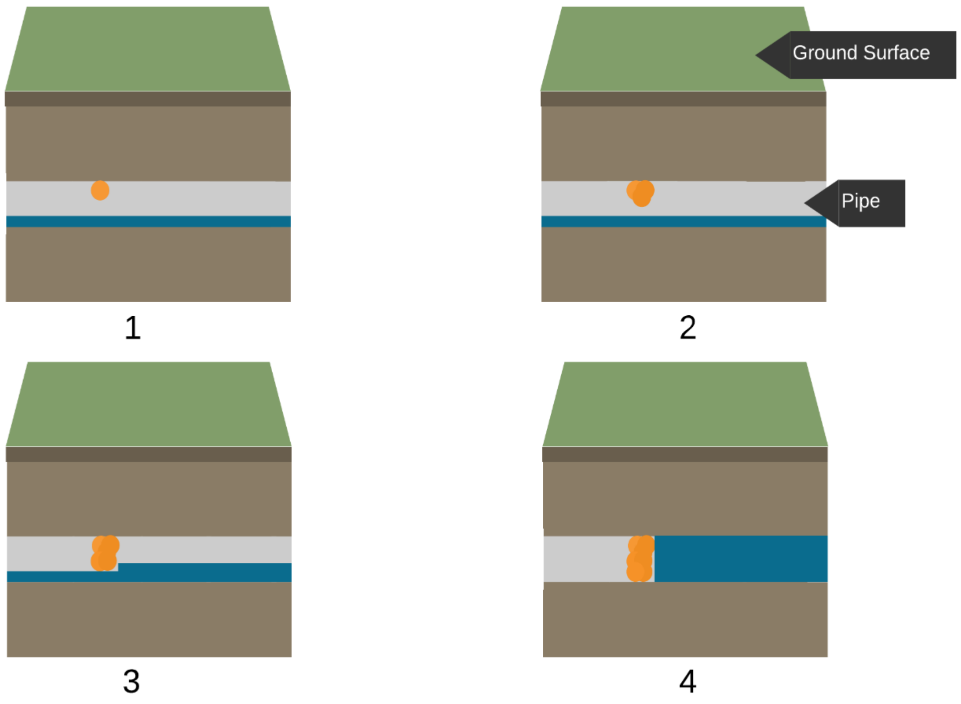


Figure 2-4 – the process of a blockage forming by objects ‘sticking’ together. In this example, the blockage is caused by deposits of *fat* sticking together and growing in size over time. This time progression is shown by the figures in order from (1) to (4).

2.8.2 Risks of Blockages

While we briefly stated the dangers of effluent exposure in Chapter 1, this section will provide additional detail on these dangers to emphasise both the real danger of effluent and the importance of our research. Early detection of blockages will allow resolution before effluent can breach the system and contaminate ground, fixtures, or properties. Following the discussion in this section, it will be evident that this early detection can potentially save lives and natural resources.

Pathogens present in effluent can cause a wide range of disabling or fatal medical conditions. The World Health Organization [17] state that inadequate sanitation is responsible for 280,000 annual deaths. Examples of diseases caused by these pathogens include salmonella, hepatitis A, trachoma, poliomyelitis (polio), cholera, typhoid, and dysentery [8] [15]. Salmonella is responsible for 450 deaths per year in the United States alone [9], and complications range from permanent heart damage [10] to brain damage and paralysis [11]. To provide an example, Australian Monika Samaan was left unable to speak and confined to a wheelchair after contracting salmonella [12]. Another example of the dangers of effluent is Trachoma, the leading cause of infectious blindness affecting 1.9 million people worldwide [13]. Worryingly, Trachoma is highly contagious and can be spread by contact with insects that have touched infected persons or effluent. Australia is the only place in the developed world where Trachoma poses an issue [14].

As previously mentioned, effluent contamination can be *direct* or *indirect* [8]. To reiterate, direct exposure involves physical contact with effluent while indirect contact involves contact with animals or insects who have had direct contact. This has alarming implications as insects are highly mobile, attracted to effluent and commonly land on humans. If one insect has had contact with effluent and lands on a human, this chance encounter could prove fatal or permanently damaging. Many of the aforementioned illnesses, especially trachoma, are easily spread through indirect contact. Combined with the hot conditions and population of flies in Australia, this has potential to create a significant public health concern. This is only worse in developing countries where sanitation is very poor, and polluted conditions combine with the often-hot weather attract an enormous number of flies. These countries often employ manual scavenging [18-19] to resolve blockages, and during 2017 a life in India was claimed every five days from this practise.

2.9 Existing Solutions for Wastewater Management

2.9.1 Robotic Solutions

Following our review of smart city technologies and IoT platforms, we will now examine robotic solutions for resolving wastewater blockages. As the name suggests, robotic solutions involve constructing a remote-controlled robot and deploying it inside wastewater infrastructure to carry out inspection. Robotic solutions reduce risk of exposure to hazardous effluent and are much less expensive than traditional solutions such as CCTV cable and probe inspection. Lower costs allow developing countries to move away from manual scavenging, while allowing developed countries to perform more frequent and simultaneous inspections.

We have reviewed literature detailing development of two separate robotic solutions- the *BhrtyArtana* robot developed by Vaani et al. [49], and an unnamed robot developed by Shrivastava et al. [48]. Both of these solutions were developed in India, likely in response to the endemic practise of manual scavenging described in Section 2.8.2. Despite some differences in implementation, both solutions discussed have a very similar overall design and purpose. Both consist of robots that navigate a sewer pipe and terminals placed at ground level. These terminals have a direct connection with the robots, meaning messages between them do not navigate the public Internet. When a blockage is encountered, the robot will send a message to the ground-level terminal that notifies an observing user. These terminals will subsequently relay messages to other devices or networks as required.

Both robots utilise wheels driven by DC gears motors for movement, however BhrtyArtana uses four wheels while Shrivastava et al.'s solution uses two rear wheels driven by a front castor wheel. Vaani et al. discovered that utilising high-friction wheels increased efficiency and produced more favourable test results, providing another design strategy for future robotic solutions. Another commonality is that both robots use an *acoustic sensor* to navigate pipes and detect blockages, notifying an above-ground terminal of any blockages found.

Vaani et al. do not specify how BhrtyArtana utilises its acoustic sensor to navigate pipes, however Shrivastava et al. have provided a detailed description. While they will not be the same due to being two different robots with different designers, it can be assumed some similarities exist. Shrivastava et al.'s description is also robust and efficient, allowing it to serve as guidance to future researchers and developers.

If an acoustic sensor detects an echo, it is assumed to either imply a corner in the pipe or a blockage. To determine which is occurring, the sensor rotates 90 degrees to the left. If no echo is detected, the pipe has turned to the left. Conversely if an echo is detected, the pipe will rotate 180 degrees to the right. If no echo is detected, the pipe has turned to the right, but if an echo is detected it can be assumed a blockage is obstructing the way forward. Rotation of the sensor and processing of readings is carried out by the control board. The robot's wheels will turn towards the appropriate direction if it is determined the pipe has turned, whereas if a blockage has been detected the robot will notify users through the ground terminal. Shrivastava et al. also state that each vehicle rotation or movement will require acoustic recalculation.

While Shrivastava et al.'s solution is completely passive and only detects blockages, BhrtyArtana includes an actuation component for resolving any blockages found. This is achieved through rotating propeller head of serrated aluminium attached to its front, capable of cutting through any blockages discovered. As the rotating motor has much higher power requirements than the robot's other components, it is powered by a separate high-capacity battery. Shrivastava et al.'s robot also sends readings to the ground station as text, while BhrtyArtana captures live footage of the sewer through an infrared LED camera; this footage is streamed to the ground station. Another extra feature provided by BhrtyArtana is its wireless connection to the ground terminal, whereas Shrivastava et al.'s robot communicates using a long cable.

Despite having no attached camera, Shrivastava et al.'s robot is capable of determining distance it travelled from its deployment point. Each revolution of its wheels triggers an attached reed switch, sending a single pulse to the robot's main microcontroller. The microcontroller then calculates distance using the following equation. In this equation W is the number of wheel revolutions, and r is the radius of rear wheels.

$$Distance\ Travelled = W(2\pi r)$$

Although BhrtyArtana's additional features provide significant advantages over Shrivastava et al.'s robot, they also raise its cost and complexity. One of our key research objectives is to minimise the cost of any system developed, and this goal should be taken even further in developing countries such as India. When taking this into consideration, it can be argued that

Shrivastava et al.'s solution provides as much benefit as BhrtyArtana's; it still detects blockages like BhrtyArtana and is likely to be much less expensive.

All testing showed that Shrivastava et al.'s robot was successful and fit for purpose, providing an inexpensive, safe and effective solution for acoustic monitoring of sewer blockages. Shrivastava et al. provided a detailed description of their testing and its outcomes, while comparatively little was provided for BhrtyArtana. Despite this, Vaani et al. state BhrtyArtana can successfully traverse through a pipe of 10 inches in diameter.

2.9.2 Urban IoT Solutions

While robotic solutions have some advantages over the conventional CCTV and acoustic probe inspection discussed in Chapter 1, the key issue with those methods still remains. Robots can only be deployed in a single asset at any given time, either on a round-robin routine or in response to a reported issue. If an issue has been reported, events noticeable to customers such as foul odour or effluent breaches have already occurred. Ideally, a solution for detecting sewer blockages would exist that is capable of monitoring an entire sewer system simultaneously.

Several systems have been developed that utilise IoT technology to deploy motes throughout wastewater infrastructure, with each simultaneously monitoring local physical conditions. These monitor urban infrastructure and intend to enhance service delivery to citizens while improving processes for municipal governments; therefore, they fall within the *smart city* concept. We will discuss them in the following section as they provide both inspiration and prior lessons for our own research.

Stoianov et al. [53] delivered a system named *PipeNet* in 2007 to assist the United States Government with improving sub-standard water and wastewater infrastructure. PipeNet involved placing a range of heterogenous sensors across infrastructure to monitor physical variables in real-time, with sensors grouped into *clusters* based on overall purpose. We will only consider *Cluster 3* in this review, as it aligns with the goals of our research by monitoring effluent levels. Pipelines named *Combined Sewer Systems* (CSSs) are used in the United States to carry both effluent and stormwater to its intended destination, with overflow collecting in the aptly named *sewer overflow collectors*. Cluster 3's sensors were deployed in these overflow collectors to measure the level of effluent currently overflowing.

Two years later in the United Kingdom, See et al. [52] developed a system to monitor effluent level that was much smaller in scope than PipeNet, but equally as relevant to our research. Sensor motes were deployed in assets named *gullies* across several properties – *gullies* are attached to a property's sewer connection and give effluent a 'safe' place to overflow if needed before breaching to the surface. By deploying level sensor motes in these gullies, See et al. hoped to build a blockage detection system. Reviewing this has significant benefit for us, as this is a very similar goal to our own research.

Much more recently, Saravanan et al. [34] developed a system to improve sanitation in the Indian village of Mori. Mori's water is supplied through a series of canals linked to a central tank, and quality issues would only be identified when observed by villagers – by which time, it could be too late to avoid contamination. Saravanan et al. developed a mote that would be deployed in the central tank and use a range of sensors to monitor different water quality variables. This holds striking similarities the mote deployed by Guibene et al. [32] in Dublin's Liffey River. Saravanan et al. also developed a series of actuation motes to remotely operate locks on Mori's canals, however we will not discuss these as they are out of our research's scope.

Each system we reviewed has the same basic architecture, consistent with what we described in Section 2.1. In relation to the architectures discussed in Section 2.7.3.2, these can all be classified as *Hub Connectivity* with either *Ubiquitous* or *Application-Layer Overly* architectures. More specifically, each uses one or more attached low-power sensors to read a physical variable from their deployed environment. An energy-efficient wireless network infrastructure is then used to send these values to a more powerful base station, which subsequently passes them to a central system for processing.

2.9.2.1 *Mote Hardware*

See et al. used *Crossbow* Mica2 units as motes, operating in the lowest possible duty cycle of 1%. During the other 99% of operational time, the motes would operate in very low power *sleep* mode consuming 20µA of power. However, in the remaining 1%, motes would enter *sensing mode* and read input from attached sensors and evaluate whether a surcharge was occurring. If so, an internal counter of surcharges would be incremented, and if not, this counter would be reset. When the counter reaches five subsequent surcharges, the microcontroller switches to *radio broadcast* mode and transmits an alarm to the base station.

Upon successful message transmission, the subsequent surcharge counter is set back to 0. These motes consumed 9mA of power during sensing mode, and 18mA during message transmission.

Stoianov et al. [53] implement Intel's *Mote* platform, connecting sensors through eight analogue channels with adjustable sample rate. These motes implement a similar duty cycling mechanism to See et al. [52], however the *sleep* mode is less energy efficient and consumes 2mA. Additional comparisons show that PipeNet's motes are less energy efficient overall, consuming 16mA in sensing mode and 30mA during data transmission. Stoianov et al. also discovered that the mote's onboard RAM was insufficient for store-and-forward processing at the required sample rate, and in response developed a *streaming* mechanism to send data between motes. Multiple buffers were introduced to allow for lower throughput and higher latency under difficult conditions, and it was observed that 600 samples could be streamed per second. This is an innovative idea for constrained systems, and our research will take advantage of this if required.

Finally, Shrivastava et al.'s motes were built using the popular *Arduino* board. Each sensor was connected using Arduino's GPIO interface, which was also utilised to communicate with a LoRa transceiver. Less information is provided about these motes; however, Arduino boards are far easier to develop with and often do not require customisation.

2.9.2.2 *Mote Sensors*

Stoianov et al. [53] and Saravanan et al. [34] both deliver motes that connect to a range of heterogenous sensors. Saravanan et al.'s system only consists of a single mote, and this has many sensors that measure different values; Oxidation Reduction Potential, pH, salinity, water level, turbidity, temperature and flow. In contrast, PipeNet involved deploying multiple motes, and these are classified into *Clusters* based on the attached sensors. Sensors in *Cluster 1* measure water pressure and pH, Cluster 2 measures water pressure, and Cluster 3 measures effluent level. For both of these systems, we will solely examine the methods used to measure water or effluent level as other information is out of our research's scope. We will not go into further detail on Saravanan et al.'s method for level detection as it uses conventional acoustic sensing with no modifications. However, both other projects implement unique level sensing methods, which we will discuss below.

While See et al. [52]'s motes only monitor the single variable of effluent level, this is achieved using multiple sensors. The implementation of multiple sensors results from See et al. developing their own custom acoustic sensing mechanism, which they undertook believing commercially available sensors were too delicate for their system's harsh environment. This mechanism has two sensors – both a transmitter and receiver. The transmitter produces sound waves, and the receiver records the amplitude of those waves. If the amplitude is above a certain threshold, it is assumed that both the transmitter and receiver are underwater, implying effluent has reached the transceiver's height. This does not produce a quantitative measure of effluent level like conventional acoustic sensors, and instead returns one of three discrete status; *Low, Medium,* or *High*. See et al. also considered using the conventional *time* domain for ultrasonic sensing, however considered it too prone to error.

Motes in PipeNet's Cluster 3 also employ an array of sensors for measuring the single variable of effluent level. Each mote contains three sensors; two pressure transducers and an ultrasonic sensor. Pressure transducers are placed at a pre-determined height and produce a reading if effluent surcharges at a given threshold. If the difference in pressure readings exceeds a pre-set value, the ultrasonic sensor will be activated and used to determine level. The ultrasonic sensor is only used if absolutely necessary, as it consumes 550mW in contrast to the pressure sensors' 10 mW.

2.9.2.3 *Network Technologies*

Both Stoianov et al. [53] and See et al. [52] utilised short-range WSN communication protocols, while Saravanan et al. utilised LoRa LPWAN technology. While WSNs have a much shorter communication range than LPWANs, they are very energy-efficient and inexpensive. See et al. utilised the popular ZigBee protocol, giving their network *multi-hop* mesh communications and self-healing capabilities. Stoianov et al. instead utilised Bluetooth technology, taking advantage of the *scatternet* formation to allow a higher number of devices per network. While Bluetooth *piconets* typically cannot contain more than 8 devices due to a 3-bit address space, the scatternet formation breaks piconets into hierarchical subnets. This can continue recursively, resulting in a star-of-stars topology.

2.9.2.4 *Base Stations*

PipeNet and See et al.'s systems have extremely similar base stations, both consisting of a *Stargate* mini-computer that receives data from connected motes and sends it to a central system. Motes in these systems both utilise a *store and forward* mechanism, where motes will cache sensor readings and wait until the base station is ready or a pre-defined threshold has been reached before sending them. See et al. [52] state this duty cycling is intended to conserve power, allowing the base station to be in *sleep mode* when not receiving messages. PipeNet builds on this duty cycling further, attaching a *cluster head* mote to the base station to collects input from all others. The cluster head is responsible for caching all messages before periodically passing them to the base station.

Again, Saravanan et al. [34]'s approach is very different, utilising a LoRa gateway for a base station. However, an unexpected similarity can be found with PipeNet in the use of a cluster head mechanism. A microcontroller connected to the LoRa gateway performs a store-and-forward action on data from sensors, manages connection acknowledgements, and administers the network. This is a logical decision, as LoRa gateways are often little more than 'relays' that communicate information between the local network and backhaul.

2.9.2.5 *Results and Evaluation*

See et al. [52] developed eight wireless sensors and deployed them in a densely-populated residential area in Bradford, UK – an area where each home had a gully placed relatively adjacent to its neighbours. The Stargate base station was placed on a lamppost in the street, with the closest sensor located 12.3 metres away and the furthest 66.5 metres away. Additionally, the shortest distance between two sensors was 5.5 metres and the longest 38.5 metres.

During the initial test, only the two sensors with line-of-sight to the base station operated correctly. Following this, two changes were made in an attempt to increase performance; the base station was given a high-gain antenna, and additional repeaters were added 40-70 metres away from it. These changes had the intended effect, allowing five sensors to communicate with the base station. Further investigation after this second test revealed that all three unreachable sensors were obstructed by concrete, rubbish, or other urban detritus. See et al. provided suggestions for improving communications in similar systems, including adding more relay points and base station aerials. Primarily, however, this shows the potentially devastating effect of urban obstructions – a fact we must consider during our own research and development.

Stoianov et al. [53] also performed a small-scale field test on PipeNet and observed similar issues to those encountered by See et al. Problems were also encountered with the Stargate minicomputer utilised, however these will not be discussed as this hardware has since been superseded. Performance results for ground-based antennas or those placed inside underground assets will be highly beneficial to us, as deploying motes in inspection shafts will create similar conditions.

On one occasion, performance was significantly degraded by snowfall covering an antenna. Rainfall also had an impact on performance; however, this was far smaller than that caused by snow. An antenna embedded in the road was accidentally destroyed during resurfacing, and a replacement was installed inside a shaft covered by a cast-iron lid 4.5 inches thick. As could be expected, being installed under a cast-iron lid caused significant reductions in performance. These results show the effects of both urban obstruction and weather conditions, with snowfall and rain forming their own obstructions. In both cases, increased link budget or lower-frequency air interfaces could potentially solve the problem.

A similar incident to the road-embedded antenna's destruction occurred when maintenance staff accidentally destroyed a Cluster 3 antenna during routine inspections of CSS pipelines. These both show the often-overlooked risks to physical security when implementing an urban IoT system, especially in areas prone to construction work or heavy human activity. Installing motes in off-limits, locked, or inaccessible places can rectify this problem, however if these assets are constructed with thick materials the issue of obstruction may occur again. Additionally, installing motes in inaccessible places results in more difficulty replacing batteries or performing maintenance.

87% of Cluster 1 messages were received, as were 62% of Cluster 2 messages and 72.3% of Cluster 3's messages. The lower percentage of received messages in Clusters 2 and 3 was likely caused by the breakage incidents and subsequent attempted fixes that negatively impacted message reliability. Stoianov et al. also stressed the issue of time synchronisation, stating that more accurate synchronisation should be a goal of future research. Keeping this in mind, we will aim for effective and accurate timekeeping.

Saravanan et al. [34] did not provide detailed information on their system's network reliability, however the paper produced implies that deployment was very successful, and information was received as appropriate. This is further supported by the fact they were able to produce a logistic regression model in the Weka data mining suite that produced 99% predictive accuracy. This potentially demonstrates an advantage gained by using LPWAN technology, a logical conclusion when considering the higher link budget, low bandwidths, and LoRa's spread-spectrum capabilities.

2.10 Conclusion

In this chapter we have reviewed and presented literature detailing current smart city systems alongside the technologies and initiatives that enable these systems, with an additional focus on LPWAN platforms. LPWAN technology provides low-power communications capable of allowing motes to connect and communicate over long distances. We also reviewed the techniques currently used for detecting sewer blockages, detailing how they operate while revealing problems and inefficiencies. Considering these issues with current techniques, and the potential of IoT-based smart city systems, we are encouraged to develop an intelligent sewer blockage detection system utilising IoT infrastructure. Current literature has revealed that no practical solution has yet been developed for doing this. In the following chapter, we present the design and implementation of such a system.

Chapter 3 Design and Implementation of Monitoring and Detection System

The first chapter showed the severe consequences of sewer blockages and need to resolve them in a manner both timely and efficient. Following this, the second chapter revealed no solution exists that can practically detect sewer blockages over a wide area. Many existing solutions employ short-range devices, requiring a large number of repeaters or base stations to effectively deploy over a wide area. Others are too ambitious, using complex and expensive sensors to detect blockages or monitoring a range of additional variables.

Our research intends to deliver a system capable of monitoring sewer blockages across a wide area, while remaining sufficiently simple and cost-effective for deployment in areas of all economic status. In this chapter, a detailed design capable of carrying out these intentions will be presented. This design encompasses custom hardware, firmware, and software working with communications protocols across all layers of the network. Chapter 2 also reviewed previous literature and technical specifications for a range of protocols and services across all layers of network architecture – LoRa, MQTT, and the *Node.js* framework. These protocols and services will prove instrumental in the design presented here.

The system's primary function is to determine when a sewer blockage is occurring and notify relevant persons. Sewer blockages are indicated by surcharges occurring at property connections, so logically, detection can be carried out through motes placed at these connections. These motes must incorporate a sensor capable of reading a physical variable indicating surcharge, alongside the typical components of microcontroller and wireless transceiver.

Using the states of all motes across the wastewater system, the central system will be capable of:

- Differentiating an actual blockage and a 'false alarm';
- Classifying blockages as either full or partial;
- Determining whether blockages are located at the property connection or in the main itself;

- Locating the property connection, or length of main between two property connections, where the blockage is occurring;

An interface provided by the central system will also expose this information to external systems, including interfaces to communication platforms (e.g., SMS or Voice APIs) and other corporate systems. Corporate system integration opens a wide range of possibilities, including synthesis of business intelligence and optimisation of existing processes. Examples of systems that can be integrated include weather services, asset management systems, business intelligence platforms, and other control or instrumentation solutions.

3.1 Overall Design

Existing literature shows that urban IoT systems conform to an overall design as illustrated in Figure 3-1 where many distributed motes communicate with a single central system. This has been previously described in Sections 2.1 and 2.7.3. Our system can be separated into two logical components (i – motes and ii – central system), separated by the public Internet allowing communications. For the system to be practical, communications to and from motes must be wireless.

Being placed throughout an entire wastewater infrastructure, motes will often be distributed over a long distance and sometimes be placed sparsely. Depending on the property owner's individual preferences and economic status, motes will be capable of connecting to the central system using one of two wireless platforms as shown in Figure 3-2 - *Wi-Fi* or an LPWAN. Both platforms are simply methods for connecting a mote to the public Internet, where it can be routed to the central server. Our network will be capable of supporting motes using both platforms, and all motes will co-exist seamlessly regardless of air interface. Messages at the central system from both platforms will be indistinguishable from one another.

Many utility providers utilise a home or business' own Wi-Fi gateway to provide connectivity to Internet-connected devices such as smart meters. Our system's motes will send and receive an extremely small amount of data per month, greatly reducing customer concerns regarding congestion and data allowance. Many urban areas also provide free Wi-Fi connections that can be utilised, and existing IoT systems have also taken advantage of this. Using existing Wi-Fi connections conforms to a *crowd-sourcing* model of network deployment and removes the often-high costs of deployment and maintenance from system providers. However, this leaves network uptime and reliability in the hands of network owners – albeit at potentially reduced legal obligation.

Wi-Fi crowdsourcing is practical in densely populated areas of wealthier cities, however more rural areas, or cities without funds for public Wi-Fi will make this model impractical. In addition, some homes and businesses will refuse to allow utilities companies to share their connection – a decision they have every right to make. For these circumstances, an LPWAN will be deployed over the covered area. LPWAN networks have been designed for use in IoT applications, and embody this trade-off by offering energy-efficient, highly scalable long-distance communications with very low data rates. While LPWAN networks provide a very long range and low power consumption, they are less reliable and slower than Wi-Fi networks.

In large urban areas, it is likely that there will be a range of motes connected to both LPWAN and Wi-Fi systems. Variation can arise from customers not willing to share their Wi-Fi connection, black spots, outages, and variations in socioeconomic status. This must be facilitated wherever possible, meaning all motes that connect to a Wi-Fi access point must also be capable of connecting to an LPWAN.

3.2 Considerations for Mote Design

Before developing a functional design for our system's motes, it is important to consider key issues known to adversely impact motes or battery-powered devices in other systems. A functional design is concerned with how a system will carry out its required tasks, however, often fails to consider the practicality and sustainability of carrying these out. Without considering these issues, our motes could send spurious data, report inaccurate readings, or drain all their power within days or even hours.

In this section we will discuss each identified issue, and our design in section 3.3 will propose a solution.

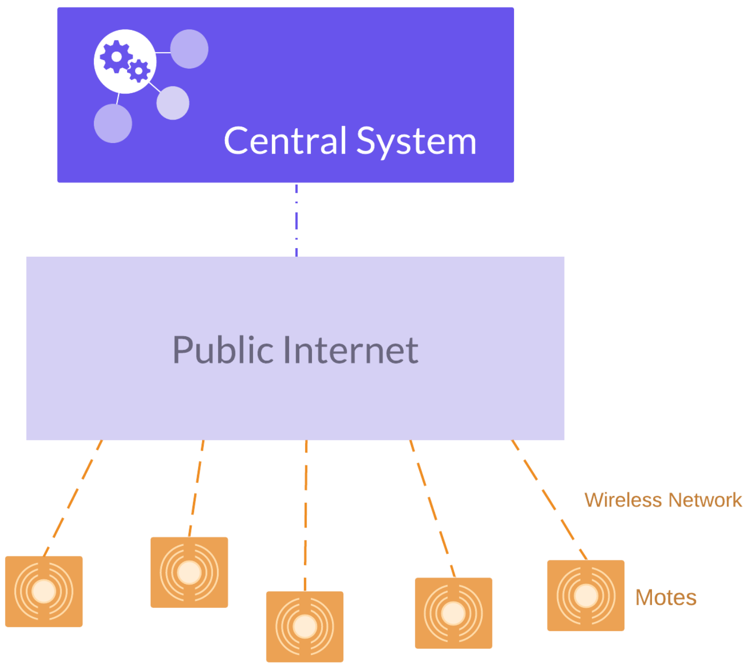


Figure 3-1: The overall design of urban IoT systems described in existing literature. Several *motes* connect to a central system responsible for communicating with all motes and collectively processing data. Motes connect to the central system using the public Internet, and their physical connection to the public Internet must be facilitated with a wireless network platform. The central server, however, does not follow the same restrictions and can use a wired connection if needed.

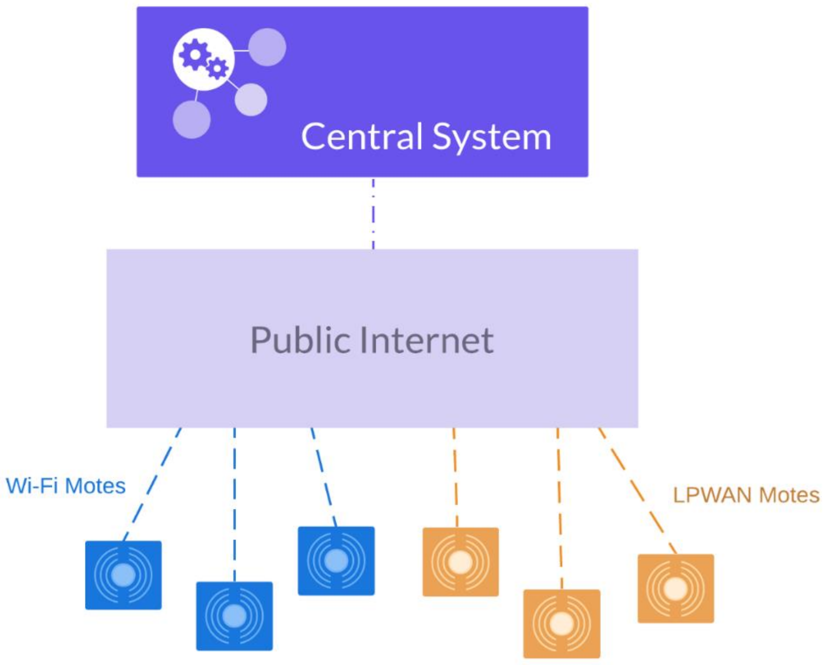


Figure 3-2: A further refinement of Figure 3-1, showing the general urban IoT architecture adapted to our system's 'dual network' configuration of co-existing Wi-Fi and LPWAN networks.

3.3.1 Power Levels

A mote operating continuously at normal power consumption will quickly drain its battery, lasting days or even hours instead of the years expected from IoT systems. If a mote detects changes in a *General-Purpose Input/Output* (GPIO) input through constant polling, this will require the mote to constantly be running at full power. If we determine an alternative to polling, the mote will no longer be constantly inspecting its GPIO pins and can be considered *idle* when not doing so. Examples of this have already been seen in Sections 2.7.2 and 2.9, where *sleep* or *low power* modes were commonly used as a power-saving measure. Low power modes are made practical by specialised processor instructions called *Interrupt Requests* (IRQs).

IRQs are very high-priority instructions to the CPU, which when received cause it to cease any lower-priority instructions and start executing a particular sequence of code. This sequence of code is named an *Interrupt Vector* and can be mapped to a particular pin in some microcontrollers, causing certain state transitions on that pin to request the code's execution. Additionally, some microcontrollers are capable of mapping IRQs to the internal clock, firing IRQ events at a specific time or when a programmed countdown timer elapses. Embedded operating systems or firmware maintain a data structure named an *Interrupt Vector Table* which maps all IRQs to a pointer to the correct vector.

Many microcontrollers offer various low-power modes of operation that each consume less power and disable more features. 'Deep sleep' modes are often available, where microcontrollers consume very little power but are almost completely inert and are unable to perform normal processing. However, some microcontrollers are capable of reacting to IRQs from GPIO state changes while in deep sleep. A powered-off microcontroller cannot monitor surcharge, however a deep-sleep microcontroller can while using very little power.

3.3.2 Switch Bounce and Environmental Fluctuations

Another issue that must be considered is *switch bouncing*, occurring when a switch's contacts bounce off each other when being pushed together [64] - a simple problem of physics. Bouncing causes a switch to rapidly fluctuate between open and closed states before finally settling at the intended state. In a digital system, this causes many fluctuations between 0 and 1 before settling at the final value.

The system under development utilises a *float switch* to determine whether surcharges are occurring. Float switches utilise a simple mechanism that is pushed up by a rising water surface, pushing two contacts together to close a circuit – these will be discussed in greater detail later in this chapter. Our system sends a message to the central server whenever the float switch's state changes. If the float switch bounces and rapidly changes state several times, theoretically each of these changes could result in a message being sent. As message transmission consumes a significant amount of power, unnecessary messages resulting from switch bounce can cause significant power wastage. In more concrete terms, theoretically power usage could increase by the amount of state changes involved in a given bounce.

Our system is also at risk of another type of bouncing – one that is not an error in the switch, but instead an attribute of the mote's own environment. Contacts close the float switch when the effluent's rising surface pushes them together. In an inspection shaft, effluent is prone to 'bobbing' or rapidly fluctuating level by slight amounts, which triggers rapid state changes. Unlike switch bounce, this is not a false reading, but a real reading the system must attempt to filter.

Switch Debouncing is a technique utilised to filter out the rapid state changes caused by switch bounce and ensure only the **final intended state** is processed by the system. Debouncing can be implemented by both hardware and software, however we will utilise software debouncing for the system under development. Software debouncing is not only less expensive and simpler to implement, but also allows real 'bobbing' state changes to be filtered. Software debouncing often involves delaying execution for a pre-determined length of time after a state change to 'wait out' any switch bounces.

3.3.4 Pin Floating

For each digital pin, microcontrollers consider a specific range of voltages *high* (1) and another range *low* (0). There is often a space between these ranges where a reading cannot be considered *high* nor *low.* A pin with voltage in this intermediate range is said to be *floating,* and this presents a common issue for electronic design. Ideally, this *floating* range should be avoided whenever possible. Microcontrollers react unpredictably to pin *floating*, however a common behaviour is rapid fluctuation between 0 and 1.

Pins not connected to a voltage source or ground are prone to *floating* as they conduct electromagnetic noise such as radio waves and static electricity. This makes sense, when considering that an exposed pin is in essence a very small antenna. Electrical noise is capable of pulling pin voltage into the high or low ranges, however, can also pull it into the floating space. Open switches are also prone to floating, as an open switch is electrically equivalent to an exposed pin.

Our system's float switches will spend the majority of their time open, when their respective inspection shafts are not surcharging. While open, these float switches will be prone to the electrical fluctuations caused by floating. This is dangerous as each fluctuation can potentially trigger a message transmission to the central system. If messages are constantly sending while float switches are open, motes could exhaust their batteries in mere hours.

Two solutions are commonly used to remedy switch floating, and these are discussed below.

The first and simplest solution is simply connecting the switch to ground, pulling an open switch to 0 while still allowing closed switches to return 1. However, any direct connection between voltage and ground results in a short circuit, which can instantly destroy or cripple microcontrollers. Placing a resistor between voltage and ground will prevent a short circuit occurring, while allowing the technique to remain effective. This is referred to as utilising a *pull-down resistor*. Figure 3-3 illustrates the problems with the initial solution, alongside how this is solved with a pull-down resistor.

Conversely, a direct line can be placed between voltage source and GPIO, with the float switch connected to both that line and ground. When the switch is open, current will flow directly from the voltage source to GPIO, registering a 1. Closing the switch will direct the current to ground and away from GPIO, causing it to register a 0. Again, a resistor must be placed between the voltage source and ground to prevent a short circuit, lending this technique its name of *pull-up resistor*. As can be observed, pull-up resistors reverse the values returned by opening and closing a switch. Figure 3-4 again illustrates the initial solution's problems and shows how pull-up resistors provide a solution.

Many microcontrollers include embedded pull-up and pull-down resistors for GPIO pins; however, it is important not to make this assumption without checking technical specifications.

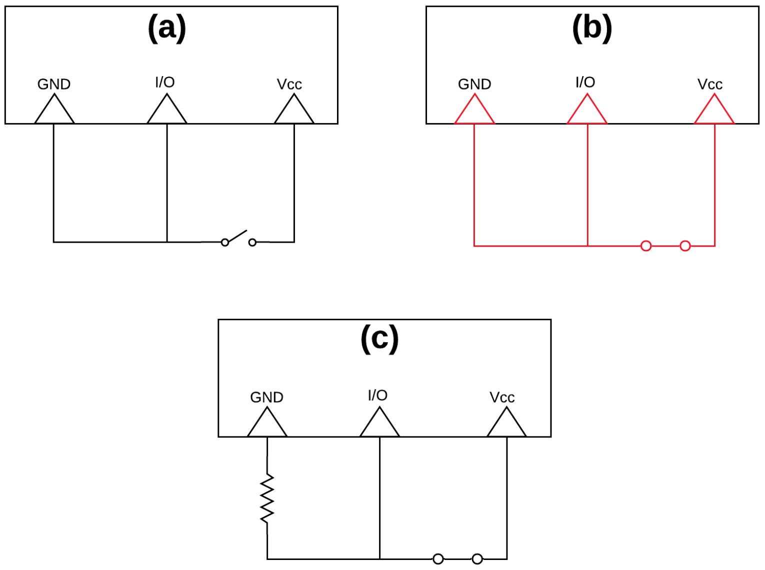


Figure 3-3: Demonstration of a pull-down resistor. In (b), closing the switch allows current to flow uninhibited from Vcc to GND – this creates a short circuit. By adding a resistor to the connection between the switch and GND as seen in (c), the current becomes inhibited by the resistor and short-circuiting does not occur.

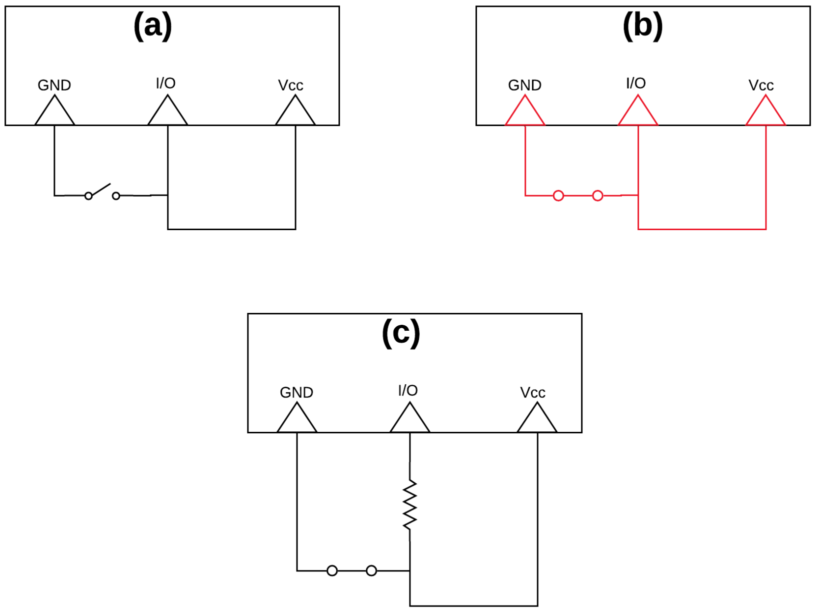


Figure 3-4: Demonstration of a pull-up resistor. In the (b), closing the switch creates an uninhibited current between Vcc and GND – resulting in a short circuit. When a resistor is placed between Vcc and I/O as seen in (c), this inhibits the current and prevents short circuiting. While the connection between the switch and GND has no resistor, Ohm's law proves the current will be the same at all points of the circuit.

3.3 Mote Design

In this section we present a design for the motes our system will deploy throughout wastewater inspection shafts. This design is built around meeting the system's core requirements and carrying out the expected functionality, while providing solutions to the issues raised in section 3.2

Each mote will consist of four key components as illustrated by Figure 3-5; i – *power supply*, *ii – surcharge detection sensor*, *iii – wireless transceiver*, and iv – *microcontroller*. Boundaries between each of these components are often blurred, as a greater number of modern solutions integrate multiple components on a single board. Examples of this include the *Raspberry Pi* and *Arduino* solutions, both of which contain microcontrollers and integrated Wi-Fi adapters.

Some existing systems detect surcharge by utilising a level sensor to measure the level of effluent in a given asset, and represent it using an analogue float or integer value. However, the presence or lack of surcharge can also be easily represented through a binary value – *surcharge* (1) or *no surcharge* (0). Our system is only concerned with whether a surcharge is occurring or not, as opposed to the water's actual level. Using binary values requires significantly less computational, storage, network, and power resources than utilising numeric values. This results in less expensive systems with faster processing and increased communications range.

Utility providers responsible for wastewater infrastructure are responsible for deciding which level in a given inspection shaft indicates a surcharge. Inspection shafts can be classified according to shape, location, property connection angles and other factors, with a single level determined for each classification. Our system's motes must utilise a simple binary sensor capable of determining if at any given time, surcharge effluent is at this level.

Thankfully, *float switch* sensors fit perfectly into our requirements. These have a long history of use in the water and wastewater industries, however, have traditionally been connected to large SCADA or instrumentation systems through a wired connection. These systems are expensive and have seldom used wireless sensors, so it would be unrealistic for any utilities providers to integrate every property's inspection shaft into them. Our research will provide further innovation by using these switches for sensing in a low-cost system deployed separately to SCADA environments.

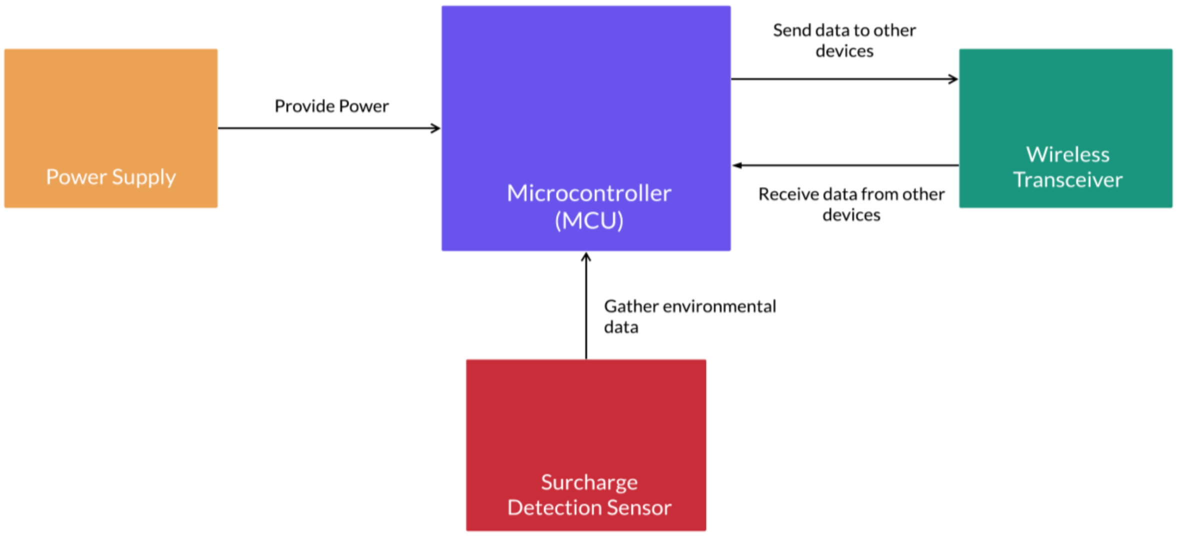

Power Supply
Provide Power
Microcontroller (MCU)
Send data to other devices
Receive data from other devices
Wireless Transceiver
Gather environmental data
Surcharge Detection Sensor

Figure 3-5: A graphical overview of a wireless mote's typical components and their interactions.

Float switches are simple digital switches that close their circuit when rising water pushes contacts together. These contacts will remain closed while water is at or above their placed level, and only open again when water falls below it. Our system will position float switches inside inspection shafts, so their contacts are at the level indicating surcharge. Contacts will push together and close the circuit when this level is reached and separate to open the circuit when effluent falls below it again. Figure 3-6 shows a float switch at its initial *open* state, while Figure 3-7 shows it when a surcharge causes it to *close*.

Advantages of float switches include their inexpensiveness, wide availability, simplicity of implementation and binary output. It should be noted that during our research, we observed that cheap float switches were likely to break from regular handling. However, during a practical deployment, float switches will experience far less human contact as they are placed in the shaft. Considering that the advantages outweighing disadvantages, we have selected float switches for our mote's sensor component.

Float switches will be connected to the mote's microcontroller using two pins. A digital GPIO pin configured in *output* mode will be connected to the positive end, providing power to the switch's circuit. The negative end will be connected to a digital GPIO pin configured in *input* mode, responsible for receiving the switch's digital value. This digital value will differ depending on the switch's state, as an open switch will return a different value to a closed switch.

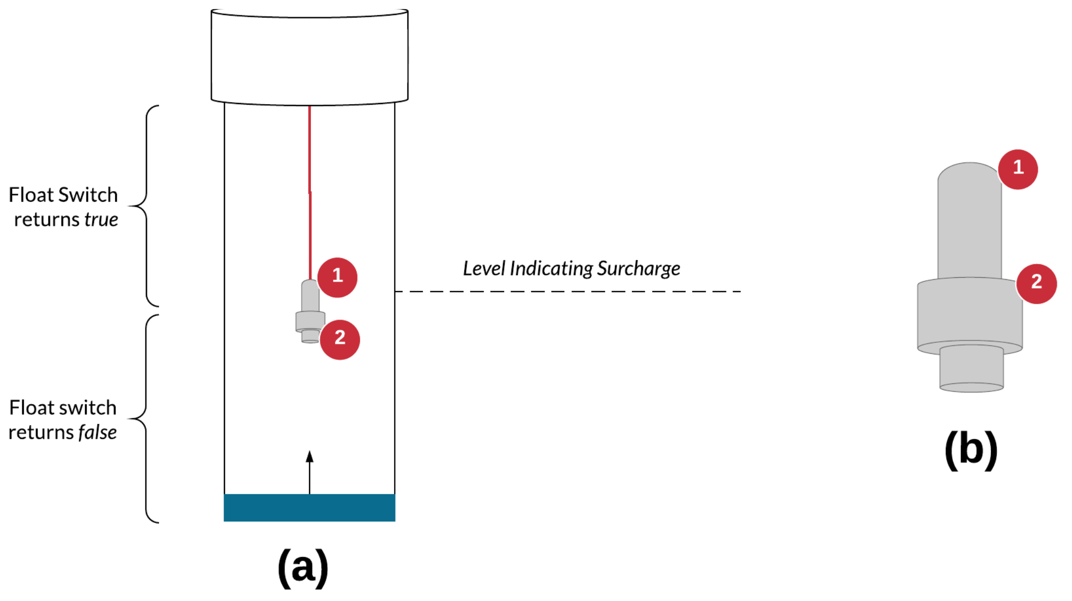

Float Switch returns *true*
Float switch returns *false*
Level Indicating Surcharge
1
2
(a)
(b)

Figure 3-6: A float switch placed in an inspection shaft, during an inspection shaft’s default *non-surcharged* state. (a) shows the float switch deployed in an inspection shaft, while (b) shows the float switch alone. The float switch’s contacts are also denoted by (1) and (2). While effluent in the pipe stays below the level indicating surcharge, gravity will pull (1) and (2) apart.

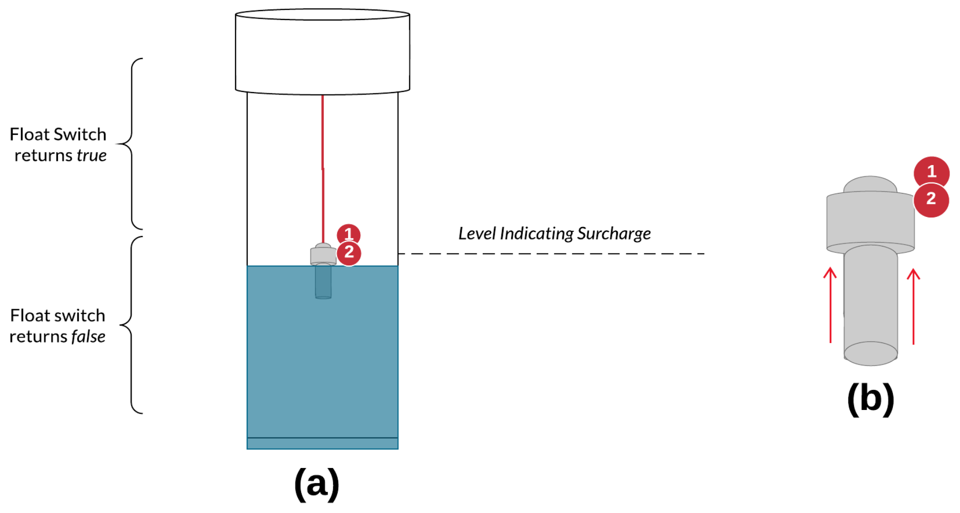

Float Switch returns *true*
Float switch returns *false*
Level Indicating Surcharge
1
2
(a)
(b)

Figure 3-7: A continuation of figure 3-6, showing the inspection shaft when effluent has reached the level indicating surcharge. The effluent’s surface has pushed the contacts shown in (1) and (2) together, causing the circuit to close. The red arrows show the range of motion permitted for the contacts.

If a pull-down resistor is used, a value of 1 will indicate a surcharge and 0 will indicate no surcharge. The opposite applies if a pull-up resistor is used, where 0 will indicate surcharge and 1 will indicate no surcharge. We selected microcontrollers with a pull-down resistor, as pull-down resistors are easier to implement and do not swap the switch's input values.

Considering this, we can develop a state machine for each mote as seen in Table 3-1; each state change is caused by a change in the float switch's GPIO channel. This state machine is also illustrated by the *State Machine Diagram* in Figure 3-8.

To conserve power, our system's motes will normally operate in deep-sleep mode and only 'wake up' when a specific IRQ is raised. Many microcontrollers raise IRQs when certain pins change state, alongside raising IRQs when the internal clock reaches a certain time or specific a period of time elapses. We can utilise this functionality to optimise our system's power consumption, ensuring motes only wake from deep sleep when a specific IRQ is raised. The correct IRQs will be mapped to an interrupt vector that carries out the appropriate functionality before placing the mote back into deep sleep. This results in a paradigm where at any given time, motes are either in deep sleep or reacting to an IRQ.

The mote's float switch will be connected to an interrupt-enabled pin, allowing the microcontroller to raise an IRQ when the switch opens or closes – meaning an IRQ is raised when surcharge states changes. This IRQ will be mapped to the vector detailed by pseudocode in *Pseudocode 3-1*, responsible for processing surcharge status and sending an update to the central system if necessary.

While battery level information will be sent to the central system every time surcharge state changes, this is not sufficient to ensure system uptime. Some motes will rarely experience surcharge, with long gaps between events. In these cases, battery level will fall unacceptably low, or even empty before the central system is made aware. If the battery's charge is completely drained, the central system will never be aware of surcharges or receive information from that mote.

Heartbeat messages regularly sent by each mote ensure the central system is kept aware of battery level. As is common in the IoT domain, this presents a trade-off between *freshness* of mote state information and energy efficiency. Increased frequency of heartbeat messages results in greater system reliability, as the longest time a mote can be down without knowledge is equal to the time between heartbeats. However, this will also consume additional power as motes transmit messages and wait for acknowledgements more frequently.

To facilitate this, we will develop an interrupt vector capable of sending a heartbeat message to the central system. This vector will be mapped to a timer configured with the microcontroller's internal clock, counting down the time between each heartbeat. When this timer elapses, the microcontroller clock will raise an IRQ to execute the appropriate vector. Sending of unnecessary messages (and therefore unnecessary power consumption) will be prevented by resetting this counter if a surcharge occurs. There is no need to send a heartbeat message if a surcharge message containing battery level has recently be sent. Pseudocode 3-2 details the functionality of the IRQ vector mapped to this timer interrupt.

Table 3-2 summarises the IRQs utilised by our motes. All functionality will be carried out by the vectors mapped to these IRQs, and the mote will spend the remainder of its time in deep sleep.

Table 3-1: Mote state machine, expressed as the sum of all possible state changes.

ID	Initial State	Event	New State (1)
A	Not Surcharged (0)	Water is below surcharge threshold and reaches it	Surcharged
B	Not surcharged (0)	Water is below surcharge threshold and exceeds it	Surcharged (1)
C	Surcharged (1)	Water is above surcharge threshold and falls below it	Not Surcharged (0)
D	Surcharged (1)	Water is above surcharge threshold and falls below it	Not Surcharged (0)
E	Surcharged (1)	Water is at surcharge threshold and exceeds it	Surcharged (1)
F	Surcharged (1)	Water is above surcharge threshold and falls to it	Surcharged (1)

Table 3-2: IRQs and vectors utilised by motes.

Trigger/Interrupt	**Purpose**	**Detailed in**
Float switch state changes	Send new surcharge state to central system	Pseudocode 3-1
Internal clock countdown elapses	Send heartbeat message to central system	Pseudocode 3-2

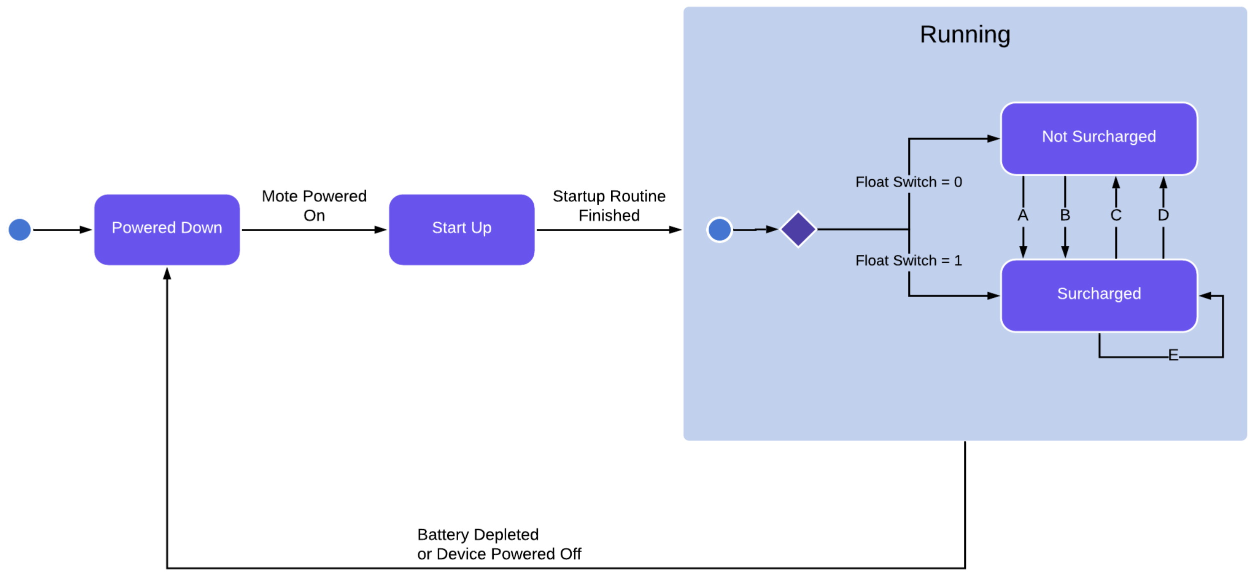


Figure 3-8: A graphical depiction of the mote's state machine. Alongside the previously detailed transitions between ***surcharged*** and ***not surcharged***, this also shows further operational states. When first purchased, a mote will be *powered down*. Powering it on by activating a switch or connecting a power supply will transition it to *Start Up*, where it performs all necessary start-up tasks such as binding IRQs to vectors. Once this has finished, it will transition to the *running state* where it will remain until powered down again. The *running* state can initially be in the *Not Surcharged* or *Surcharged* substate, depending on the deployed inspection shaft's current state. Following this, the mote will transition between *Surcharged* and *Not Surcharged* as detailed in Table 3-1.

Pseudocode 3-1: Process undertaken by IRQ vector when surcharge state changes.

```
//Debounce switch input. Period of time should be configurable.
wait(debounce_time);
var status = read_float_switch();

if (status == last_reading_sent)
{
    go_to_sleep();
}
else
{
   var to_send = marshall_for_server(status, battery_level);
   var to_send_encapsulated = packet_encapsulate(to_send);

   //How many times to try sending data to central server. Example is 10.
   var how_many_times = 10;

   //How many attempts have been made to transmit data
   var attempts = 0;

   //Set to true when message is successfully acknowledged
   var acked = false;

   while(!acked || attempts < how_many_times)
   {
      /* Assume the attempt_to_send method returns true if successful
         and false if unsuccessful or acknowledgement times out */
      acked = attempt_to_send(to_send_encapsulated);
   }

   /* Only update last value sent to server and restart heartbeat countdown
      if message was successfully sent */
   if (acked)
   {
      //The values/results of these statements are kept while device is sleeping
      last_message_sent_to_server = status;
      restart_heartbeat_countdown();

   }
   go_to_sleep();
}
```

Pseudocode 3-2: Process undertaken by IRQ vector when heartbeat countdown elapses.

```
var to_send = marshall_for_server(battery_level);
var to_send_encapsulated = packet_encapsulate(to_send);

//How many times to try sending data to central server. Example is 10.
var how_many_times = 10;

//How many attempts have been made to transmit data
var attempts = 0;

//Set to true when message is successfully acknowledged
var acked = false;

while(!acked || attempts < how_many_times)
{
   /* Assume the attempt_to_send method returns true if successful
      and false if unsuccessful or acknowledgement times out */
   acked = attempt_to_send(to_send_encapsulated);
}

restart_heartbeat_countdown();

go_to_sleep();
```

3.4 Network Design

Before arriving at the central system, messages must be sent over the air to a *base station.* Base stations are collection devices consisting of either a Wi-Fi *Access Point* (AP) or LPWAN gateway, depending on the air interface chosen. Figure 3-9 further develops the architecture shown in Figure 3-2, showing the relationship between motes, base stations, and the central system. Base stations have no transformative effects on the information sent, simply encapsulating it into appropriate transport-layer PDUs and routing it to the central system over the public Internet backhaul. Conversely, data sent to the base station over the public Internet will have transport-layer data removed be forwarded to the correct device.

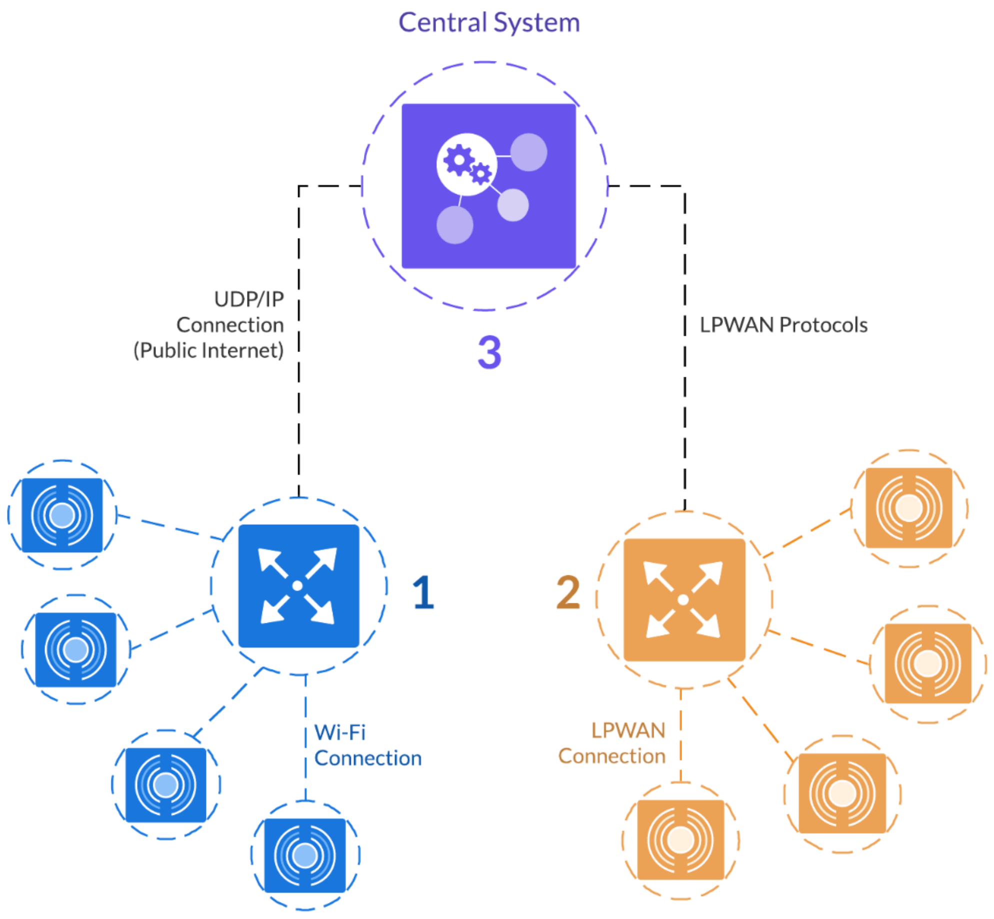


Figure 3-9: A further elaboration of Figures 3-1 and 3-2, showing the relationship between motes, gateways, and the central system. Wi-Fi APs (1) and LPWAN Base Stations (2) connect to the central system (3) using a TCP or UDP/IP connection, however the more lightweight UDP is often recommended for IoT. Each mote will be connected to its own gateway using either Wi-Fi or LPWAN air interface, however each gateway's connection to the central server will involve standard Internet protocols (if not the same). Therefore, the only technical distinction (from a networking perspective) is the air interfaces used by each mote to connect to their gateway.

We will not provide a detailed discussion on Wi-Fi in this paper, as its specifications and implementation methods have been discussed ad nauseum in previous research. Wi-Fi is prominent and ubiquitous in our modern world, with almost every home having at least one compatible device. Consequentially, a wealth of resources are available for establishing Wi-Fi networks, with many single-board computers or motes having antennas built in during manufacture.

However, LPWANs are yet to reach this level of permeation, with commercial networks only now beginning to enter the market. There is much less knowledge about LPWANs widely available, and the term is hardly mainstream. We discussed the LoRa platform in Section 2.3, and for reasons discussed therein we have chosen this platform to implement in our design. Chapter 2 also discussed both MQTT and MQTT-SN *publish-subscribe* protocols and their suitability to IoT systems. MQTT has been described by several studies as well-suited to IoT, and MQTT-SN builds on this by decreasing bandwidth and processing requirements. For these reasons, we will implement the MQTT-SN and MQTT protocols for application layer communications.

Sections 3.4.1 – 3.4.3 will discuss how these protocols are bought together across multiple layers.

3.4.1 LoRa Configuration

We discussed many of LoRa's configuration parameters in Chapter 2, and how they can be adjusted by developers. In this section, we will briefly address the values we have selected for these parameters and how they are implemented for our system. Our selections have been based on our system's unique requirements, and how we have tailored our system to best fulfil them.

LoRa can theoretically utilise any bandwidth between 7.8 and 500 kHz, however Finnegan and Brown [67] note that only 125, 250 and 500 kHz are used in practise. Higher bandwidths result in a higher data rate and greater resistance to interference, but conversely lower communications range. Our system should select the lowest practical bandwidth to maximise communications range, while still maintaining an acceptable data rate. Weighing the high requirements of range and obstacle permeation with the low speed requirements, a 125 kHz bandwidth will be utilised. This also has the advantage of being the most commonly studied bandwidth in existing literature

Table 2-1 outlined the effects of adjusting spreading factor, with higher spreading factors lowering bit rate but increasing link budget. This is supplemented by Table 2-2, which presented that increasing spreading factor decreases the number of messages that can be sent per day. Considering the data provided, it would be sensible for spreading factors equal to or less than 10 be used, with 11 only being suitable if distance or urban obstruction made lower link budgets completely unacceptable. We therefore recommend a spreading factor of 10, with 11 being a contingency option in remote or highly obstructed environments.

If we use a 125kHz bandwidth and spreading factor of 10, our motes will be capable of sending 122 symbols per second and 66 messages per day. Conversely, using the same bandwidth, a spreading factor of 11 will be capable of sending 61 symbols per second and 30 messages per day. This is quite a significant loss of performance and reinforces our statement that 11 is strictly a contingency option.

Chapter 2 also discussed the taxonomy provided by LoRa for connected nodes depending on how often their receive windows are open. Despite Class A being the most power-efficient choice, we will configure motes to be Class B with a single RX window open each day. This allows for future versions or individual implementations of the system where remote configuration or over-the-air updates might be required.

Table 3-3 reviews the configuration parameters we have selected thus far for our LoRa implementation.

Table 3-3: Required configuration for system LoRa air interface.

Carrier Frequency	2.1GHz, 5.0GHz or 443 MHz
Bandwidth	125 kHz
Spreading Factor	10
Device Class	Class B

3.4.2 MQTT and MQTT-SN Configuration

Utilising the previous discussion of MQTT-SN and the system's design goals, we can define the entire range of messages sent between mote and base station from an application level. All data will be sent through an MQTT-SN message, and this will be the same from both Wi-Fi and LPWAN notes. The reason for this is obvious – the only distinction between Wi-Fi and LPWAN motes is the modulation used to carry messages from mote to base station.

A connection between mote and MQTT broker must be established each time a mote is powered on or 'wakes' from low-power mode. All messages sent using this connection must be translated from MQTT-SN to standard MQTT by the MQTT-SN GW, which will forward all received messages to the broker. Consequentially, all messages destined for the broker will be sent to the MQTT-SN GW, and their destination address must be that of the MQTT-SN GW. The MQTT-SN GW will be aware of the broker's address, and simply forward translated messages. Establishing this connection is achieved through the message exchange detailed in Table 3-4.

MQTT requires each client to be uniquely identified with a string between 1 and 23 characters, which is represented with between 8 and 184 bits. To ensure each mote has a unique identifier that will always fit within these limits, we will simply use the primary key of that mote's database record. Primary keys often consist of a simple integer value and given the potentially massive number of motes this could eventually reach a very high number. We will plan for a maximum of seven characters identifying motes in each MQTT message, as this will allow 9,999,999 unique motes. This consequentially results in a maximum ID length of 56 bits.

To utilise the LWT functionality discussed in Section 2.4, a message under the topic '*lost'* will be published with the abruptly disconnected mote's unique ID. The application server will subscribe to the *lost* topic and process it accordingly by notifying system administrators. MQTT-SN packet headers can also vary in length, with the *Length* field being between 1 and 3 bytes. As our solution is unlikely to exceed 256 bytes of packet data, we will use a single byte to store message length.

When a surcharge occurs, a mote will publish both *state* and *status* information under the 'surcharge' topic. The PUBLISH message sent by the mote to facilitate this will contain the following data:

- Client ID (56-bit);
- Timestamp;

- Battery level;
- The current surcharge status;

A 64-bit integer will be used for storing timestamp as seconds since the Unix epoch, with 64 bits used to avoid overflow in the year 2038. While optimistic, it is prudent to ensure systems are viable for as long into the future as possible. Battery level will be stored as an 8-bit integer representing percentage, as an 8-bit value is the shortest capable of reaching 100. Following this, a single bit will be required for storing the surcharge status – 1 indicates a surcharge is occurring, while 0 indicates no surcharge is occurring. Each of these fields will be separated by a single 8-bit character representing a space, resulting in a 153-bit payload. To make the payload evenly divisible into bytes, seven trailing zeroes will be added to the end – bringing its length to 20 bytes.

Heartbeat messages contain all fields found in surcharge messages, with the exception of *surcharge status*. As surcharge status is not included, the message does not require the 8-bit space character following battery level. This results in a smaller payload of 144 bits, which evenly divides into 18 bytes. Because of this even division, no trailing messages are required.

Finally, when a mote re-enters low power 'sleep' mode after it has finished message transmission, it must gracefully disconnect from the Broker. An unexpected or 'ungraceful' connection will cause that mote's LWT mechanism to activate, falsely notifying the application server of an error.

Tables 3-5 – 3-7 outline message exchanges between motes and the MQTT broker facilitating each of the previously discussed communications. Each message's length is determined by adding the previously discussed payload lengths to a 2-byte header and message variable part. The lengths of each message type's variable part can be found in the MQTT-SN specification [83] As discussed in Section 2.4.2, messages shorter than 256 bytes only require a 2-byte header. The first of these bytes is the message's length, and the second is the message type. Each message sent by our system is shorter than 256 bytes and will therefore have a 2-byte header.

With the entire range of communications defined from an application and transport-layer level, we can now shift focus to the network layer of communications and below. This involves the routing, packetisation, and modulation of data. As backhauling over the public Internet will be the same across data from any type of mote, we must specifically examine how both LPWAN and Wi-Fi air interfaces will work to deliver messages from mote to the base stations commencing this backhaul.

3.4.4 Data Communications with LoRa

With spreading factor selected, we can determine the number of symbols required for transmitting each MQTT command and its payload over the LoRa interface. This is determined using equation 2-1 as seen in Chapter 2, and we have put placed results in Table 3-8. Time on air is determined using the rate of 122 symbols per second, which results from our spreading factor of 10. The largest MQTT-format message sent is the CONNECT command at 64 bytes in length. This easily fits into a 256-byte payload, meaning that only a single LoRa frame will ever need to be sent between mote and gateway. These data rates are sufficient to meet requirements, and still falls under a single second for most commands while never exceeding 1.2 seconds for others. Table 3-9 estimates performance seen for communications between motes and base stations using LoRa.

Table 3-4: The exchange of MQTT-SN messages occurring when a mote establishes an MQTT-SN connection with the central system. Message length is derived from the MQTT-SN specification [83].

Direction	Message Type	Notes	Length (Bytes)
Mote – Broker	CONNECT	*CleanSession* is set to 1. *Will* is set to 1. *KeepAlive* is set to 120 seconds.	64
Broker – Mote	CONNACK		3
Broker-Mote	WILLTOPICREQ		2
Mote-Broker	WILLTOPIC	*Retain* is set to 0. Message retaining is not needed as the application server will be the sole subscriber. LWT topic is '*lost'*, which is represented by 4 bytes.	7
Broker-Mote	WILLMSGREQ		2
Mote-Broker	WILLMSG	LWT message is 16-byte unique mote ID.	18
Mote-Broker	REGISTER	Get unique ID for '*surcharge'* topic. This topic name is represented by 8 bytes.	14
Broker-Mote	REGACK		7
Mote-Broker	REGISTER	Get unique ID for '*heartbeat'* topic. This topic name is represented by 9 bytes.	15
Broker-Mote	REGACK		7
Total Bytes Transferred			**139**

Table 3-5: The exchange of MQTT-SN messages when a mote publishes a message of the *surcharge* topic.

Direction	Message Type	Notes	Length (Bytes)
Mote – Broker	PUBLISH	*Retain* is set to 0. *TopicID* is set to 0 to specify a standard topic ID.	26
Broker – Mote	PUBACK		7
Total Bytes Transferred			**33**

Table 3-6: The exchange of MQTT-SN messages when a mote publishes a message of the *heartbeat* topic.

Direction	Message Type	Notes	Length (Bytes)
Mote – Broker	PUBLISH	*Retain* is set to 0. *TopicID* is set to 0	24
Broker – Mote	PUBACK		7
Total Bytes Transferred			**31**

Table 3-7: The exchange of messages when a mote gracefully disconnects from the central system by re-entering sleep mode.

Direction	Message Type	Notes	Length (Bytes)
Mote – Broker	DISCONNECT	The *duration* field is not used as motes will not receive any messages.	2

Table 3-8: Symbols required and time on air for each MQTT-SN command using LoRa.

MQTT Command	**Packet Size**	**Symbols Required**	**Time on Air (Seconds)**
CONNECT	64	137	1.123
CONNACK	3	15	0.123
WILLTOPICREQ	2	13	0.107
WILLTOPIC	7	23	0.189
WILLMSGREQ	2	13	0.107
WILLMSG	18	45	0.369
REGISTER (*surcharge* topic)	14	37	0.303
REGISTER (*heartbeat* topic)	15	39	0.320
REGACK	7	23	0.189
PUBLISH (*surcharge* topic)	30	69	0.566
PUBLISH (*heartbeat* topic)	26	61	0.500
PUBACK	7	23	0.189
DISCONNECT	2	13	0.107

Table 3-9: Estimated performance of system LoRa communications.

Message Size (with MAC Overhead)	30 bytes
Packets per Communication	1
Symbols per Communication	69
Symbol Rate	122 symbols/second
Code Rate	4/8, or 0.5 (50%)
Time on Air	0.566 seconds
Data Rate	1.77 packets per second (53 bytes/sec)

3.5 Central System Overview

The *central system* encompasses a variety of linked servers and components separated from individual motes via the public Internet backhaul. This includes the *Application Server* (AS), LoRa *Network Servers* (NS) MQTT broker, and the MQTT-SN GW responsible for receiving messages from base stations and forwarding them as appropriate. Messages sent through both Wi-Fi and LoRa connections will initially enter the *central system* at the MQTT-SN GW, encapsulated in packets for the UDP/IP protocol stack. The MQTT-SN GW and broker must be fully compatible with the UDP/IP protocol stack, and consequentially able to extract MQTT PDUs from its payload.

Using a wireless platform is advisable, and resultingly backhaul networks will be implemented using either GSM or LTE. Both GSM and LTE transceivers will be provided at base stations, and which is used will depend on availability at the deployment area. As both LTE and GSM communications are relatively expensive and have data quotas, transmission from base station to central server should be minimised. While the system aims to minimise cost, the much lower number of base stations compared to motes makes this increased cost acceptable.

3.6 Application Server

If the MQTT broker acts as the system's spinal cord carrying messages between nodes and components, the *Application Server* can be imagined as the brain carrying out more intelligent processing. Readings from sensors are simply *data* with no meaning, and the application server uses these readings to synthesise meaningful *information* that is passed to external parties as *knowledge*.

Earlier we outlined the high-level basic requirements of the application server;

1. Differentiating between an actual blockage and a 'false alarm'.
2. Classifying blockages as either full or partial.
3. Determining whether blockages are located at the property connection or in the main itself.
4. Locating the property connection, or length of main between two property connections, where the blockage is occurring.

These requirements outline the *information* that needs to be produced from mote data. This information must be passed to the appropriate external systems to generate *knowledge,* alongside being stored in a non-volatile data source for future analysis and use by external systems.

As our system's data flow is developed around the MQTT protocol, the application server will receive all incoming data from motes using MQTT. MQTT makes this process relatively simple, as the application server simply needs to subscribe to the *surcharge* and *heartbeat* topics published by motes. In addition, future versions of the system can support remote configuration with the application server publishing topics that motes subscribe to. While mote messages will originally arrive in the MQTT-SN format, they will be translated to standard MQTT by the MQTT-SN GW before ever reaching the broker.

Considering the above, we can develop a simple process flow for the application server to adhere to when receiving *surcharge* data. The process flow is shown in Figure 3-10 and each component described in the following section. As the visual representation in Figure 3-10 makes clear, this follows a *pipe-and-filter* architecture.

3.6.2 Receive MQTT Messages

The first component is responsible for receiving MQTT messages from the subscribed *surcharge* topic, which will be facilitated through an MQTT driver or library. As a result, the raw payload data wrapped in an MQTT packet at the mote will be extracted. Upon being received and extracted from the MQTT payload, this value will be perceived by the server as a single 232-bit raw binary value. While this is an accurate portrayal of the how the data was transported, it is useless for higher-level processing. This binary value will be passed to the next step, where the original and discrete variables can be extracted.

3.6.3 Extract Application Data

This component will begin processing immediately after receiving a binary value from the first. Earlier, we demonstrated how each discrete data value from a mote was encoded into a single binary value for transportation using MQTT-SN. Once the data has finished transportation, this process can be reversed to decode the string into its original variables. Table 3-10 shows how the variables are structured in the original binary string.

Before beginning processing, basic integrity checks should be applied to the string. This will ensure that if a malformed or 'bad' value reaches the server, it will not waste valuable computation resources on wasted decoding attempts.

Alongside saving time, integrity checks can potentially prevent critical errors encountered when attempting to decode erroneous data. By doing this, we can contribute further to meeting requirements for minimising resource consumption and maximising uptime.

However, considering the requirement of minimising resource consumption, we must also ensure that performing these integrity checks does not consume more resources than attempting to decode malformed data. As a result, initial integrity checks will consist of;

- Ensuring the value is 232 bits long.
- Ensuring the trailing seven bits are equal to 10x0.
- Ensuring the 17th, 26th, and 35th bytes are equal to an ASCII space (10x32, or 2x0100000).

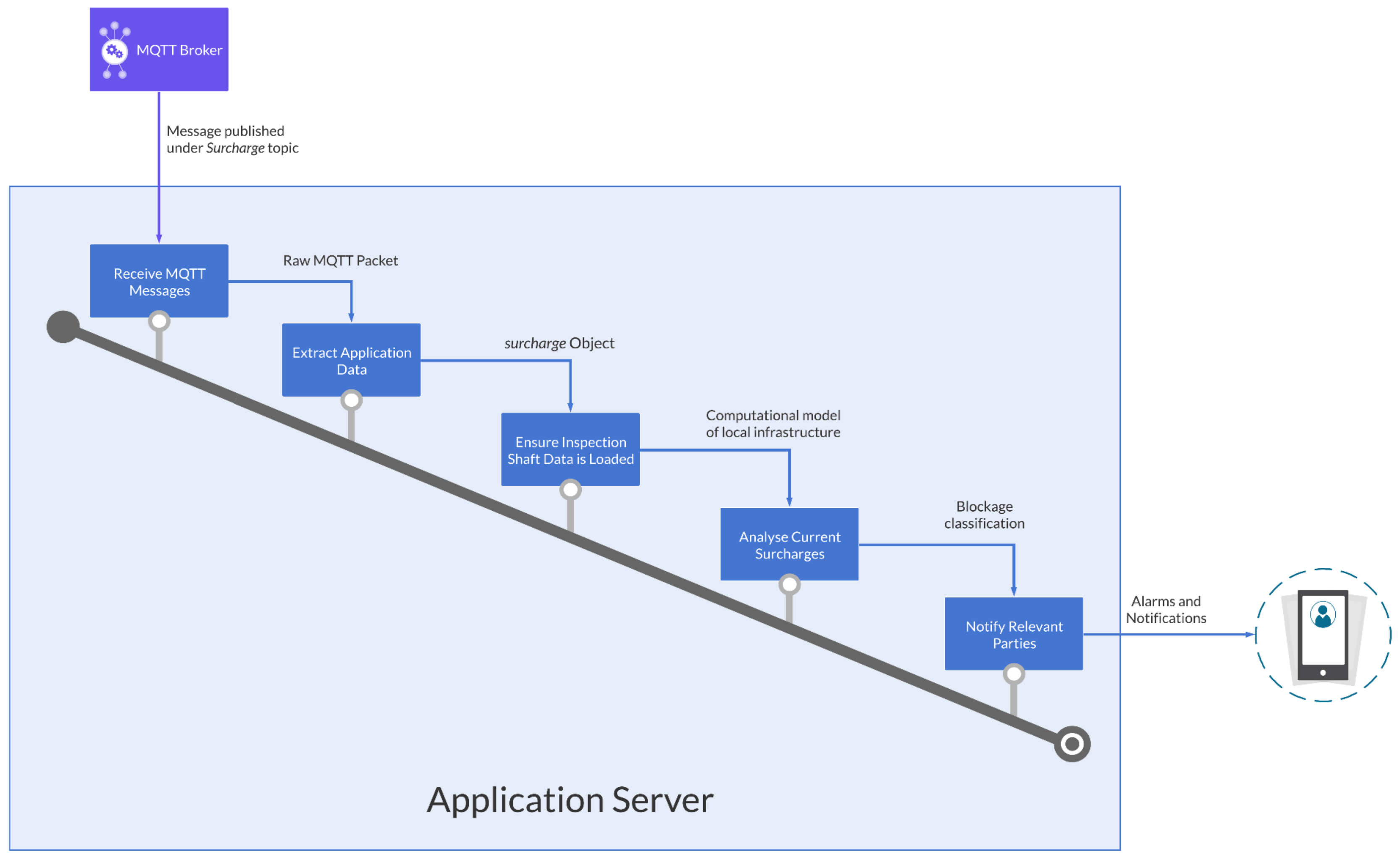


Figure 3-10: The pipeline for *surcharge* messages that arrive at the application server.

These checks use very little computational resources and can be performed very quickly, while preventing the most serious data malformations and potential critical errors. They should also be performed in order, with failing one check preventing all others from executing. As the checks are arranged in ascending order of complexity, this introduces further potential for conservation of resources.

If all checks are passed, each value can be extracted from the binary string and cast to the relevant data type. Client ID will be cast to a string, Timestamp to a Date/Time value, Battery Level to an integer, and Surcharge Status to a boolean.

These values can then be used to create a temporary object of the *SurchargeMessage* class shown in Figure 3-11, which will be sent to the next component. *SurchargeMessage* is itself an extension of the more generic *ClientMessage* class – *ClientMessage* represents any message sent from a mote and contains the minimum fields every message must contain regardless of type. In contrast, *SurchargeMessage* contains fields needed only when the message indicates surcharge. The previously mentioned temporary object forms both the output of this component and the input of the next.

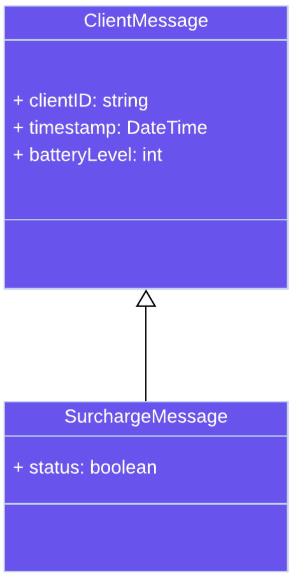


Figure 3-11: The ClientMessage and SurchargeMessage classes.

Table 3-10: Bitwise composition of application data binary string	
Data	**Bit Length**
Client ID	128
“ “ character	8
Timestamp	64
“ “ character	8
Battery Level	8
“ “ character	8
Surcharge Status	1
Trailing zeroes	7
Total	232

3.6.4 Ensure Inspection Shaft Data is Loaded

With the temporary *SurchargeMessage* object loaded into this component, the server can now begin to convert that *data* into *information* – or, use those data values to represent tangible attributes of a mote placed in an inspection shaft.

Records for each mote and its surrounding inspection shaft will be stored in a permanent SQL database. Each *surcharge* message arriving from a given mote will include that mote’s latest surcharge status (*surcharged* or *not surcharged*), alongside that mote’s unique ID. This ID will also be stored in the mote’s database record, and as it is unique to each mote, can be used to retrieve it with a standard database query.

Reading and potentially writing database records each time a surcharge message arrives could be highly inefficient, especially in partial blockages where surcharge status can fluctuate rapidly in a short amount of time. If a blockage occurs in the sewer main, it will also result in several motes communicating with the application server, which will all require records to be retrieved. Combine this with multiple potential fluctuations, and the amount of database interactions quickly becomes impractical.

After considering the application server and system requirements in combination with design principles, we have identified an intuitive solution that allows an accurate representation of surcharges and their involved sewer infrastructure while maintaining efficiency and conserving resources.

Mote records will only be retrieved from the database and converted to objects on an *as-needed* basis. Objects representing all retrieved motes will be stored in a dictionary and indexed by their unique ID – using a dictionary allows the object to be accessed in constant O(1) time if the key is known. Each surcharge message's client ID can be used to check if the mote has been retrieved by searching the dictionary and retrieve its object if so. Both of these operations can be completed in O(1) time.

If a surcharge message arrives for a given mote, its representing object will be placed in the aforementioned dictionary which we will refer to as the *active motes dictionary*. Whether the message represents a true blockage or false alarm, the mote's object will be stored in this dictionary until a pre-defined time passes with no activity. Once this time passes for any given mote, it will be removed from the dictionary to preserve volatile memory.

Every mote's object contains the unique ID of its downstream neighbour, alongside a field for storing a link to that neighbour's object. When a surcharge message arrives for a mote not stored in the *active motes dictionary*, records will be retrieved for both the concerned mote and its downstream neighbour. These records will be used to create objects representing both motes, with the downstream neighbour ID populated for both. The actual surcharging mote will also have its link field populated, linking it to the downstream neighbour. The downstream neighbour's link field cannot be populated, as its own neighbour has not been retrieved. This creates a *linked-list* data structure for neighbouring motes. Following this retrieval and linking, both mote objects will be placed in the *active motes dictionary*.

By default, the link to neighbour object will be **null**, and is only populated if that mote receives a surcharge message. However, if a mote object's downstream neighbour ID is also **null**, the database has no records of downstream neighbours for that mote. Our system will assume that mote is the last downstream on its sewer main. If a mote is the last downstream on its sewer main, having no downstream neighbour record is the correct way to represent it.

If a surcharge message arrives for a mote currently present in the *active motes dictionary*, the matching mote object will be retrieved and updated to match the surcharge message. If that mote object has a downstream neighbour (the downstream neighbour ID is not null) but the link is empty, its downstream neighbour will be retrieved from the database and linked. The retrieved downstream neighbour, as always, will store the ID of its own neighbour.

This process is illustrated in Figures 3-12 and 3-13 and explained through the pseudocode in *Pseudocode 3-3*.

One gap is still present in this model when considering the analyses to be performed – the *time* domain. Rapid fluctuations in surcharge status are used to differentiate *partial* blockages from full blockages, and the length of a single surcharge often distinguishes a genuine blockage from a false alarm. Considering the above, utilisation of the time domain is essential when developing our computational model.

Each surcharge state change that occurs for a mote should be given its own object, storing both the current state and time of change. When a surcharge message arrives at the application server, the extracted surcharge status and timestamp will be used to create one of these objects. Once the relevant mote has been loaded into volatile memory and linked with its upstream and downstream neighbours, this will be added to that mote's object.

Each of these *state change* objects should be stored in a *stack* data structure for each mode, with the stack's LIFO processing allowing processing of state changes from latest to earliest. Using a stack also ensures that the current value for that mote can be retrieved instantly.

Following this section, we have a comprehensive and efficient model for representing a sewer network in the application server's active memory. With this model established, we can now discuss how it is used to determine surcharge cause and differentiate types of blockages from false alarms.

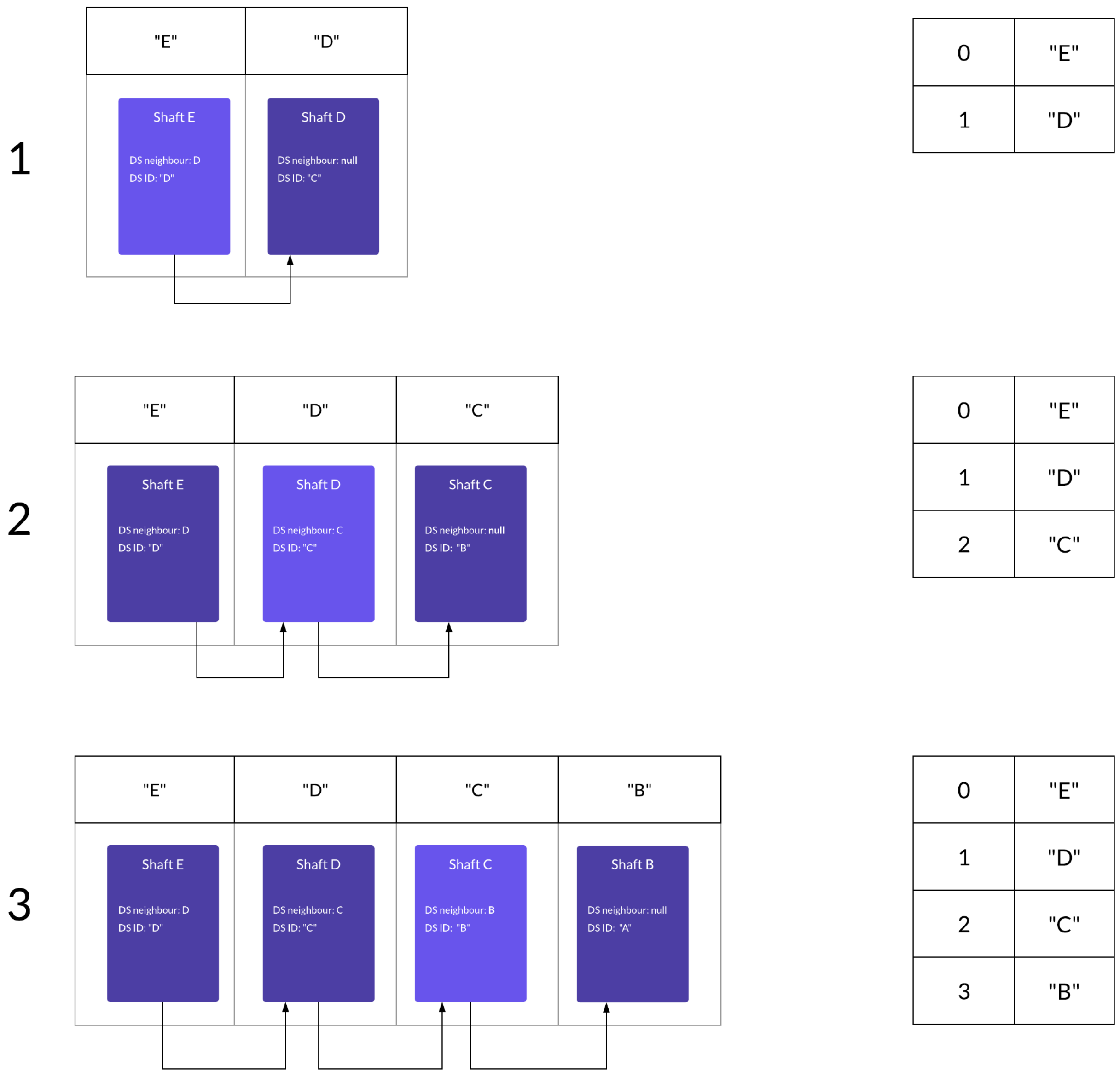


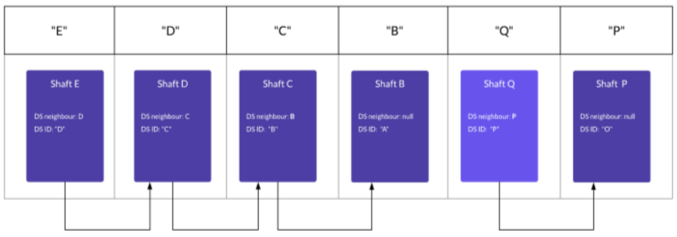
Figure 3-12: A demonstration of how mote records are dynamically retrieved. All mote records are stored in a dictionary data structure object keyed by their unique ID. The rightmost figure lists these dictionary keys (which map to the relevant mote records) in their indexed order – also the order they were retrieved.

In (1), a surcharge message arrives for *Shaft E*. Both Shaft E and Shaft D's mote records are retrieved, and a field in shaft E is populated with a link to shaft D. Like all mote records, Shaft D contains the ID of its downstream neighbour, even if no reference exists.

In (2), a surcharge message arrives for Shaft D. Shaft D's record has already been retrieved, however its downstream neighbour has not. The ID of its downstream neighbour is used to query the database, and the matching mote record (Shaft C) is retrieved. Shaft D is then updated to include a link to Shaft C's object.

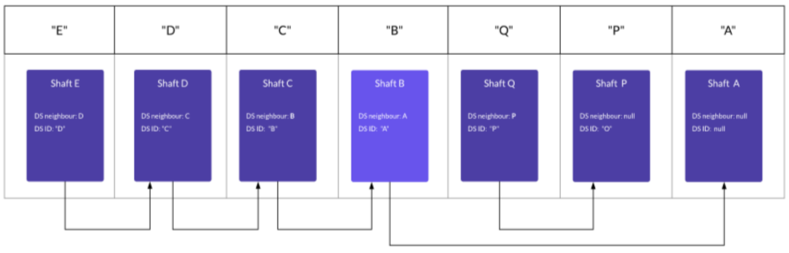
Finally, in (3), a surcharge message arrives for Shaft C. The above process repeats, downloading a record for Shaft B and linking it to Shaft C's record.

4



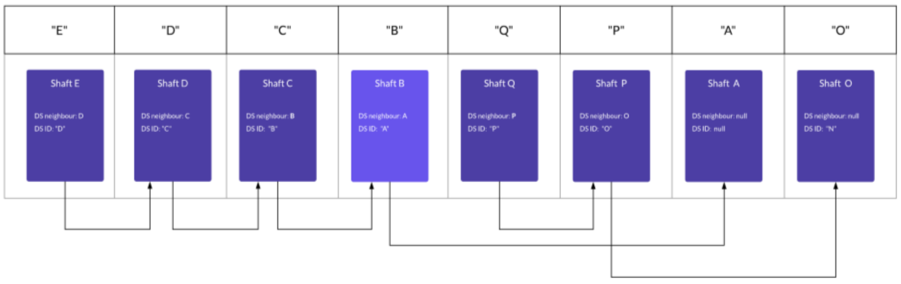
0	"E"
1	"D"
2	"C"
3	"B"
4	"Q"
5	"P"

5



0	"E"
1	"D"
2	"C"
3	"B"
4	"Q"
5	"P"
6	"A"

6



0	"E"
1	"D"
2	"C"
3	"B"
4	"Q"
5	"P"
6	"A"
7	"O"

Figure 3-13 : A continuation of Figure 3-12, showing further scenarios where surcharge messages arrive for different motes.

In (4), a surcharge message arrives for Shaft Q. As this is not downstream to any previously surcharging motes, both it and its own downstream mote records will be downloaded. As always, Shaft Q will contain a link to Shaft P. These will be placed after the previously retrieved motes at the next two indices.

In (5), a surcharge message arrives for Shaft B. Its upstream neighbour (Shaft C already surcharged in Figure 3-16, so Shaft B's record is also stored. Like always, its downstream neighbour (Shaft A) will be downloaded and linked to Shaft B. Both the downstream mote link and ID are **null** in Shaft A, meaning that it is the end of the main and has no downstream neighbours. Despite being linked to Shaft B, Shaft A's record will be stored two places removed from it in the dictionary of all retrieved mote records.

In (6), Shaft P surcharges and its downstream neighbour (Shaft O) is retrieved with a link added to Shaft P. This furthers the example shown in (5), being placed after Shaft A in the dictionary despite the direct link.

Pseudocode 3-3: Process used for loading inspection shaft data.

```
/* param theSurcharge is SurchargeMessage object passed from ‘Extract
   Application Data’ component */

function load_inspection_shaft_data(SurchargeMessage theSurcharge)
{
   Mote surchargingMote;

   if(!activeMotesDictionary.hasKey(theSurcharge.ID))
   {
      surchargingMote = retrieve_from_database(theSurcharge.ID);

      downstreamMote = retrieve_from_database(surchargingMote.downstreamID);

      surchargingMote.downstream = downstreamMote;

      //Assume this method takes a key and value to add to dictionary
      activeMotesDictionary.add(surchargingMote.ID, surchargingMote);
   }
   else
   {
      surchargingMote = activeMotesDictionary[theSurcharge.ID];

      /* If the surcharging mote has a downstream neighbour that is not yet
         loaded - mote’s upstream neighbour has surcharged in the past. If
         downstream ID is null, the mote has no neighbour. */
      if(surchargingMote.downstreamID != null && surchargingMote.downstream ==
         null)
      {
         neighbour = retrieve_from_database(surchargingMote.downstreamID);

         surchargingMote.downstream = neighbour;
         activeMotesDictionary.add(neighbour.ID, neighbour);

      }
   }

   return surchargingMote;
}
```

3.6.5 Analyse Current Surcharges

With a model of all actively surcharging motes and their neighbours loaded into application memory, it is possible to utilise a single mote's state change message to determine the exact cause of that state change. We aim to determine if the simple sensor reading of whether effluent is above a certain level indicates;

- A partial blockage
- A full blockage
- A blockage located at the inspection shaft
- A blockage located along the sewer main
- A false alarm (e.g., daily mass ejection)

Every surcharge message will be individually analysed in three steps; i – *Alarm Veracity,* ii – *Spatial Analysis*, and iii – *Time Analysis*. If conditions are met for each analysis, its results will be passed to the next stage – another example of *pipe and filter* architecture. Each of these stages are briefly discussed below, and graphically illustrated in Figures 3-14, 3-15 and 3-16. Following this *Pseudocode 3-4, 3-5, and 3-6* implements each stage in pseudocode to provide a detailed description.

When entering this process, every *SurchargeMessage* object is converted to or merged with an object of the *Event* class. *Event* class represents any occurrence in sewer infrastructure causing surcharges across one or more assets – this includes false alarms and all types of blockage. An event contains all surcharges resulting from the same root cause, and therefore motes can only belong to one currently occurring event at any given time. After some time without an involved mote experiencing surcharge, events will become *inactive*, meaning the root cause (blockage or false alarm) is no longer causing surcharges. This time is configurable by system administrators and is referred to as the *inactive timer*.

All motes involved in an event object will be located downstream from one another – the only way for a blockage to span multiple motes is a sewer main blockage, and main blockages surcharge neighbouring inspection shafts. Each *Event* object contains a link to a single *Mote* object, and all other surcharging motes can be accessed from the linked mote's own link to its downstream neighbours. This is effective as all surcharging motes on the same main form a *linked list*, as detailed in Section 3.6.4. Each *Mote* object has a *Stack* of objects belonging to a class named *StateChange,* that represent the surcharge state changes occurring for that mote in the current event. *StateChange* objects contain both the current state (as of that change), and the time of state change. This can be seen in the application server's *class diagram* at Figure 3-17.

3.6.5.1 Alarm Veracity

Alarm Veracity accepts the raw sensor reading from the source mote as a *SurchargeMessage* object and determines whether it represents an actual blockage or is a false alarm. False alarms, like partial blockages, usually involve rapid fluctuations between *surcharged* and *not surcharged*. However, unlike partial blockages, false alarms will have much quicker fluctuations and 'settle' after a small number. This difference in fluctuation number and speed will be used to differentiate the two.

Before any other step is taken, the *active motes dictionary* is checked to determine whether that mote or its downstream neighbour have had recent surcharge activity. If the mote and neighbour have had no recent activity, an *Event* object will be created for that mote and surcharge, and the *false alarm* timer will begin counting down. If this elapses and no other surcharges are received for that mote or its neighbour, the Event object is passed to *Spatial Analysis.*

If an additional surcharge arrives while the false alarm timer is counting down, the Event object will be added to a list of current *false alarms*. If more than two surcharge messages arrive for that mote or downstream neighbour while the *Event* object is classified as a false alarm, it could in fact be an unusual partial blockage. To determine whether this is the case, it will be released from the *false alarms* list and passed to Spatial Analysis.

If the mote or its downstream neighbour have had recent surcharge activity and there are either no false alarms or a greater number than 2 false alarms stored (as previously mentioned), that surcharge will be used to create a *StateChange* object and added to the correct *Mote* object as stored in the *active motes dictionary*. That mote will already be referenced in an *Event* object's linked list of affected motes. As all objects in our application server's code are treated as *references*, any update to the object will affect its presence everywhere.

If the stack is not empty, the arriving state change is a continuation of the event which caused the previous ones. Differentiating these is important; a surcharge that occurs for a long period of time is very different to one which quickly disappears. Adding new state changes to an existing event also assists with analysing that event and determining both its cause and fluctuation rate.

3.6.5.2 *Spatial Classification*

As previously discussed, *spatial classification* determines whether a blockage is occurring at a property connection or main and is conceptually rather simple. If the blockage's Event mote contains more than one mote, it will be a *main* surcharge. All events will have a single Mote value, however if that Mote contains a reference to another Mote downstream, it will form a *linked list*. These motes will chain together through the *downstream* reference until the furthest downstream or end of the main is reached.

Every surcharge message will be added to a retrieved mote's stack of state changes by looking that mote up in the *active motes dictionary*. As both the dictionary and Event linked list contain references to the same object (and therefore memory location), updating one will update the other. Using the dictionary not only results in less complicated programming but allows updates to be performed in real time.

As main surcharges progress, surcharges will arrive for motes that do not yet belong to an event – however, their downstream neighbour does. The surcharges in these motes will obviously belong to the same event as their downstream neighbours, as they will have the same root cause. The downstream neighbour will be assigned to the newly surcharging mote's own *downstream* reference, placing it at the front of the linked list. Following this, the Event's *mote* reference will be overwritten with the newly surcharging mote. This replaces the existing linked list 'starting point' with a new one that includes all surcharging motes.

The *Event* class' *mote* reference will be made private so any changes must be facilitated through a *mutator* method. Each time this mutator method is called to replace the linked-list, additional code in that mutator's body will be executed. This code will measure the length of the linked-list and classify the event accordingly. If an event has a *mote* value with no downstream reference, it is classified as a '*property connection or main'* blockage. Following this classification, a timer named the *property_connection_timer* will begin to count down. If this timer elapses and no further motes are added to the event, it will be fully classified as a *property connection* blockage.

A Property Connection or Main blockage appears the same as a standard property connection blockage, however, is separately classified for the benefit of field staff. This ensures that if field staff attend the site and do not find the blockage in the property connection, they can assume it is occurring in the main between that connection and its downstream neighbour. By implementing this distinction, we have accounted for the fact that all main blockages are initially indistinguishable from property connection blockages. This potentially also allows for main blockages to be resolved while still only affecting a single property.

If the *mote* value for an event is overriden with a mote that has a *downstream* value pointing to another mote, the mutator will reclassify it as a *main* blockage if it has surcharged since the event's commencement.

Following this, the Event object can use its own *mote* value to provide an estimated location for the blockage. If the Event is a *main* blockage, it will assume the blockage is located between its furthest downstream surcharging mote and that mote's own downstream neighbour. Conversely, if the Event is a *property connection* blockage, it will assume the blockage is occurring at that property connection. Early in the Event's inception, it will be classified as a *property connection or main* blockage, allowing for both possibilities. In this situation both of the above potential locations will be raised, with the incorrect one being removed on further classification.

With spatial classification complete and the blockage located, the Event object will be passed into the *Time Classification* component.

3.6.5.3 Temporal Classification

In contrast to spatial classification which was concerned with the number of motes linked to an Event object, *temporal* classification will examine each of these motes' surcharge event histories. While it is easy to determine the 'latest' surcharge in a property connection blockage involving a single mote, it is more difficult in a main blockage that occurs across several motes. To keep things relatively simple, our system will compare the most recent surcharge for all involved motes by *popping* their individual stacks. Consequentially, the 'latest' surcharge for a main blockage will be the most recent from all involved motes.

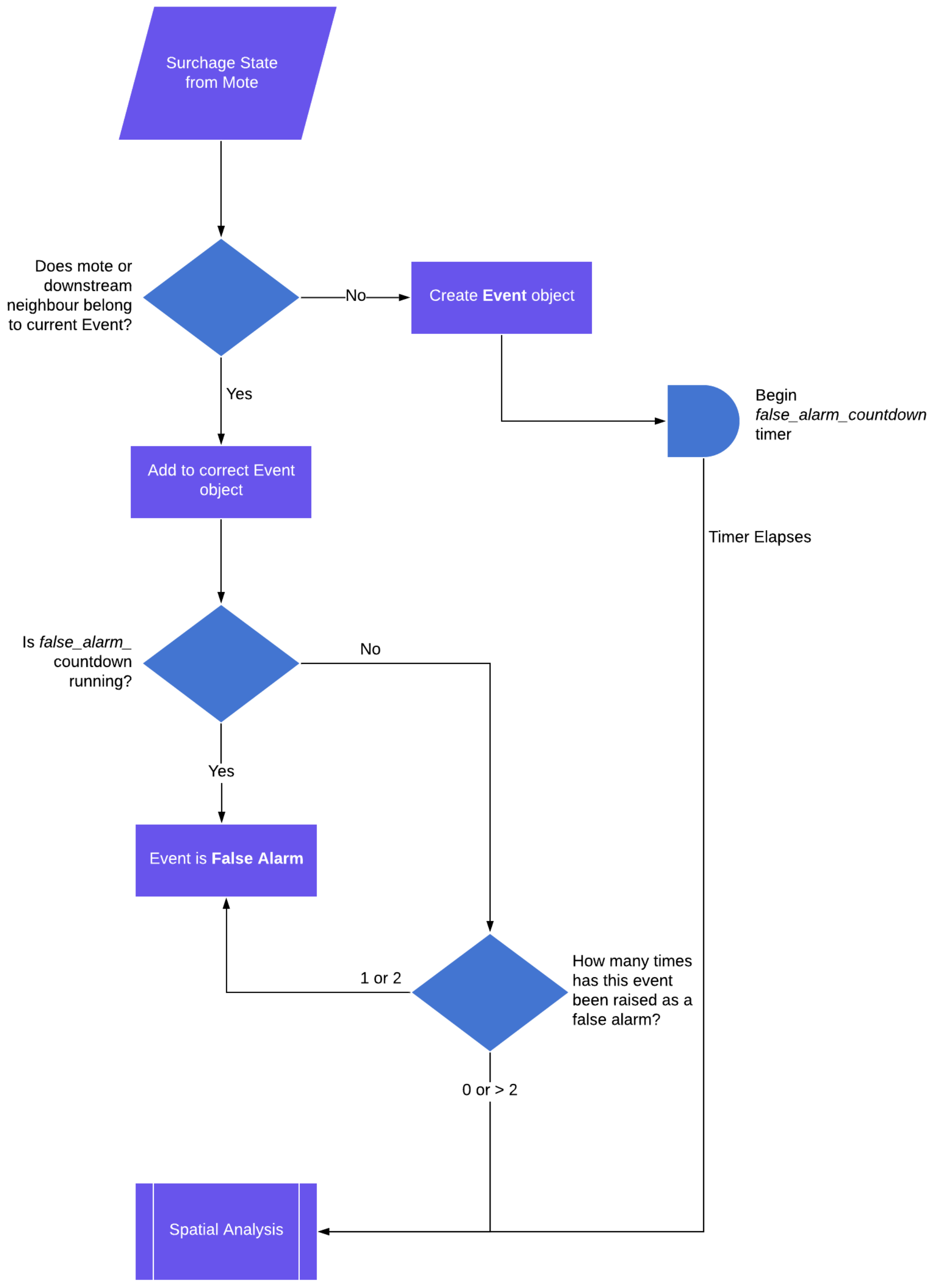
Every time a surcharge status of 0 arrives for a given event, our system will check whether all involved motes have statuses of 0. If this is the case, a timer named *inactive_countdown* will begin counting down. If this timer elapses with no further state changes arriving for the Event, it will be considered *resolved* and removed from the storage of active events. However, if a further state change arrives, this indicates a new fluctuation in a partial blockage. The blockage has therefore been classified as *partial*.

If a surcharge status of 1 arrives, the stacks of all involved Motes will be checked. If any Mote's stack has previously cycled between both 1 and 0, the blockage is causing fluctuating effluent levels and therefore cycling between *surcharged* and *not surcharged*. Our system assumes this indicates a partial blockage. Conversely, if the Event's latest surcharge status is 1 and no motes have previously cycled between 1 and 0, the application server will assume this is a *full* blockage as no fluctuations are occurring.

Whenever a blockage is classified as partial and all involved motes have surcharge statuses of 1, a timer named *partial_to_full_countdown* will begin counting down. If this time elapses with no further state changes for motes involved at the time the countdown begun, the blockage will be reclassified as *full*.

This discussion shows the need for several background *timers* counting down for successful classification of surcharge events. As a consequence, the platform and language chosen for developing the application server must be highly suited to asynchronous and parallel programming.

Each time a blockage is classified or reclassified, a clone of the relevant *Event* object will be passed into the main application server pipeline's final component – *Notify Relevant Parties.*



Surchage State from Mote
Does mote or downstream neighbour belong to current Event?
No
Create **Event** object
Begin *false_alarm_countdown* timer
Yes
Add to correct Event object
Timer Elapses
Is *false_alarm_* countdown running?
No
Yes
Event is **False Alarm**
1 or 2
How many times has this event been raised as a false alarm?
0 or > 2
Spatial Analysis

Figure 3-14: An illustrated diagram of the *Alarm Veracity* algorithm. The *falseAlarmCount* value of the *Event* object stores how many times an Event has been determined as a ‘false alarm’ by the system.

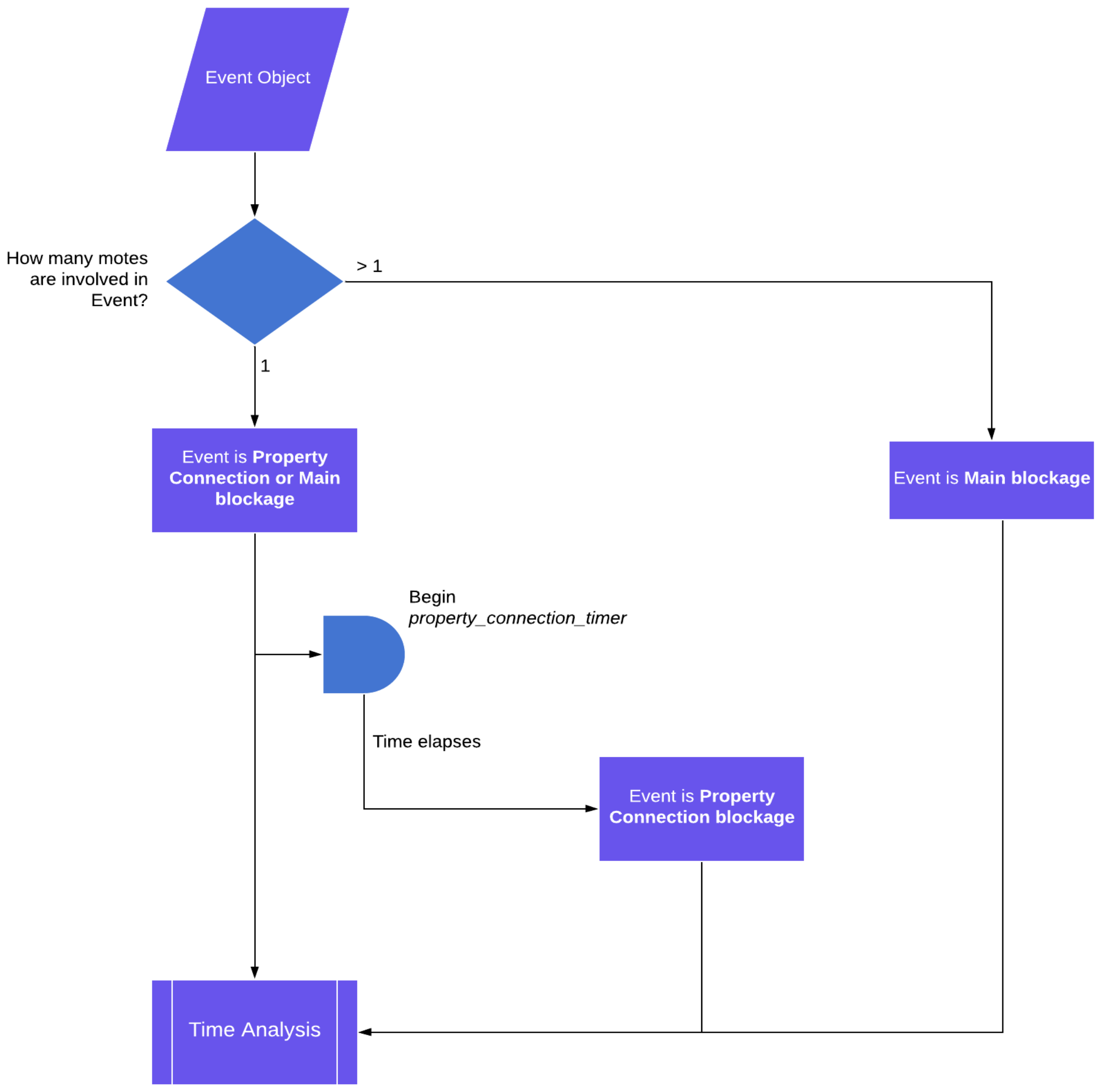
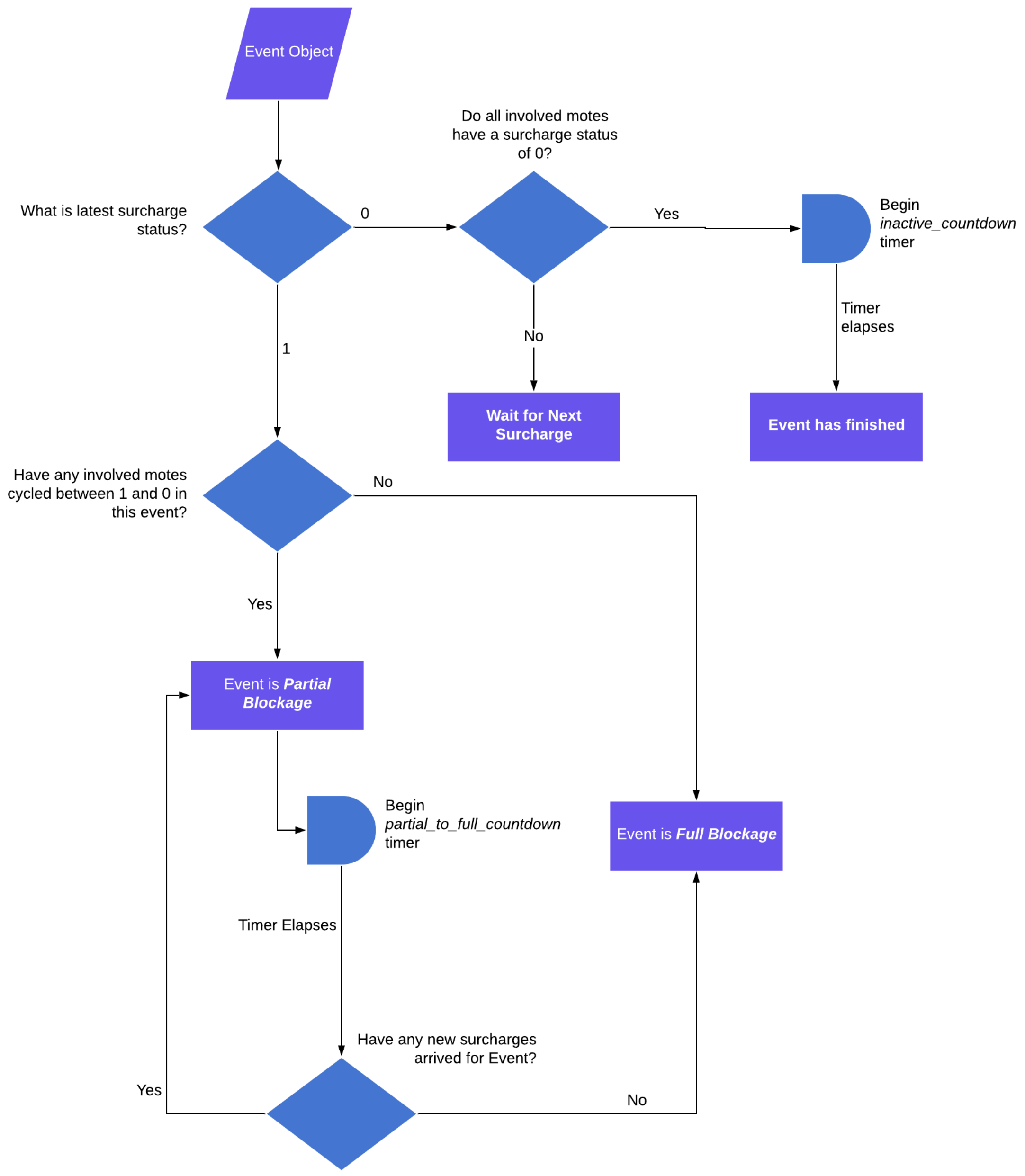

Event Object
How many motes are involved in Event?
> 1
1
Event is Property Connection or Main blockage
Event is Main blockage
Begin property_connection_timer
Time elapses
Event is Property Connection blockage
Time Analysis

Figure 3-15: An illustrated diagram of the *Spatial Classification* algorithm. An Event with one mote surcharging will initially be classified as a *Property Connection or Main* blockage, and this will be passed into Time Classification. If the *property_connection_timer* elapses, it will be reclassified as a *Property Connection* blockage. The reclassified Event will be passed into the *Time Analysis* again to determine if things have changed since its last time analysis.



Event Object
What is latest surcharge status?
0
Do all involved motes have a surcharge status of 0?
Yes
Begin *inactive_countdown* timer
Timer elapses
Event has finished
No
Wait for Next Surcharge
1
Have any involved motes cycled between 1 and 0 in this event?
No
Yes
Event is ***Partial Blockage***
Begin *partial_to_full_countdown* timer
Event is ***Full Blockage***
Timer Elapses
Have any new surcharges arrived for Event?
Yes
No

Figure 3-16: An illustrated diagram of the *Temporal Classification* algorithm. Note that it is assumed the *partial_to_full_countdown* and *inactive_countdown* timers will be cancelled whenever a new surcharge occurs for a mote in the same event.

Pseudocode 3-4: Alarm Veracity process for application server – See figure 3-14

```
// surchargingMote is Mote object returned from function in Pseudocode 3-3

// Check if surchargingMote belongs to existing event.
Mote concernedEvent = null;

foreach(Event next in currentEvents)
{
   /* Assume search() recursively searches linked list of Motes for matching.
      (Match = mote where ID field matches the search string). Will return
       matching mote if found, and null if no match is found.

      First search for mote itself, then for downstream neighbour.
   */
   Mote directHit = next.involvedMotes.search(surchargingMote.ID);

   if(directHit != null)
   {
      concernedEvent = next;
      break;
   }
   else
   {
      Mote neighbourHit =
next.involvedMotes.search(surchargingMote.downstreamID);

       if(neighbourHit != null)
     {
        concernedEvent = next;
        break;
     }
   }

}

//If concernedEvent has not been found, no current Event exists. Create one.
if(concernedEvent == null)
{
   Event theEvent = new Event(surchargingMote);
   currentEvents.add(theEvent);
   theEvent.begin_false_alarm_countdown();
}
   if(concernedEvent.false_alarm_countdown_running)
   {
      concernedEvent.falseAlarmFlags += 1;
   }
   //concernedEvent false alarm countdown is not running
   else
   {
      if(concernedEvent.falseAlarmFlags.length == 1 ||
         concernedEvent.falseAlarmFlags.length == 2)
      {
         concernedEvent.falseAlarmFlags +=1;
      }
      else
      {
         concernedEvent.falseAlarmFlags = 0;
         //Assume this begins spatial analysis with event.
         begin_spatial_analysis(concernedEvent);
      }
   }
```

Pseudocode 3-5: Spatial Analysis algorithm utilised by application server – See figure 3-15

```
// concernedEvent is Event object passed to this stage in Pseudocode 3-4

// Assume getLength() recursively gets the length of a linked list.
if (concernedEvent.involvedMotes.getLength() > 1)
{
   concernedEvent.spatialClassification = “MAIN”;

   //Assume this begins time analysis for event
   begin_time_analysis(concernedEvent);
}
else
{
   concernedEvent.spatialClassification = “MAIN OR PROPERTY CONNECTION”;
   begin_time_analysis(concernedEvent);
}

/* Assume code in then() executes when asynchronous wait is over. This is based
   on ‘Promsies’ in asynchronous programming. */
asynchronous_wait(property_connection_timer).then
(
   if(concernedEvent.involvedMotes.getLength() > 1)
   {
      concernedEvent.spatialClassification = “MAIN”;
      begin_time_analysis(concernedEvent);
   }
);
```

Pseudocode 3-6: Time Analysis algorithm utilised by application server – See figure 3-16

```
// concernedEvent is Event object passed to this stage in Pseudocode 3-5

//Get latest state change for Event
StateChange latest = null;

// Assume toArray() recursively converts linked list to array
foreach(Mote next in involvedList.involvedMotes.toArray())
{
   StateChange nextChange = next.stateChanges.pop();
   if(latest == null || (nextChange.time > latest.time))
   {
      latest = nextChange;
   }
}

if(latest.status == 0)
{
    /* Use check_all_inactive (Detailed below) to check if all Motes involved in
       this Event are currently not experiencing surcharge. This will return
       true if all Motes are inactive, and false if vice versa. */
    if(check_all_inactive(concernedEvent))
    {
       asynchronous_wait(inactive_timer).then
       (
         if (check_all_inactive(concernedEvent))
            concernedEvent.resolved = true;
       );
    }
}
```

```
// Latest surcharge status is 1
else
{
   /* Use check_previous_fluctuations to determine if any Motes in this Event
      have previously cycled between 0 and 1. This is done by the
      check_previous_fluctuations method detailed below.
   */

   if(check_previous_fluctuations(concernedEvent))
   {

      changesBeforeWait = get_state_change_count(concernedEvent);
      event.timeClassification = “PARTIAL”;

      asynchronous_wait(partial_to_full_timer).then
       (
         /* See if event has had any new state changes arrive. If so, the
            count performed now will be different to the one performed
            earlier. */
         changesAfterWait = get_state_change_count(concernedEvent);

         if (changesBeforeWait == changesAfterWait)
            event.timeClassification = “FULL”;
       );
   }
   else
   {
      event.timeClassification = “FULL”;
   }
}
```

```
/* Determine if any Motes in this Event (toCheck) have previously cycled between
   0 and 1. If a Mote’s stateChanges stack has a length > 2, this will be the
   case. */
function check_previous_fluctuations(Event toCheck)
{
   foreach(Mote next in toCheck.involvedMotes.toArray())
   {
      if(next.stateChanges.length > 2)
      return true;
   }

   return false;
}

// Determine if all Motes in this event (toCheck) are currently not surcharged.
function check_all_inactive(Event toCheck)
{
   foreach(Mote next in toCheck.involvedMotes.toArray())
   {
      if(next.stateChanges.pop().status == 1)
         return false;
   }
   return true;
}

// Gets the total number of state changes for all Motes in an Event
function get_state_change_count(Event toCount)
{
   var toReturn = 0;
   foreach(Mote next in toCheck.involvedMotes.toArray())
   {
      toReturn += next.stateChanges.length;
   }
   return toReturn;
}
```

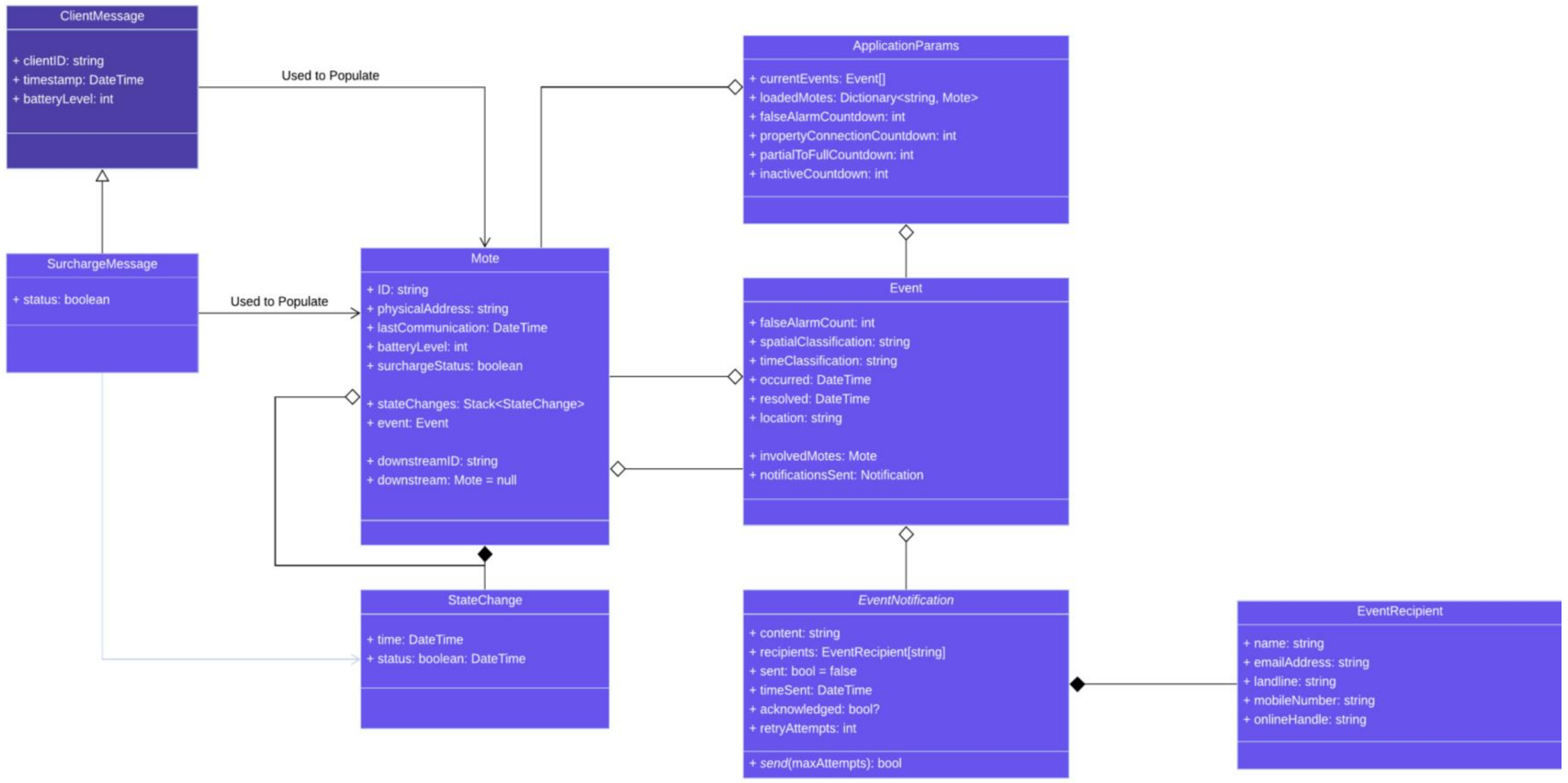


Figure 3-17: A class diagram for the central system’s application server. The *Application* class is used to store data structures accessible to all areas of the application, alongside configuration values set by user such as timer length. Considering this, it can be thought of as storage for global variables.

Note that this is unlikely to completely match the final set of classes, namespaces etc included in our solution – this is simply a theoretical design. Our implementation presented in the next chapter is based on this design.

3.6.6 Notify Relevant Parties

Whenever an Event is classified or reclassified as a blockage, it will be passed into this component which is responsible for ensuring required external parties can be made aware of the Event. Our system will automatically perform the minimum required functionality by sending an email message to a list of addresses stored in an external configuration file. However, solely sending emails is not sufficient for organisations (such as the majority of utility providers) that do not have staff constantly monitoring emails. In response, we have developed a mechanism for ‘connecting’ the application server to other systems that provide more sophisticated communications such as SMS and telephony.

The application server will expose a RESTful API that allows all current alarms to be retrieved by external systems as a list of JSON objects. Organisations can easily write small scripts or use specialised software to extract this list, convert it to the relevant format, and pass it to an API allowing more sophisticated communications. Many APIs exist for SMS, telephony, VoIP, and other messaging platforms. As most modern APIs utilise JSON, conversion should be relatively simple and resource efficient. The model discussed is illustrated below.

3.6.6 Analysing Heartbeats

Compared to state change messages, heartbeat messages are simple to process and react to. Heartbeats arrive at the application server through an MQTT message, and battery level is compared with a pre-set threshold. If battery level is below this threshold, an email will be sent to relevant staff members. Regardless of whether this threshold is met, a record of that heartbeat will be written to the database. If this database write occurs, it will be performed in parallel with the email notification.

A small program will execute outside the main application server software on an hourly basis, scanning each mote’s database records for when the last heartbeat was received. If this exceeds a certain threshold and heartbeats have not been received for an unacceptable time, an email will be sent to relevant parties. An additional record will be updated with the time this email was sent, and future scans will re-send the email if another day has passed.

3.6.7 Database Integration

While we have touted the benefits of our data structure solution located in application memory, it is important that permanent database storage is not neglected. Databases allow information to be retained, retrieved for future analysis, and harvested by stakeholders for a wide variety of business cases. Most importantly, building a valid model of inspection shafts and motes requires a permanent storage of information to be based on. Creating a valid database mechanism requires defining -

- The schema (logical structure) of how data is stored.
- When information will be written to the database.

Information must be periodically written to the database for obvious reasons, however as always, a balance must be struck. Excessive database writes will impact system performance and consume excessive resources, while too few database writes could result in data loss and untimely data. If the system experiences interruption while processing surcharges, it should be possible to reload the   retrieve information from the database which is as recent and relevant as possible. Ideally, data retrieved from the database will detail currently occurring events and differentiate them from historic events.

Considering the above, records should be updated whenever the object representing them is significantly changed. Database writes passing updated object states should occur when;

- An event starts, is classified or reclassified, or ends.
- A new surcharge is detected.
- A mote's details are significantly updated.

Figure 3-18 shows a SQL relational schema for a database theoretically capable of meeting all requirements.

The second point may raise some contention, as every message from a mote will contain updated information such as battery life. In events that involve frequent messages (such as partial blockages), mote information is unlikely to significantly change and writing it to the database will prove a waste of resources. To combat this, mote information will only be updated by heartbeat messages, or surcharge messages which deliver changed values.

From a programming perspective, it would be inefficient to have a monitor routine constantly running in the background and checking for object changes. Instead, code for updating the database and evaluating necessity should be implemented into *mutator* methods for object values. *Alarms* and notifications should be prioritised and written to the database in parallel to being 'sent' out via email.

Finally, database operations should all be executed as *transactions* and continually re-attempt if an error is encountered. If the number of unsuccessful encounters exceeds a given value, an alarm should be sent to relevant parties.

The previously mentioned RESTful API should operate almost exclusively by read/write operations to this database, further increasing the need for regular updates and database writes. This is especially true when considering that external alarm systems will use this API (and therefore the database) for sending critical alarms to required parties.

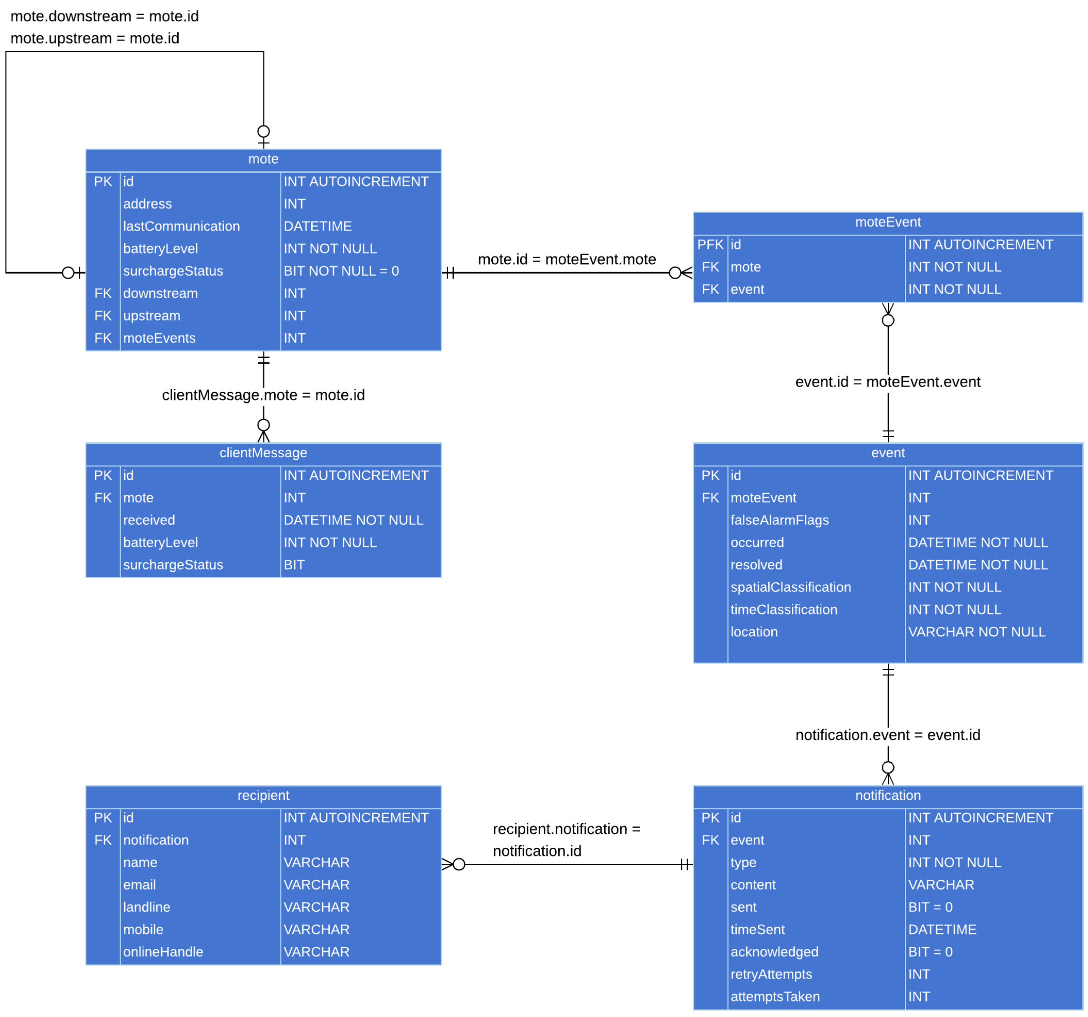


Figure 3-18: A relational database schema for the central system’s SQL database. The falseAlarmFlags field in event stores how many times that event was classified as a false alarm – whether correctly or not. Any notification sent to relevant users through any communications system should also be stored here as a row of the *notification* table. The *type* field of this table provides a numeric code that can be mapped to the type of communication used. Individual system administrators can specify different numbers for different communication platforms; however, we recommend that 1 is always used for email.

Finally, the overall architecture for the entire system can be succinctly described in Figure 3-19.

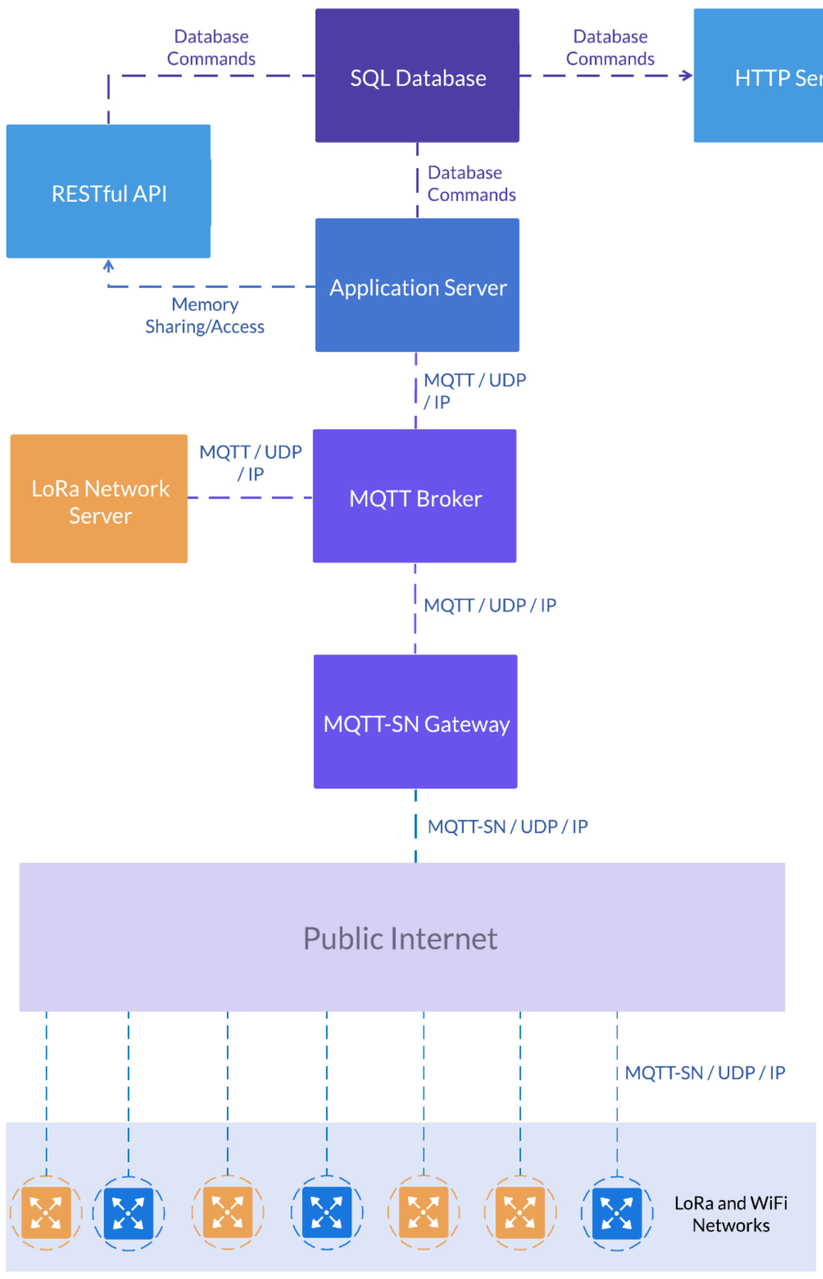


Figure 3-19: The overall architecture of our system.

3.7 Conclusion

As discussed at the beginning of this chapter, we intend to deliver a system capable of monitoring sewer blockages across a wide area while remaining practical and cost-effective. Successfully implementing this system requires a robust design to follow during development, providing us with answers for how to meet our research requirements. This chapter presented detailed designs for a sensor mote, central system, and network architecture.

Our mote design outlined how constrained hardware and software capable of detecting wastewater surcharges. We chose to utilise a float switch for physically sensing whether a surcharge is occurring and produced a theoretical model and set of requirements for a microcontroller capable of interfacing with it. This model was then accompanied with algorithms and designs for software that can process float switch data and send appropriate messages to the central system.

Following this, we designed a network facilitating communication between motes following the aforementioned design and a central system. The design allows both LPWAN and Wi-Fi connections to co-exist, implementing MQTT and MQTT-SN protocol communications. We also ensured that surcharge and heartbeat data can be sent through these protocols while still being sufficiently lightweight for LPWAN networks.

Finally, our central system design specifies how to receive incoming messages from motes, process these messages, differentiate actual blockages or false alarms, and classify blockages. This design was presented through a combination of traditional software engineering tools such as UML diagrams and database schemas, alongside visual and pseudocode representation of complex algorithms.

Collectively, these produced an overall design for a system capable of meeting our research requirements. However, successfully developing a design only proved our system is *theoretically* possible. In the following chapter we will develop a prototype derived from this design to verify it is capable of delivering a real, working system.

Chapter 4 Building and Evaluating Prototype

In the first two chapters we demonstrated the need for a system capable of detecting sewer blockages on a city-wide scale and found that no practical solution currently exists. To fulfil the goal of building an IoT-based blockage detection system, we presented a detailed design based on that goal in Chapter 3. Our design is capable of carrying out the much-needed sewer blockage system, and most importantly is practical for large-scale deployment in a variety of environmental and socioeconomic conditions.

However, even the most robust and detailed design requires a form of prototyping and practical evaluation. Things that appear effective or fulfil all requirements in a theoretical environment are often thwarted by real-world variables. Testing also reveals previously undetected issues that are not obvious during purely theoretical design and allows those issues to be resolved in future iterations or versions of design.

We have developed a prototype implementation of Chapter 3's design capable of demonstrating its practicality and showing that it can produce a working system. First, this delivers a mote device adhering to the design specified in Chapter 3. Follow this, we deliver an implementation of that chapter's design for a *central system*. While some functionality such as email alerts and a user interface have not been delivered, these are secondary to the main innovation and purpose of our system. With a mote and central system prototype developed, a practical demonstration of the design at all layers of architecture is possible.

In this chapter, we detail the prototype mote and central system produced, and provide instructions for their practical implementation. This allows future endeavours to adopt our work and implement their own version of the system on a commercial scale. To verify the prototype system, and therefore design, are operating as intended we have also performed some testing. The testing process shows the prototype system successfully reading surcharge events and classifying blockages, and both the process and results are outlined in this chapter.

4.1 Prototype Mote Development

To build the mote, we decided to use commercially available and easily configurable hardware components as many of these are designed for electronic prototyping. This assembly of pre-existing components will prove our design can be implemented effectively, allowing future implementations to apply our design to dedicated, manufactured components. This section will first discuss the selection and configuration of these components, before detailing the software developed to carry out the required functionality.

Only one mote was produced, as this will be sufficient to prove that motes can effectively send surcharge messages to the central system. Our intention when developing the prototypes is to prove that the design can be implemented, and one mote will be enough to do so. This also allowed us to work within the time and financial constraints of development, and within the scope of the degree.

4.1.1 Prototype Mote Hardware

After considering several single-board computers, microcontrollers, and electronic devices, we selected the *Raspberry Pi Zero* (RPZ) single board computer. RPZs are physically compact at only 65x30x5mm in size [110], extremely cost-effective with a price per unit averaging only $21 at the time of writing and include an integrated Wi-Fi transceiver. Unlike many other single-board computers, RPZs utilise a fully featured Linux operating system developers can interact with using a GUI or terminal. Most user-facing features will be disabled during testing to conserve power; however, these will prove invaluable during the development and testing process. To utilise the GUI, the RPZ provides a *mini-HDMI* port for more conventional displays and a proprietary PiTFT video connection for Raspberry Pi-specific touch screens [111]. A photograph of an RPZ without any connected wires or cables is shown in Figure 4-1.

In addition to the connections mentioned above, the RPZ also contains two micro USB ports. The port closest to the HDMI connection is used for connecting peripheral devices, while the other is reserved for use as a power supply. Providing power through the micro USB port is very easy, as USB to Micro USB cables are widely available alongside USB wall adapters. Making this even easier, many modern buildings now include USB ports in electrical outlets. Power can also be provided through the RPZ's GPIO pins, with the 3.3V and 5V *power* pins accepting electrical input – however, using the micro USB cable is much more likely to work correctly. Despite this, using GPIO pins is a good option when power supplies not compatible with USB are required.

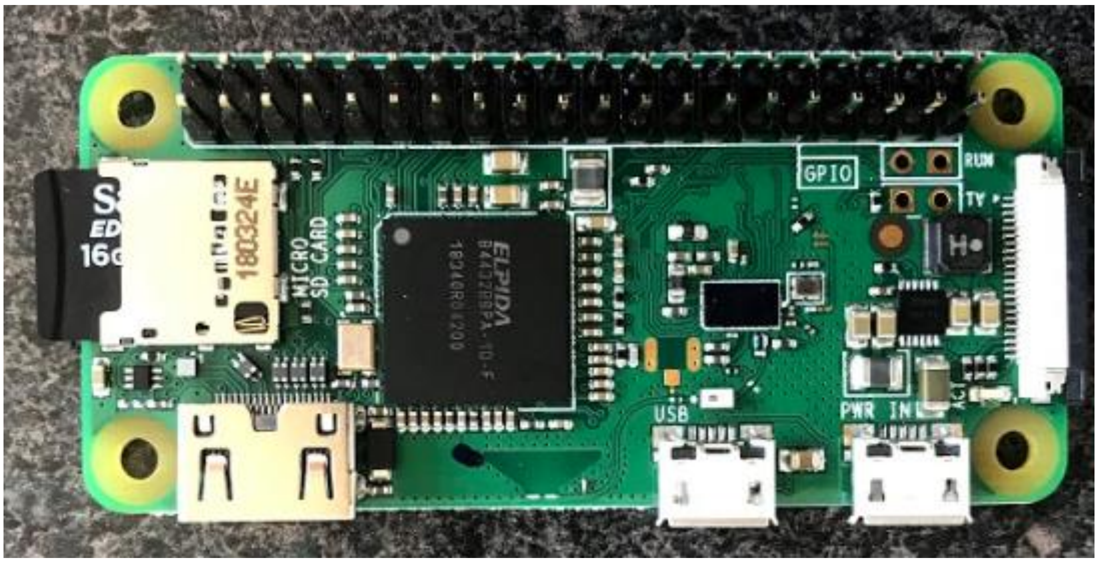


Figure 4-1: A photograph of a Raspberry Pi Zero (RPZ). This is the same RPZ used to build the mote for our prototype system.

Sensors and other devices connect to the RPZ using its GPIO interface. The RPZ exposes 40 pins, laid out in a grid 2 pins wide and 20 high. Not all of these pins are able to be used as GPIO interfaces, with some being reserved for power, grounding, and system functionality. Of the pins with GPIO capabilities, some of these will be unreachable by default as they are reserved for other protocols or applications. These pins can be utilised if absolutely necessary, however it is not recommended they be used. This is particularly relevant with the UART RX and TX pins, which are required by an enormous number of other devices.

Pins are accessed by the operating system and its applications using identifying numbers mapped to a physical pin. There are four numbering schemes available to the RPZ [90] – *Broadcom Pin Number* (BCM), *WiringPi, Physical* and *Rev 1 Pi.* Notably, the *Physical* scheme uses numbers identical to those printed on the board next to each GPIO pin, providing an exact replica of the physical pin number. *WiringPi* could be recommended for more performance-critical or advanced systems, as it provides a library using the C programming language [92].

We will utilise the *BCM* scheme for several reasons – perhaps most importantly, it is specified by *Broadcom,* who also developed the RPZ's internal CPU. Under most circumstances, it is advisable to utilise any specifications developed by the original hardware manufacturer. Table

4-1 lists each pin available on the RPZ in order of its physical number and provides some more information [91] [92]. Following this, Figure 4-2 provides an illustration of all GPIO pins, components, and other ports on the RPZ.

Serendipitously, the RPZ has built-in pull-up and pull-down resistors for every I/O pin, which can be activated or deactivated from the operating system and many software libraries [92].

As the RPZ integrates most of the components required by our system's motes into a single board, very few electrical connections are needed. The float switch used to detect surcharges will be connected to both a 3.3V output and GPIO pin, with one providing power to the switch while the other provides a digital signal to the mote's microcontroller. Power will constantly flow from the 3.3V pin, however if the float switch is open it will not reach the GPIO pin – resulting in a value of 0. Once the float switch closes, however, the 3.3V power will flow through to the GPIO pin and cause a value of 1. A pull-down resistor must be enabled on the GPIO pin chosen to prevent short circuiting, as a closed float switch results in a direct connection between power and ground.

Ideally, the float power and GPIO pins will be placed as close together as possible, to maximise the amount of float switch cable utilised. If the pins used are placed far apart, a significant amount of float switch cabling will be wasted stretching between the pins. After considering the pin layout as shown in both Table 4-1 and Figure 4-2, we have decided to use pin 1 as the float switch's power source and pin 7 as the input. This results in a very simple schematic as shown in Figure 4-3, that is used to produce the device shown in Figures 4-4 and 4-5.

We utilised the micro USB port to power our mote, connecting it to a 2200mAh portable power-pack manufactured by *Gecko Gear* [131]. This power pack was chosen as a result of its high capacity, availability, low price, and water resistance. While this power pack was large and unable to fit into a standard pipe cap, this was seen as irrelevant for the current prototype. Future deployments of our design will use different power supplies depending on local requirements and availability and will likely use different devices. Consequentially, there is no need to specifically test the power supply being used, with testing focused on the system's functionality when implementing our design. Searching the Internet has also shown that many smaller batteries compatible with our mote design and RPZ are available.

Altogether, the prototype mote cost $85.35 AUD to develop. The RPZ was $21.41, the 16GB MicroSD card for storing its operating system was $23.59, the power supply was $24.95, and the float switch was $9.54. Bringing the total to its final value of $85.35, a transparent case was also purchased for $5.50. This inexpensive addition proves extremely valuable, as physical ruggedization and protection is an essential yet often overlooked part of mote development. Mass-manufacturing can be expected to significantly decrease this price, especially if less expensive power sources or purpose-built microcontrollers are utilised.

Table 4-1: A list of Raspberry Pi Zero (RPZ) pins, in order of their physical arrangement

Physical Number	**Purpose**	**BCM Number**	**Comments**
1	3.3V I/O		
2	5V I/O		
3	I/O	GPIO2	Reserved for I2C protocol
4	5V I/O		
5	I/O	GPIO3	Reserved for I2C protocol
6	Ground pin		
7	I/O	GPIO4	
8	I/O	GPIO14	Reserved for UART serial protocol
9	Ground Pin		
10	I/O	GPIO15	Reserved for UART serial protocol
11	I/O	GPIO17	
12	I/O	GPIO18	
13	I/O	GPIO27	
14	Ground Pin		
15	I/O	GPIO22	
16	I/O	GPIO23	
17	3.3V I/O		
18	I/O	GPIO24	

19	I/O	GPIO10	Reserved for SPI protocol
20	Ground Pin		
21	I/O	GPIO9	Reserved for SPI protocol
22	I/O	GPIO25	
23	I/O	GPIO11	Reserved for SPI protocol
24	I/O	GPIO8	Reserved for SPI protocol
25	Ground Pin		
26	I/O	GPIO7	Reserved for SPI protocol
27	Do not connect		
28	Do not connect		
29	I/O	GPIO5	
30	Ground Pin		
31	I/O	GPIO6	
32	I/O	GPIO12	
33	I/O	GPIO13	
34	Ground Pin		
35	I/O	GPIO19	
36	I/O	GPIO16	
37	I/O	GPIO26	
38	I/O	GPIO20	
39	Ground Pin		
40	I/O	GPIO21	

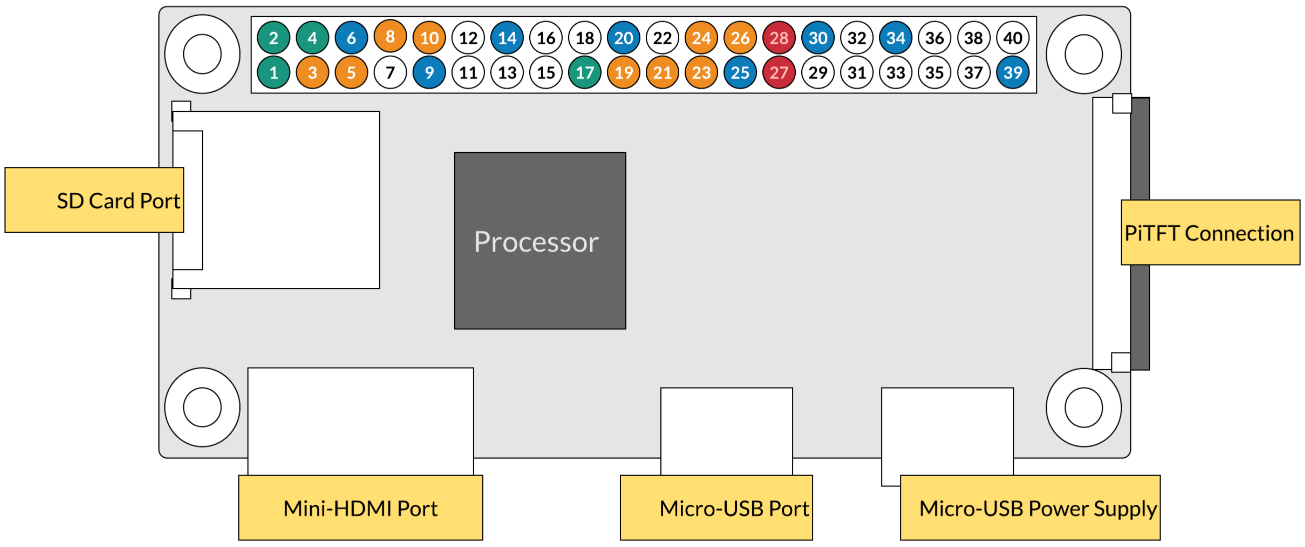


Figure 4-2: All ports (including GPIO pins) on the Raspberry Pi Zero. Each GPIO pin illustrated has the same numbering as shown in *Table 4-1*, with added colour coding for clarity. *Green* pins are power outputs, while *blue* pins are grounds. Red pins are unable to be used under any circumstances, while orange pins are able to be used but it is not recommended. No wired communication port is provided, and the Wi-Fi antenna is embedded in the board so cannot be seen.

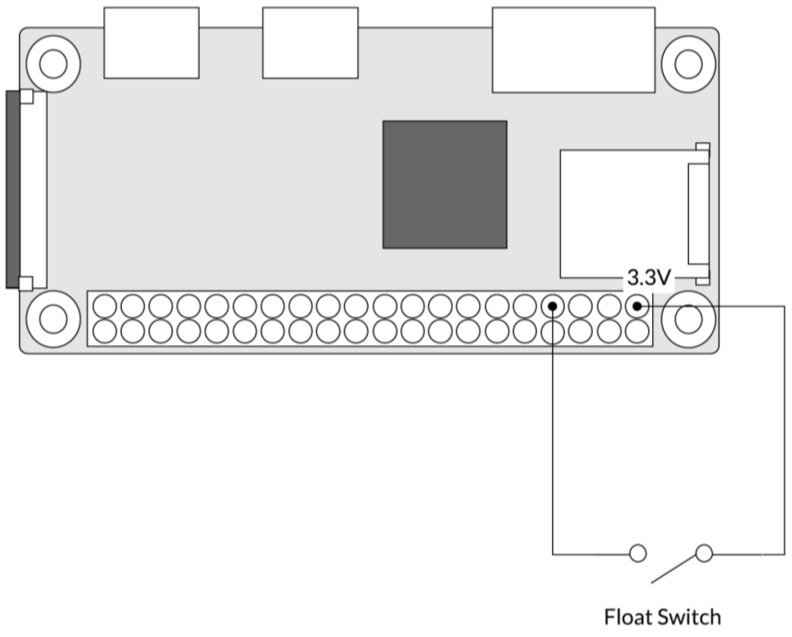


Figure 4-3: The schematic of our prototype mote. Our mote is relatively simple but highly workable, as the RPZ integrates most necessary features including a pull-down resistor. Here a float switch is shown connected to GPIO pin 1 for 3.3V power and providing input to GPIO pin 7.

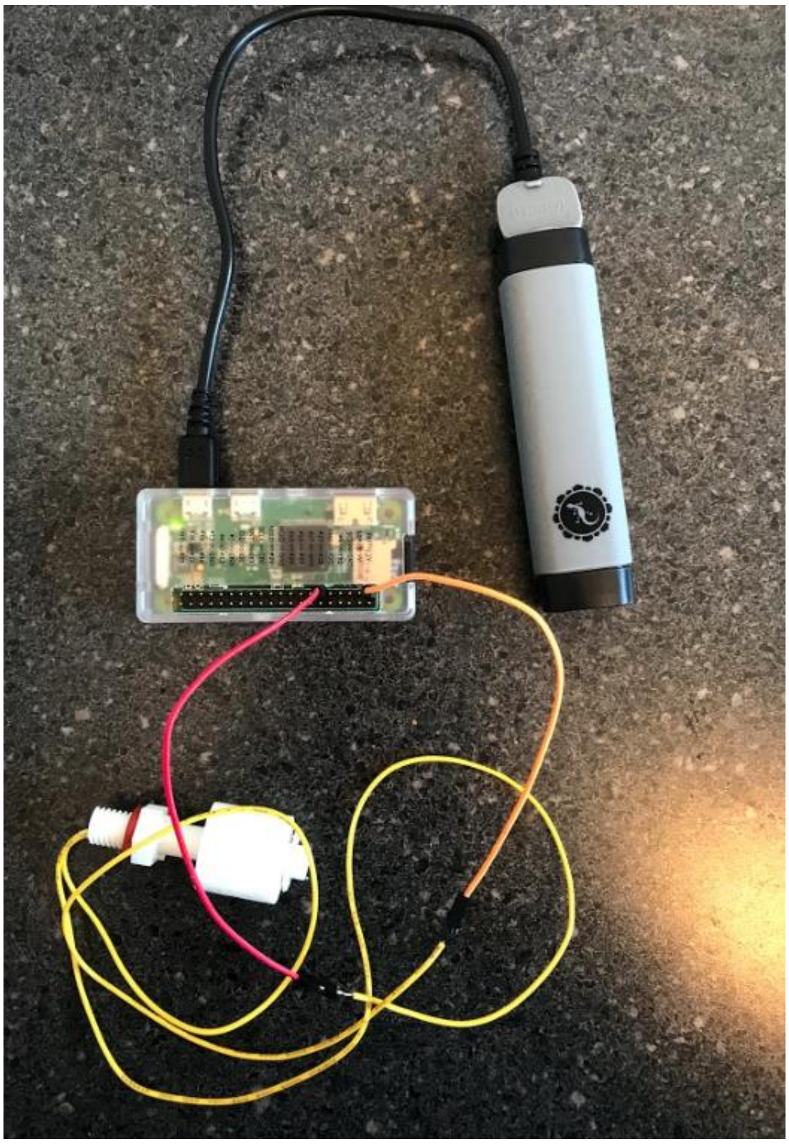


Figure 4-4: Our prototype mote. The Raspberry Pi Zero (RPZ) can be seen in the centre of the image, protected by a clear case. The float switch is connected to the device through the GPIO pins and can be seen at the bottom left-hand side of the image. Finally, the battery we utilised can be seen to the right of the RPZ, where it connects to the Micro USB power port.

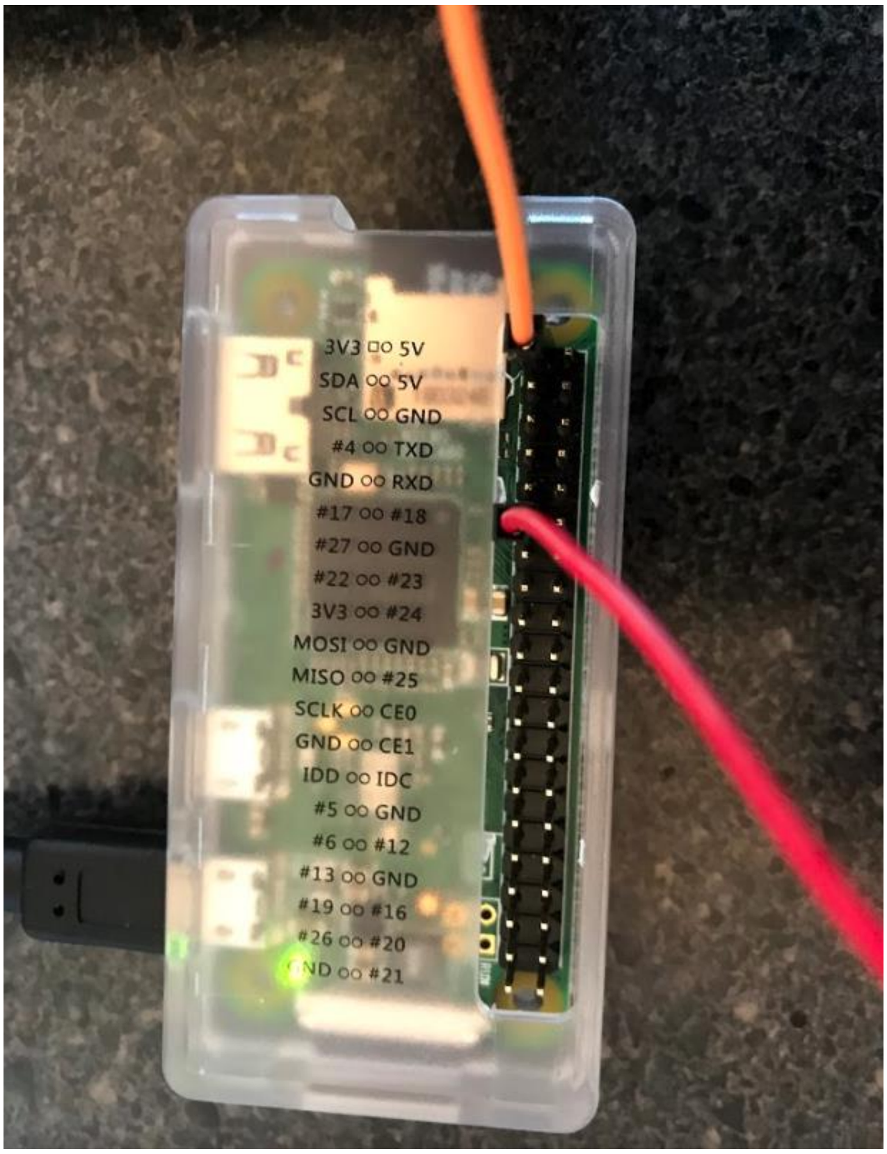


Figure 4-5: Another view of our prototype mote, clearly showing the GPIO pins. Conforming to our design and produced schematic, the float switch is connected to both a 3.3V power pin and GPIO pin 17. The power pin provides power to the circuit, and pin 17 will therefore report on the circuit's status. If the circuit is broken by the float switch being open, GPIO Pin 17 will be low current (0). Conversely, if the float switch is closed and the circuit is complete, GPIO 17 will be equal to the current flowing from the power pin (1).

Geerling [109] provides commands that conserve power by disabling the HDMI port, along with disabling a built-in LED. While the RPZ does not have a dedicated *deep-sleep* mode, disabling these components can partially replicate its functionality. Our testing will also utilise a mains power connection or portable battery unit, as we are not concerned with testing power consumption. Future, non-prototype implementations of our design will implement custom-built hardware or microcontrollers with their own power-saving functionality or increasingly efficient hardware. Table 4-2 lists the commands provided by Geerling to conserve power.

In addition to these commands, Geerling states that further power can be consumed by unplugging all peripherals and terminating background daemons. Our mote will have no keyboard, mouse, or other peripherals attached, which will save at least 50mA. We will also install the *Raspbian Lite* Linux distribution on our motes, which has a lower number of background daemons than standard Raspbian.

Table 4-2: Geerling's power-saving techniques for Raspberry Pi Zero [109].

Command	Purpose	Power Saved (mA)
/usr/bin/tvservice -o	Disable HDMI Output	25
echo none | sudo tee /sys/class/leds/led0/trigger **echo 1 | sudo tee /sys/class/leds/led0/brightness**	Disable LED	5

4.1.2 Prototype Mote Software

Python is the most common language used for prototyping and development on the Raspberry Pi and has grown into a somewhat unofficial standard for the platform. We originally intended to follow this standard and utilise the Python language, however during development of the central system we discovered a GPIO package for Node.js named *onoff* [113]. Further research into *onoff* revealed that not only was it developed considering the Raspberry Pi platform, but it would allow us to meet all design requirements.

Utilising Node.js instead of Python has many advantages, most notably allowing a shared language and platform between mote and application server. In addition, Node.js has more familiar syntax, is fully capable of asynchronous and parallel programming, and has a large online following. Our Mote software utilises both the aforementioned *onoff* package, and the same *mqtt* package used in application server development.

4.1.2.1 The *onoff* Package (GPIO)

Using the *onoff* package, developers create an object of the *Gpio* class for each GPIO pin on the board they wish to read from or write to. When creating a new *Gpio* object, developers are required to pass both the pin's identifying number and whether it is acting as an input ("in") or output ("out") to the constructor. For example, if the developer wants to configure the GPIO pin located at 17 as an input;

```
let thePin = new Gpio(17, “in”);
```

Gpio objects have a function named *watch* that accepts an anonymous function as a parameter, with that anonymous function having two of its own parameters - *err* and *value*. When current at the physical pin represented by a *Gpio* object changes, the anonymous function passed into *watch* will execute. The *value* parameter will be equal to the pin's current value, while *err* will store information on any errors that occur.

Gpio's constructor also has two optional parameters – *edge* and *options*. If the microcontroller being utilised supports interrupts, *edge* specifies which state changes in current will trigger an interrupt. Acceptable values for the edge parameter are shown in Table 4-3.

Conversely, the *options* parameter accepts a JSON-style anonymous object with one or more of the attributes shown in Table 4-4. Developers can use as many or as few of these attributes as the situation requires, and we will definitely utilise the *debounceTimeout* option to prevent false alarms. This also complies with our mote design's emphasis on preventing false alarms through switch debouncing.

Table 4-3: Acceptable *values* for the *Gpio* object’s *edge* parameter

Value	**Outcome**
`“none”`	The pin will never fire an interrupt
`“rising”`	The pin will fire an interrupt when the value changes from low current (0) to high current (1)
`“falling”`	The pin will fire an interrupt when the value changes from high current (1) to low current (0)
`“both”`	Any current changes between high (1) and low (0) or vice versa will fire an interrupt

Table 4-4: Acceptable *fields* for an anonymous object passed as the *options* parameter

Value	**Outcome**
`debounceTimeout`	Performs software debouncing on a pin by waiting a given period of time after each state change. This specifies how long to wait in milliseconds.
`activeLow`	Whether to invert values read from or written to the pin – low current will register 1, and high current will register 0. This also applies to any interrupts.
`reconfigureDirection`	If the involved pin has already had its direction (input/output) configured by another application or instance of this application, these states whether to ‘clear’ it for this object. This is expressed as a boolean value and is *true* by default.

4.1.2.2 Mote Software

Utilising the *onoff* and *mqtt* packages alongside custom Node.js code, we have developed a single script for carrying out all mote functionality named Mote.js. Like all Node.js applications the mote software is built from modules, however in this case only consists of a single module containing all needed code. With the *onoff* and *mqtt* packages already discussed, we will now examine the remaining tasks carried out with custom code and the language's own features.

Scheduled heartbeats will be handled using Node.js' built-in *setInterval()* function.– if a surcharge message is sent, the heartbeat interval will be cancelled and recreated to 'reset' it. An anonymous function will be passed to the interval, along with the length between heartbeats in milliseconds. As 24 hours is the generally agreed-on time for heartbeats, this will be an integer value of 86483647. Thankfully, the maximum value for *setInterval* is 2147483647 milliseconds (596.5 hours), so this is an acceptable value. In addition, the anonymous function will carry out the correct MQTT *publish* operation needed to send the heartbeat message to the central system.

Mote.js executes automatically when the mote is powered on, and its function can be summarised as the following algorithm;

- Check *configuration parameters*.
- Check whether the MQTT broker is reachable.
- Create *Gpio* object for Pin 7. Pass the correct parameters to debounce by waiting for the time specified in config.json and throw interrupts on all state changes.
- Bind *watch* event listener to Pin 7's *Gpio* object. This will be a function that executes every time Pin 7's state changes. More details are provided for this function below.
- If Pin 7 is at its default 0 value, execute the power-saving commands in Table 4-2.
- Create the *interval* object for sending heartbeat messages every 24 hours.

Following this, the script will wait until the *watch* event for Pin 7 is 'caught' or the heartbeat interval elapses. While actual implementations of our design should enter a *deep-sleep* mode and only *wake* on interrupts at the float switch input, we have replicated as much of this functionality as possible on the RPZ. All non-essential system functions such as Wi-Fi will be disabled by default, and only re-activate when reacting to a heartbeat or surcharge state change. Once the MQTT message has been sent in response to these events, these non-essential components will be de-activated again.

Any functions executed when Pin 7 fires a *watch* event or the heartbeat timer elapses can be classified as *event listeners*. Event listeners in our mote's software largely perform the same task, which we will briefly outline before stating the small differences.

The event listeners will both begin by re-activating the Wi-Fi adapter, before using the mqtt package's *connect* method to establish a connection with the broker. If the connection can be established, the same package's *publish* method will then transmit the correct MQTT message to the broker. If the connection cannot be established or the message cannot be sent, the failed task will be re-attempted the number of times specified in config parameters. Following this, the Wi-Fi adapter will be deactivated.

Between the two event listeners, the only significant difference is the actual message being published. These will both publish a space-separated string containing the Mote's address, a timestamp and battery level. However, the *watch* function will have an additional space-separated value at the end of the string containing the current surcharge status. In addition, the *watch* function will publish its message under the "*surcharge*" topic, while the interval will publish it under the "*heartbeat'* topic.

In addition, when a message is sent using the *watch* event listener, there is no need to execute the regular heartbeat for another 24 hours. Consequentially, the *watch* function will clear and recreate the interval function.

4.1.2.3 Mote Configuration Parameters

Like the application server, mote configuration parameters are contained in a JSON file named config.json. Table 4-5 shows the parameters we have implemented in this configuration file, and their effect on the system.

Table 4-5: Configuration parameters used by **mote.js** script.

Variable Name	**Purpose**
`moteID`	The unique ID of this mote in the application server database.
`debounceTime`	How long to wait when a surcharge state change is detected to filter switch bounce.
`gatewayAddress`	The IP address of the MQTT Broker.
`gatewayPort`	The port used to communicate with the MQTT Broker.
`surchargeAttempts`	How many times to attempt sending a *surcharge* message.
`heartbeatAttempts`	How many times to attempt sending a *heartbeat* message.

4.2 Prototype Network

Recent years have seen organisations shift from individual servers hosted on a single physical machine to many servers running on fewer, more powerful hosts. These powerful hosts can be located in an on-site server room, however, are even more frequently located at powerful offsite data centres – a paradigm often referred to as 'the cloud'. Therefore, it is reasonable to assume that many organisations deploying our solution will host the central system on one or more virtual machines.

Our *central system* design actually consists of three servers; an MQTT-SN gateway, an MQTT broker, and an Application Server. We have developed a virtual machine for each of these servers, hosted on a single physical *host* machine. All virtual servers utilise the *Ubuntu Server* operating system, however assigned resources differ between servers depending on their predicted processing requirements. The specifications for each virtual server are shown in Table 4-6. The host machine we utilised is a Windows 10 desktop computer with an Intel Core i7-4770k CPU, running 8 cores at 3.50 GHz. This host machine also has 32GB of RAM and a Nvidia GTX 980 TI graphics card – considering these points, there is a significantly large resource pool for virtual machines to draw from.

4.2.1 Virtual Network

Each virtual server was also given a virtual network adapter, and the host machine's own adapter was bridged to these adapters to share connectivity. Virtual network adapters are assigned their own IP address using the LAN's DHCP service, and all packets passing through the host machine's physical adapter are intercepted by the bridging service. Packets with a virtual network adapter's IP address are relayed to the appropriate adapter, while packets with the host machine's network address are processed as normal by the host. From an administrator's perspective, this gives the appearance of all virtual servers being individual entities on the same LAN as the physical host. This is illustrated in Figure 4-6.

Table 4-6: Specifications for each virtual machine in our virtual network.

	Processor	**Memory**	**Storage**	**Video Memory**	**Operating System**
Test MQTT Client	1 CPU	4096 MB	30.44 GB	16 MB	Ubuntu Server x64
MQTT-SN Gateway	1 CPU	2048 MB	30.44 GB	16 MB	Ubuntu Server x64
MQTT Broker	1 CPU	4096 MB	30.44 GB	16 MB	Ubuntu Server x64
Application Server	1 CPU	8192 MB	41.08 GB	16 MB	Ubuntu Server x64

MQTT servers were implemented using pre-existing software provided by the *Eclipse Foundation.* Eclipse provide a range of client and server solutions for both MQTT and MQTT-SN, including the *Mosquitto* [95] MQTT broker and MQTT-SN gateway [97] developed by Eclipse's *Paho* project. The *Paho* project is especially relevant to this research, as it involves Eclipse's attempt to develop MQTT and MQTT-SN solutions for IoT networks [98]. In addition, current literature shows that many systems developed for research using MQTT have successfully utilised the Mosquitto broker [99-103]. This gives Mosquitto an unofficial 'recommendation' among academia.

4.2.2 MQTT-SN Gateway

While developing mote software, we quickly realised that no reputable MQTT-SN library was available for the Node.js language. Upon further investigation, it appeared this issue was endemic across many languages and frameworks; MQTT-SN libraries are relatively uncommon. Developing our own MQTT-SN library would prove both difficult and time-consuming, and consequentially utilisation of MQTT-SN has been postponed. We will develop and test a robust MQTT-SN library in a future research project, and potentially even create the currently missing industry standard. However, for this project, we will use standard MQTT to prove the design's concept.

While we will not implement the MQTT-SN protocol in this prototype, we have ensured the system is fully compatible with it for future research. Previous literature provided very little information on practical implementation of MQTT-SN, so we instead turned to developer sites, specifications, and Internet blogs for previous experience. Notably, a blog by J.P. Talusan [115] provided extremely useful information. Talusan proposed that not all MQTT brokers will work well with MQTT-SN and recommended using the RSMB *(Really Small Message Broker*) [117] developed by the Eclipse Foundation's *Paho* project. RSMB is a close relative of the ubiquitous *Mosquitto* broker, however unlike its more common counterpart has full support for MQTT-SN.

4.2.3 MQTT Broker

To ensure compatibility with future research of MQTT-SN and resulting implementations, we utilised RSMB for our central system's broker. RSMB was installed under the */usr/sbin* directory on its virtual server, placed in a dedicated sub-directory named *rsmb*. Following installation, the bash script for starting the broker using a given configuration file were located in the following directory;

```
/usr/sbin/rsmb/rsmb/src
```

The bash script for running the broker is named *broker_mqtts*, and the configuration file *broker.cfg*. Before it could be used, the broker was configured to listen for the correct addresses on the correct port. We determined that the broker should communicate with clients or gateways on any IP address, and it should use the MQTT standard port of 1883. To achieve this, the following line was added to the configuration file;

```
listener 1883 INADDR_ANY
```

A *listener* is any combination of ports and IP addresses that the broker can communicate with and is defined by affixing the string listener with a port and address range. INADDR_ANY simply means that all IP addresses are acceptable for the given port.

To start the broker, the script must be executed with the configuration file to use as a parameter. To ensure the broker held root permissions, we used the following command;

```
sudo ./broker_mqtts broker.cfg
```

4.2.4 Application Server

Application server software and the MySQL database it utilised are both hosted on the same virtual server. MySQL installation followed the standard process for Ubuntu server, installed using apt-get with no changes to configuration. Editing a file named *mysqld.cnf* changes the configuration parameters and environment variables for the MySQL server, and we used this to allow communications with the application server software. This file was located at /etc/mysql/mysql.conf.d/ on the virtual server.

Configuring the MySQL server was a simple process, as only one parameter needed to be changed. The *bind-address* parameter was changed to ‘0.0.0.0’, allowing any host at any IP to access the database. While this would raise security concerns for a practical deployment, it was sufficient for our isolated prototype. While *localhost* would have given exclusive access to the application server, we required the ability to administer and configure the database from the physical host using *MySQL Workbench.*

In addition, the application server’s actual software is stored at a directory named /srv/node/app_server, where *node* is a custom directory for storing Node.js applications and *app_server* is a custom directory to store application server files. Section 2.6 describes the Node.js framework and provides more information on why it was selected.

This sub-section has only provided details on the virtual server used to host the application server’s software, and how it was configured. The actual application server software is detailed in Section 4.3, as it is sufficiently complex to warrant its own section.

4.3 Prototype Application Server

While motes produce surcharge data and MQTT components facilitate its delivery, the application server is responsible for converting the simple binary data to robust information.

There is currently no solution capable of detecting and classifying sewer blockages, especially given the inexpensive and simple nature of our motes. Consequentially, development of the server was highly complicated and had potential to be very time-consuming. Thankfully, by utilising the Node.js framework, we were able to develop the server in a practical timeframe.

4.3.1 Application Server Architecture and Modules

The application server is started by executing a script named run_appserver.js, located in a custom directory named /srv/node/app_server. This script contains the main flow of execution for the server, executing all tasks required on start-up before beginning the *run* loop. These tasks include importing all required packages, loading configuration parameters, and creating data structures that will be shared throughout all asynchronous executions. Following these, the server will attempt to establish a connection with the MQTT broker. If the connection is successful, the server will execute its infinite run loop while waiting for events – this includes MQTT messages being received.

Many of the modules utilised by the application server were developed by us to carry out the server's unique functionality. We will now detail each of these custom-built modules in the following sub-sections, as they collectively carry out Chapter 3's design. Each module and its purpose are listed in Table 4-7. These modules, along with those retrieved from a repository and developed by third parties, are located in a sub-directory named *node_modules*. As this subdirectory is searched by the required method, there is no need to use a fully qualified file path for importing each package. The *node_modules* directory also contains a JSON file named config.json that stores all configuration parameters used by the server. While future versions of the system could potentially allow these parameters to be changed while the server is running, currently these are only set once when the server is initially started. Consequentially, each change to these parameters requires the server be restarted.

Table 4-8 shows the parameters outlined in the config.json file, along with the values we have selected for our prototype. We have set the timer values to short times to carry out testing more efficiently, however real deployments should meet with field staff to determine the best values.

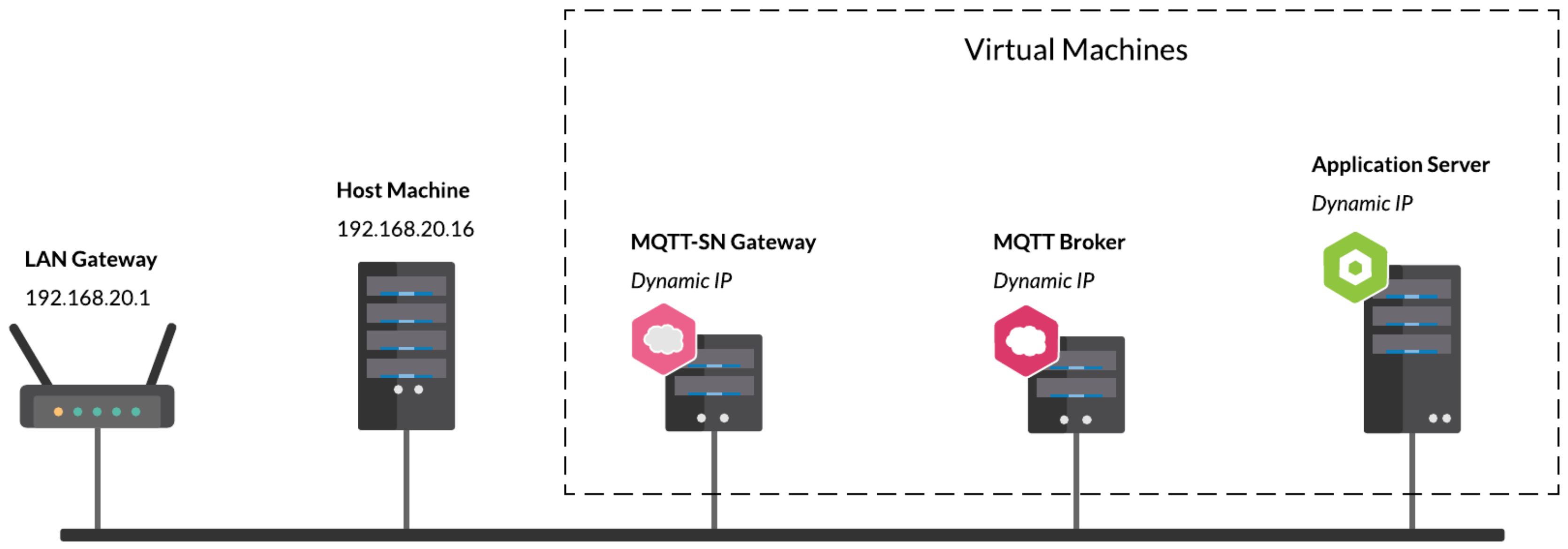


Figure 4-6: The virtualised network architecture. LAN Gateway and Host Machine are both physical devices, while the others are virtual servers. This is a standard LAN, with the *Host Machine* and *LAN Gateway* connected using a physical Cat6 medium.

Table 4-7: Application server modules.

Module File Name	**Purpose**
classes.js	All classes and global or environment variables used by the application server.
surcharge_pipeline.js	Programmatic implementation of the *surcharge pipeline* discussed in Section 3.6.5
heartbeat_pipeline.js	Programmatic implementation of the *heartbeat pipeline* discussed in Section 3.6.6.

Table 4-8: Configuration parameters stored in the config.json file.

Field	**Purpose**
brokerAddress	IP Address of MQTT broker.
falseAlarm	Length of *False Alarm Timer* (ms)
propertyConnectionOrMain	Length of *Property Connection or Main* timer (ms)
partialOrFull	Length of *Partial or Full Blockage* timer (ms)
inactive	Length of I*nactive Countdown* timer (ms)
batteryThreshold	If a Mote's battery is below this amount, notify relevant users. Set to 999 so all notifications on prototype will print.
dbAddress	Address of MySQL database. Set to localhost, as database server is located on same host.
dbPort	Port used to connect to MySQL database server.
dbUser	Account used to access MySQL database.
dbPassword	Password for above mentioned account.
dbInitial	Name of database on server to connect with.

4.3.1.1 The classes Module

In object-oriented programming, *entity* classes represent a tangible 'thing' - for our project, this includes *motes, events,* and *surcharges.* As the name suggests, the *classes* module contains all entity classes utilised throughout the server. Most entity classes are relatively simple collections of fields; however, the *Mote* class is significantly more complex. Each mote is represented by a linked list containing itself and all downstream neighbours on the same main and may itself be a 'link' in an upstream neighbour's list. Consequentially, the *Mote* class has a myriad of recursive methods used for linked-list navigation and processing.

Two additional classes are also specified by this module that do not represent 'real' entities but are instead used by the server for sharing data between parallel executions. The first, *AppServerSession*, stores data structures that must remain consistent across parallel executions of the same method. When simultaneous processes attempt to access the *AppServerSession*, they will be accessing the same memory location without creating clones. The second, *ApplicationParams*, is used to store configuration parameters and environment variables loaded from config.json.

Mote, Event, SurchargeMessage and *ClientMessage* classes have a function named *print()* that prints a detailed description of the object invoked on to standard output. As our prototype does not include the alarm and notification sub-system, this method has been invaluable for evaluating the server's performance and fulfilment of requirements. This also utilises Node's *Chalk* package [118] to create more visually output. Tables 4-9 to 4-12 show the values printed for each class and their arrangement. These provide examples of what will be printed when the method is called.

It should also be noted that the classes we developed ended up diverging from the diagram shown in Figure 3-17; this is common during software development, and often occurs when further requirements emerge.

Table 4-9: Template for ClientMessage print() function.
***********ClientMessage***********
Mote: [ClientMessage.clientID]
Timestamp: [ClientMessage.timestamp]
Battery Level: [ClientMessage.batteryLevel]

Table 4-10: Template for *SurchargeMessage* print() function.
*********SurchargeMessage*********
Mote: [SurchargeMessage.clientID]
Timestamp: [SurchargeMessage.timestamp]
Battery Level: [SurchargeMessage.batteryLevel]
Surcharge Value: [SurchargeMessage.surchargeStatus]

In both of the above outputs, the *battery level* value will be given a different coloured background depending on its value. If the battery level is above 70% it will be green, if it is below 30% it will be red, and any other value will be amber. For *SurchargeMessage*, the surcharge value will also have a red background if it is 1.

Table 4-11: Template for Mote **print()** function.
********** Mote ***********
Database ID: [Mote.id]
MAC Address: [Mote.physicalAddress]
Location: [Mote.streetAddress]
Battery Level: [Mote.batteryLevel]
Last Communication: [Mote.lastCommunication]
Surcharge Status: [Mote.surchargeStatus]
Downstream Mote:
ID: [Mote.downstreamID];
Physical Address: [Mote.downstream.physicalAddress];
Location: [Mote.downstream.streetAddress];
Surcharge Status: [Mote.downstream.surchargeStatus];

The *Mote's battery level* value will follow the same 'traffic light' colouration as *ClientMessage* and *SurchargeMessage*. Like *SurchargeMessage*, the *Surcharge Status* value will have a red background if it is 1.

Table 4-12: Template for Event **print()** method.
*********** Event ***********
Database ID: [Event.id]
Started: [Event.occurred]
Status: *See below*
False Alarm? *See below*
Spatial Classification: [Event.spatialClassification]
Time Classification: [Event.timeClassification]
Predicted Location: [Event.location]
Involved Motes:
****
See Below
***************** ***********

4.3.1.3 The surcharge_pipeline Module

As the name suggests, the *surcharge_pipeline* module implements the surcharge pipeline design we presented in Section 3.6.5. Consistent with our design, this exports a single function named *surchargePipeline*() as an 'entry point' to the pipeline. No other functions are exposed, keeping the pipeline's actual processing opaque to other modules. *surchargePipeline* accepts an *AppServerSession* and *ApplicationParams* object as parameters, meaning these are passed from the calling function into the pipeline.

surchargePipeline returns a *Promise* object when executed that is immediately returned to the main script (run_appserver.js). As typical, this will remain as a virtually empty *unresolved promise* object until all functions of the pipeline are complete or processing could not occur. Following this, the main script will notify observers through standard output that the *Promise* is resolved or rejected, inferring that the pipeline has either completed classification or encountered an error. If an error is encountered, it will be *rejected,* and details printed to standard output.

The *Promise* concept is utilised further throughout the surcharge pipeline's implementation, becoming a key component of the pipeline's architecture. Each stage of the pipeline as defined in Section 3.6.5 is given its own function that returns a *Promise* object. *surchargePipeline* itself returns a *Promise*, and each call to these functions returning their own *Promise* is called from that promise. This means that not only is *surchargePipeline* returning a *Promise*, but the *Promise* it returns is itself waiting on a sequence of *Promsies*. This can be conceptualised as the asynchronous equivalent of invoking a function that subsequently invokes several other functions, before returning a single value determined by the results of those invocations.

If a stage function's returned *Promise* object *resolves*, the next stage will be executed to return its own *Promise*, and this process will repeat until the pipeline has been completed. This is a commonly utilised architecture in Node.js programming named *promise chaining*, allowing asynchronous tasks to be completed in a defined order. Once all stages of the pipeline are complete, the final *Event* object will be *resolved* to *run_appserver*. Conversely, if a stage function's promise object *rejects*, the entire pipeline will *reject* and pass the *error* to *run_appserver*.

4.3.1.3-i Memory Management and Data Coordination

Co-ordinating the surcharge pipeline's code to handle simultaneous messages from motes was the greatest challenge encountered during our research. When multiple motes belong to the same event, these can transmit messages to the application server with little-to-no time between transmission. This not only requires the server to support multiple parallel executions of the same pipeline, but for data to remain consistent across these parallel executions. If a data structure is changed in one pipeline execution, this change must affect all others currently occurring.

Sharing data structures between parallel processes also introduces the challenges of race conditions, conflicts, and duplication of work. For example, if two messages arrive for the same event and do not change its classification, only the first should produce a notification.

Section 3.6.5's design for the surcharge pipeline utilised dictionary data structures for storing currently occurring *Events*, inactive *Events*, and active *Mote* objects. These dictionaries are stored as fields of the *AppServerSession* class, and a reference to an object of this class is passed into each invocation of *surchargePipeline*(). As the object is declared in the parent function of *surchargePipeline*, it will be *passed by reference* (memory location) and remain consistent across each invocation. Consequentially, all pipeline executions will interact with the exact same *Motes* and *Events* at consistent memory addresses.

4.3.1.3-ii Timers and Sequencing

As discussed in Section 4.2.3, all blockage and false alarm classifications require a *timer*. Node's *setTimeout*() method is used to create the required timer, however some additional effort has gone into co-ordinating them across parallel processes. Each *timer* was given a dedicated field in the *Event* class, allowing each *Event* to have its own instance of that timer. For each *timer* field added to the *Event* class, a corresponding boolean field was also added to state whether the *timer* is currently active. The surcharge pipeline populates both *timer* and boolean fields as appropriate.

Timers are often utilised by the surcharge pipeline for comparing the state of an *Event* object and its *Motes* before and after the time has elapsed. If a message arrives and initiates a *timer*, other messages for the same *Event* will continue to make changes while the *timer* is running. However, in some cases, an arriving message can make the *timer* irrelevant – to prevent wasting time, under these circumstances the *timer* will be cancelled with *clearTimeout*().

Functions utilising *timers* have been designed to return *Promise* objects. Once the *timer* has completed its countdown, or if the countdown is interrupted by an expected event, the function will *resolve* with the involved object's new state. However, if the function is invoked while the countdown is running and there is no need to stop it, the *Promise* will instead *reject*. Rejected *Promises* will not progress to the next stage of the pipeline. This ensures that only a single instance of each *Event* will pass to each stage of the pipeline, and any potential duplicates produced by parallel processing are discarded.

4.3.1.3-iii Observing Results

While the surcharge pipeline design concludes with notifying relevant parties, we have not included this feature in our prototype server. There has already been sufficient research on integrating email and communication systems with Node.js, and therefore this part is straightforward. Instead, we will utilise the *print()* method of each relevant class to observe pipeline results. This can be seen in Section 4.4, where results of testing are shown as screenshots of produced standard output.

A function named *__prototypePrintResults()* has been added to the surcharge pipeline for invoking *print()* on relevant objects.

4.3.1.3-iv Database Interaction

Our original intention was to implement a separate module for database interaction, containing several *data access classes* with functions that read and/or wrote database records for a single entity class. This is a staple of traditional object-oriented programming, where every entity class with a database presence is given a data access class. However, we quickly discovered this was not viable in Node.js – as always, the *asynchronous* nature of the language challenged traditional programming conventions.

We used Node.js' mysql.js package for communicating with the application server's MySQL database. This package provides several functions for reading and writing database records, however unlike conventional languages, each of these functions was asynchronous. Invoking one of these to perform a database operation would fail if called from a conventional function, as it would instantly return an empty object (or *unresolved Promise*) before completing the task in the background. When the task was complete, any results would be unassigned and disposed of by the garbage collector.

Instead, any invocations of *mysql.js* functions required placement in a *Promise* object. This would wait for the asynchronous database operations to be carried out, before *resolving* or *rejecting* the results. Considering the surcharge pipeline already contains a complex chain of *Promises*, we determined creating a separate module with its own Promises would be unwieldly. Instead, we performed the required database operations directly within the relevant pipeline functions.

Database operations were also limited to those absolutely necessary, with no database writes (SQL INSERT, UPDATE or DELETE methods) performed. Plenty of examples already exist proving that mysql.js database writes are possible and given our limited timeframe we only included functionality relevant for testing. As our testing focuses on surcharge detection and immediate classification, there is no benefit to be gained from writing database records. Additionally, of data created during prototyping is spurious and would require deletion before practical deployment.

4.3.1.4 The heartbeat_pipeline Module

Another module whose function is made obvious by name, *heartbeat_pipeline* implements the design presented in Section 3.6.6 for the system's heartbeat processing pipeline. As a pipeline, this is similar in structure and concept to *surcharge_pipeline* – however, this is far less complex and does not require coordination of parallel processes. Only one heartbeat can occur at any time for a given mote, and heartbeats for the same mote are separated by long periods of time. Consequentially, we have significantly decreased complexity by making this pipeline's code mostly synchronous. Asynchronous functionality is restricted to implementing *Promises* for database operations, where required by the asynchronous mysql.js library.

Like *surcharge_pipeline*, a single method named *heartbeatPipeline()* is exposed by this module as an entry point to the pipeline itself. This does not require an *AppServerSession* object as a parameter as no data structures are shared among parallel executions, however it does require the *ApplicationParams* object for accessing configuration parameters.

As we have not implemented email or other communications platforms in the prototype, this pipeline's final stage of notifying relevant parties has also been removed. Like in *surcharge_pipeline*, this has been replaced with a temporary method that invokes the relevant *print()* methods to produce standard output.

4.4 Evaluation

While our design is the primary deliverable, it must be demonstrated that the design can be practically implemented. Even the most robust design is relatively useless if it cannot be proven in reality. The prototype detailed throughout the first part of chapter has implemented our design, and in this section, we will test that implementation. Testing not only reveals flaws in the implementation, but also errors made during the actual design process. Following testing, we are able to make valid and informed suggestions for future designs or implementations.

Ideally, we would perform the full scope of our testing in a real-world environment by deploying a series of motes across a wastewater infrastructure's inspection shafts. However, this was impractical for several reasons. Deployment across wastewater infrastructure requires gaining permission from a managing utilities provider, and unfortunately, we were unable to obtain this within the timeframe of this Masters project. Our research was also conducted under significant time constraints, and this would have likely made real-world deployment impractical. In response, we have performed a single test to evaluate the system's entire process in a laboratory environment simulating wastewater infrastructure and surcharge events. Additional tests are concerned with classification algorithms present at the application server, and therefore do not require field testing as such. Consequentially, these will be carried out with a simulated virtual model.

4.4.1 Testing MQTT Communications

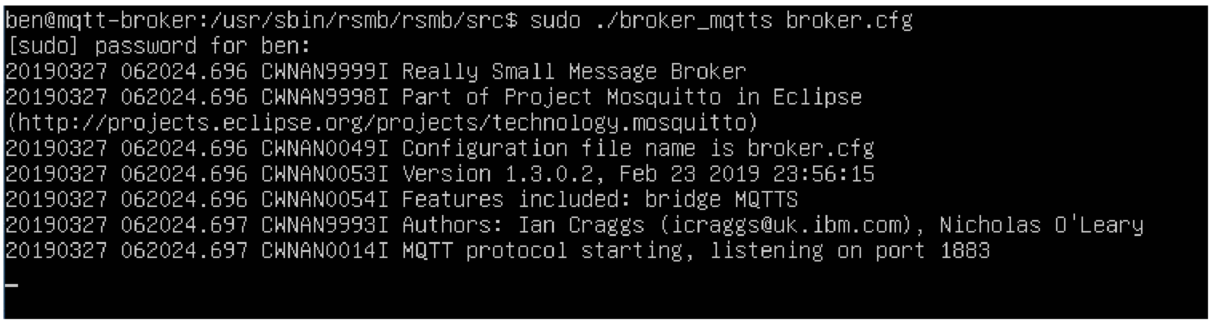
Before testing the mote or central system, it was essential to verify that MQTT communications are operating as expected. Doing so guaranteed that any future communications problems are caused by faults in hardware or software implementations, and unrelated to the actual network. This created something similar to a control variable and helped us narrow the scope of future repairs through process elimination.

To perform this testing, we developed a virtual machine named *Test MQTT Client.* This virtual machine was implemented like the others in our prototype, using a virtual network adapter bridged with the host machine's physical adapter. Table 4-6 shows the specifications for this virtual machine, and Figure 4-6 shows its place in the virtual network alongside all other prototype machines. This virtual machine was also used in many other tests we performed, as detailed in Sections 4.4.4 - 4.4.5.

All MQTT testing for MQTT communications was performed using Node.js' *mqtt* package to accurately determine if this package could carry out its intended function in the remainder of prototype software. A simple script named mqtt-test as shown in *Code 4-1* was written, publishing MQTT messages with the subject *test.* Following this, additional code was added to run_appserver.js for processing messages with the topic *test.*

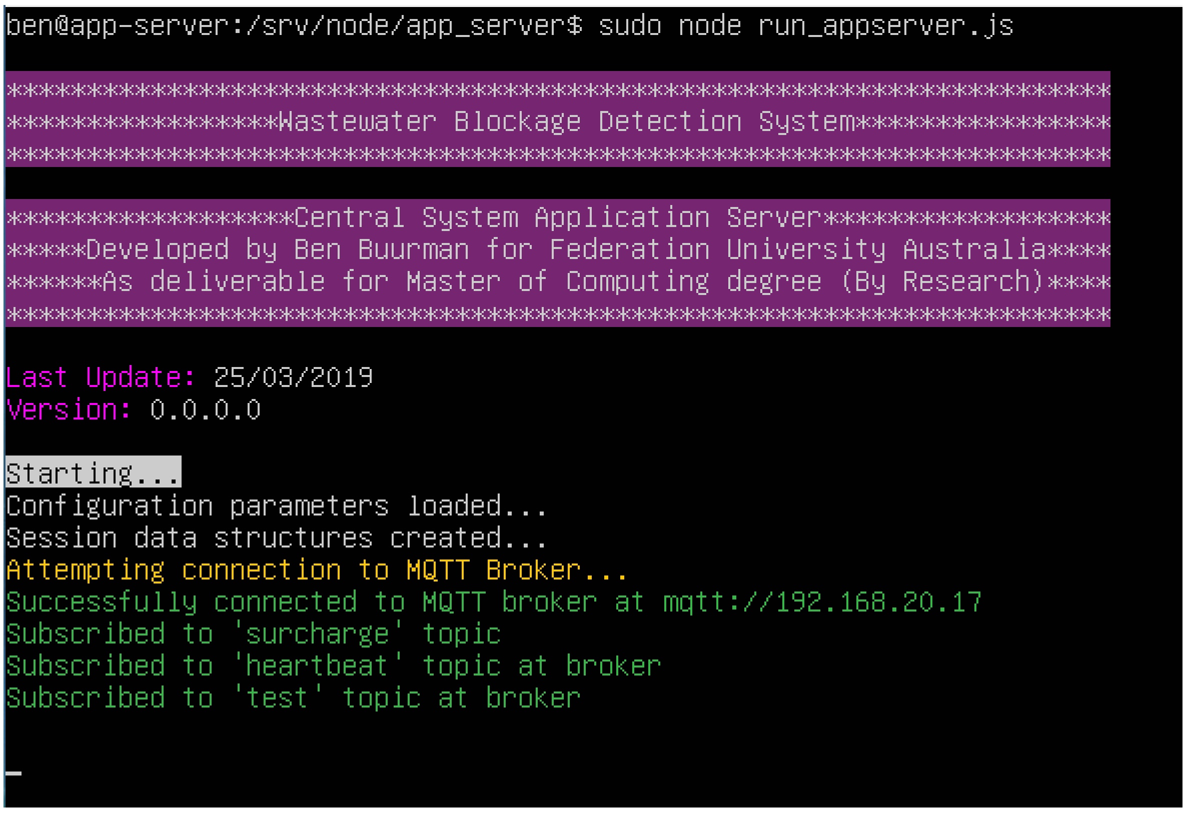
Before running this script, the MQTT broker had produced the output shown in Figure 4-7 and the application server had produced the output shown in Figure 4-8. These figures demonstrate the *default* values of these systems, allowing a comparison to be made with their output following the script's execution.

When executed, the mqtt-test produced the output shown in Figure 4.9. This showed the message had successfully been sent to the broker, displaying the contents of the message sent. The MQTT broker output displayed the messages shown in Figure 4-10, confirming that a message had been received. Finally, the application server produced the output shown in Figure 4-11, proving MQTT communications were functioning as intended. This also confirmed that Node.js' mqtt library is capable of establishing a connection and correctly facilitating publish and subscribe operations. Additionally, we could be confident that the *RSMB* broker is able to establish a connection with a single host and manage all subscriptions.



```
ben@mqtt-broker:/usr/sbin/rsmb/rsmb/src$ sudo ./broker_mqtts broker.cfg
[sudo] password for ben:
20190327 062024.696 CWNAN9999I Really Small Message Broker
20190327 062024.696 CWNAN9998I Part of Project Mosquitto in Eclipse
(http://projects.eclipse.org/projects/technology.mosquitto)
20190327 062024.696 CWNAN0049I Configuration file name is broker.cfg
20190327 062024.696 CWNAN0053I Version 1.3.0.2, Feb 23 2019 23:56:15
20190327 062024.696 CWNAN0054I Features included: bridge MQTTS
20190327 062024.697 CWNAN9993I Authors: Ian Craggs (icraggs@uk.ibm.com), Nicholas O'Leary
20190327 062024.697 CWNAN0014I MQTT protocol starting, listening on port 1883
_
```

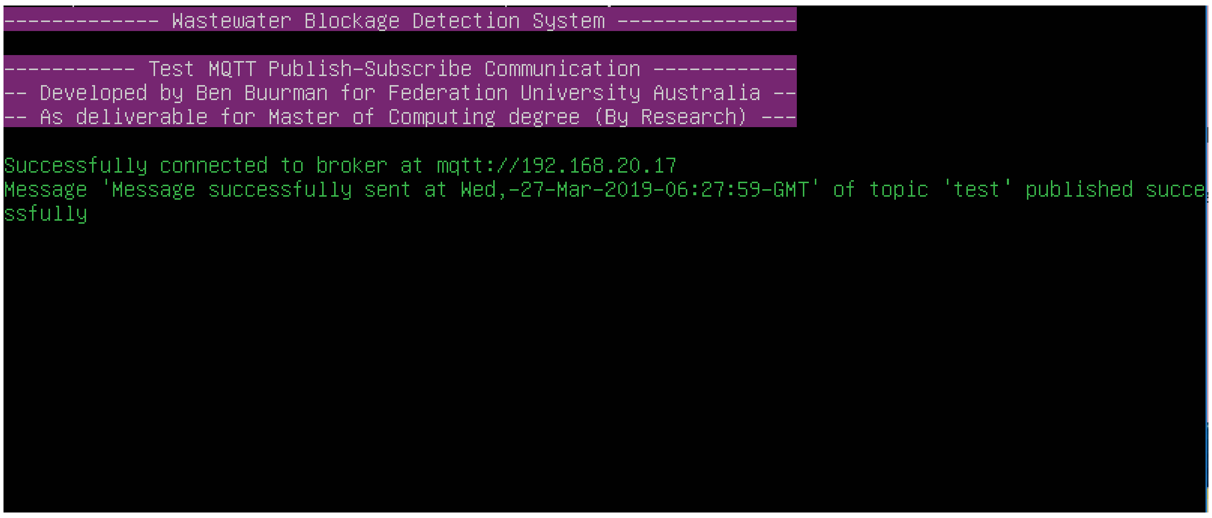
Figure 4-7: The output produced by the *RSMB* MQTT Broker when it is first started.
This provides some information on the broker itself and its developers, alongside stating the configuration file used. When the *MQTT Protocol Starting, listening on port* message is displayed, the broker is fully operational and is now waiting for any incoming messages on that port. It is now possible to test the broker, and consequentially the greater MQTT system.



```
ben@app-server:/srv/node/app_server$ sudo node run_appserver.js

*****************************************************************************
******************Wastewater Blockage Detection System*****************
*****************************************************************************

*******************Central System Application Server*******************
*****Developed by Ben Buurman for Federation University Australia****
*******As deliverable for Master of Computing degree (By Research)****
*****************************************************************************

Last Update: 25/03/2019
Version: 0.0.0.0

Starting...
Configuration parameters loaded...
Session data structures created...
Attempting connection to MQTT Broker...
Successfully connected to MQTT broker at mqtt://192.168.20.17
Subscribed to 'surcharge' topic
Subscribed to 'heartbeat' topic at broker
Subscribed to 'test' topic at broker

_
```

Figure 4-8: The output produced by the application server when first started.

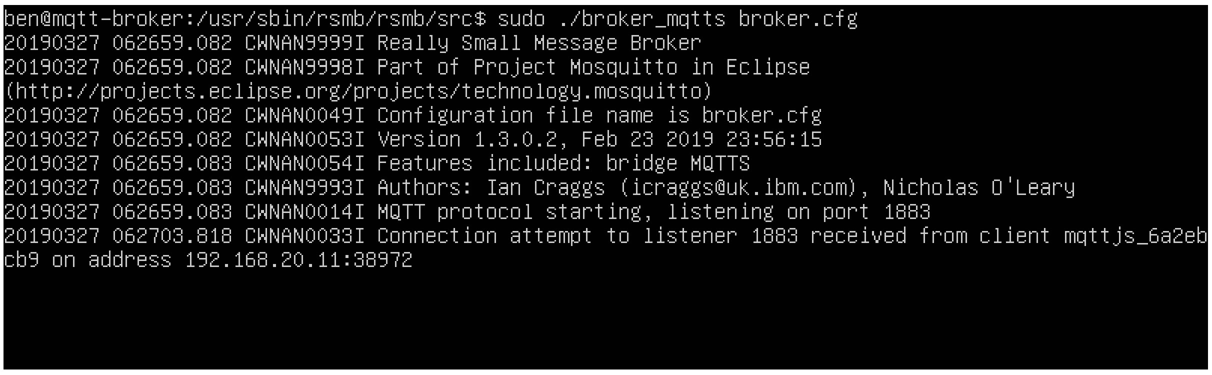
This states that the server has successfully created its required data structures (ApplicationParams and AppServerSession), which additionally implies required modules have been loaded. Following this, the output demonstrates that the server was able to successfully connect to the MQTT broker and subscribe to the surcharge, heartb eat, and test topics – the last of which is used for this test.



```
------------- Wastewater Blockage Detection System ---------------

----------- Test MQTT Publish-Subscribe Communication ------------
-- Developed by Ben Buurman for Federation University Australia --
-- As deliverable for Master of Computing degree (By Research) ---

Successfully connected to broker at mqtt://192.168.20.17
Message 'Message successfully sent at Wed,-27-Mar-2019-06:27:59-GMT' of topic 'test' published succe
ssfully
```

Figure 4-9: Output produced by the test script **mqtt-test**.

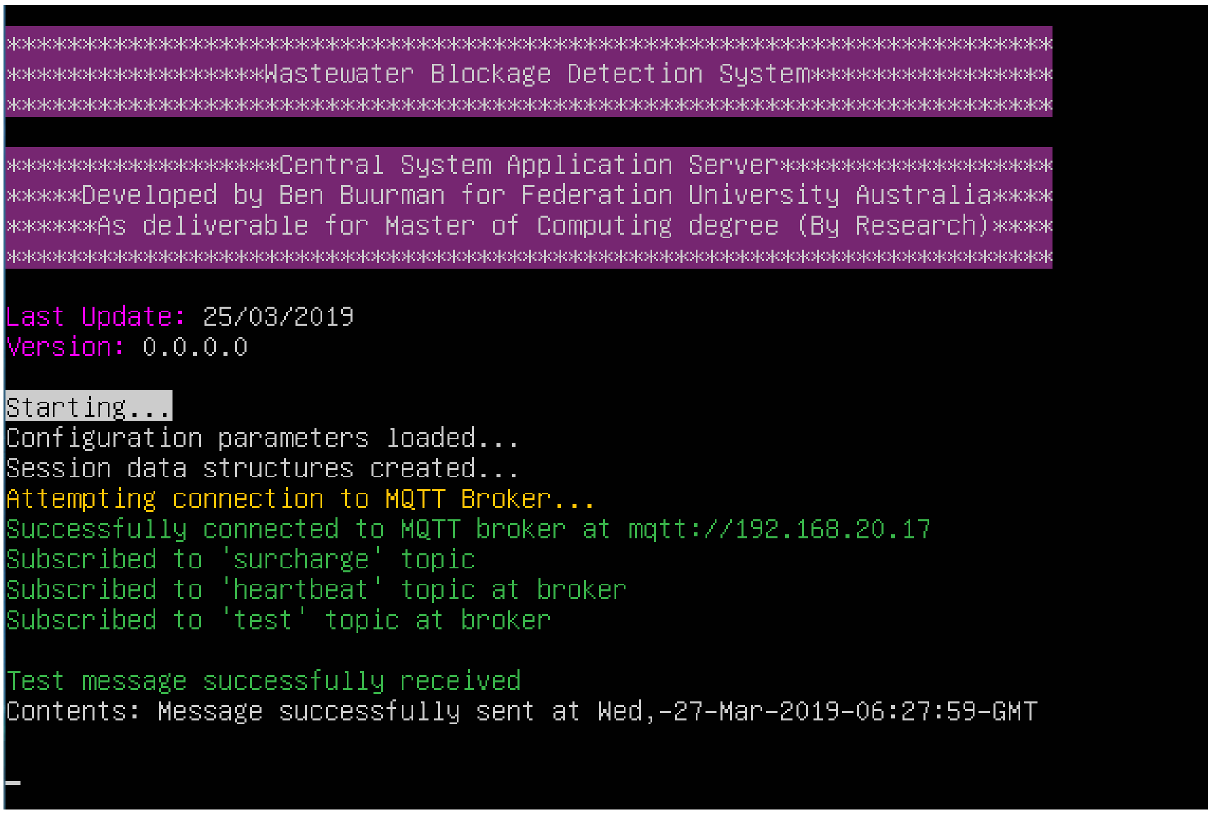
his shows that the message has been successfully *sent* to the broker and displays the message topic and contents.



```
ben@mqtt-broker:/usr/sbin/rsmb/rsmb/src$ sudo ./broker_mqtts broker.cfg
20190327 062659.082 CWNAN9999I Really Small Message Broker
20190327 062659.082 CWNAN9998I Part of Project Mosquitto in Eclipse
(http://projects.eclipse.org/projects/technology.mosquitto)
20190327 062659.082 CWNAN0049I Configuration file name is broker.cfg
20190327 062659.082 CWNAN0053I Version 1.3.0.2, Feb 23 2019 23:56:15
20190327 062659.083 CWNAN0054I Features included: bridge MQTTS
20190327 062659.083 CWNAN9993I Authors: Ian Craggs (icraggs@uk.ibm.com), Nicholas O'Leary
20190327 062659.083 CWNAN0014I MQTT protocol starting, listening on port 1883
20190327 062703.818 CWNAN0033I Connection attempt to listener 1883 received from client mqttjs_6a2eb
cb9 on address 192.168.20.11:38972
```

Figure 4-10: Broker output after the application server is started.

This confirms what is shown in Figure 4-9, that the server is able to successfully connect to the broker.



```
*********************************************************************
*****************Wastewater Blockage Detection System****************
*********************************************************************

******************Central System Application Server******************
*****Developed by Ben Buurman for Federation University Australia****
******As deliverable for Master of Computing degree (By Research)****
*********************************************************************

Last Update: 25/03/2019
Version: 0.0.0.0

Starting...
Configuration parameters loaded...
Session data structures created...
Attempting connection to MQTT Broker...
Successfully connected to MQTT broker at mqtt://192.168.20.17
Subscribed to 'surcharge' topic
Subscribed to 'heartbeat' topic at broker
Subscribed to 'test' topic at broker

Test message successfully received
Contents: Message successfully sent at Wed,-27-Mar-2019-06:27:59-GMT


_
```

Figure 4-11: Application server after **mqtt-test** sends the *test* message.

The response is stating that a message with the topic *test* has been received and displays the message's contents. It can be observed that the received message's contents are the exact same as the contents sent in Figure 4-9. Consequentially, it is proven that the broker is fully capable of processing MQTT messages, and the application server is capable of subscribing to topics and receiving messages of those topics.

Code 4-1: mqtt_test.js

```
const MQTT = require("mqtt");
const chalk = require("chalk");
const BROKER_ADDRESS = "mqtt://192.168.20.17";
//Get date and time in string format readable by application server
function currentDate()
{
    let raw = new Date();
    let utcString = raw.toUTCString();

    //Replace spaces with hyphens for processing at server. These will be
replaced with spaces again once this processing is complete.
    return utcString.replace(/ /g, "-");
}
//Publishes an MQTT message of a given topic, using the mqtt library.
function mqttPublish(topic, message)
{
    let conn = MQTT.connect(BROKER_ADDRESS);

    conn.on("connect", () =>
    {
        console.log(chalk.green("Successfully connected to broker at " +
BROKER_ADDRESS));

        conn.publish(topic, message, (err)=>
        {
            if(err)
                console.error(err);
            else
                console.log(chalk.green("Message '" + message + "' of topic '" +
topic + "' published successfully"));
            console.log("");
        });
    });
}
```

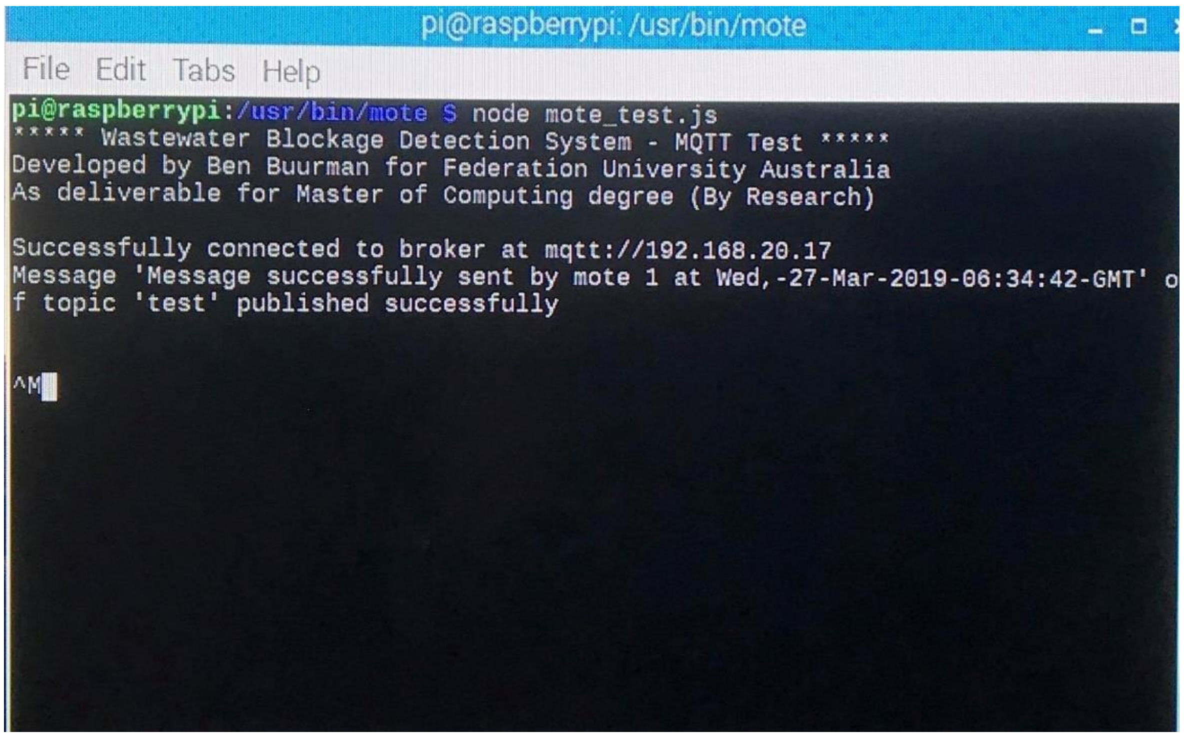
```
// 'main' and entry function for firmware.

function main()

{

    console.log(chalk.bgMagenta("------------- Wastewater Blockage Detection
System ---------------"));

    console.log("");

    console.log(chalk.bgMagenta("----------- Test MQTT Publish-Subscribe
Communication ------------"));

    console.log(chalk.bgMagenta("-- Developed by Ben Buurman for Federation
University Australia --"));

    console.log(chalk.bgMagenta("-- As deliverable for Master of Computing
degree (By Research) ---"));

    console.log("");

    //Send first message

    mqttPublish("test", "Message successfully sent at " + currentDate());

}


//Begin processing - execute main method

main();
```

4.3.2 Testing Mote Connectivity

With confirmation our MQTT infrastructure was operating as intended, we next tested whether motes were capable of connecting to and utilising this infrastructure. As our design places responsibility for initiating MQTT communications entirely on motes, their ability to access and utilise the infrastructure is critical.

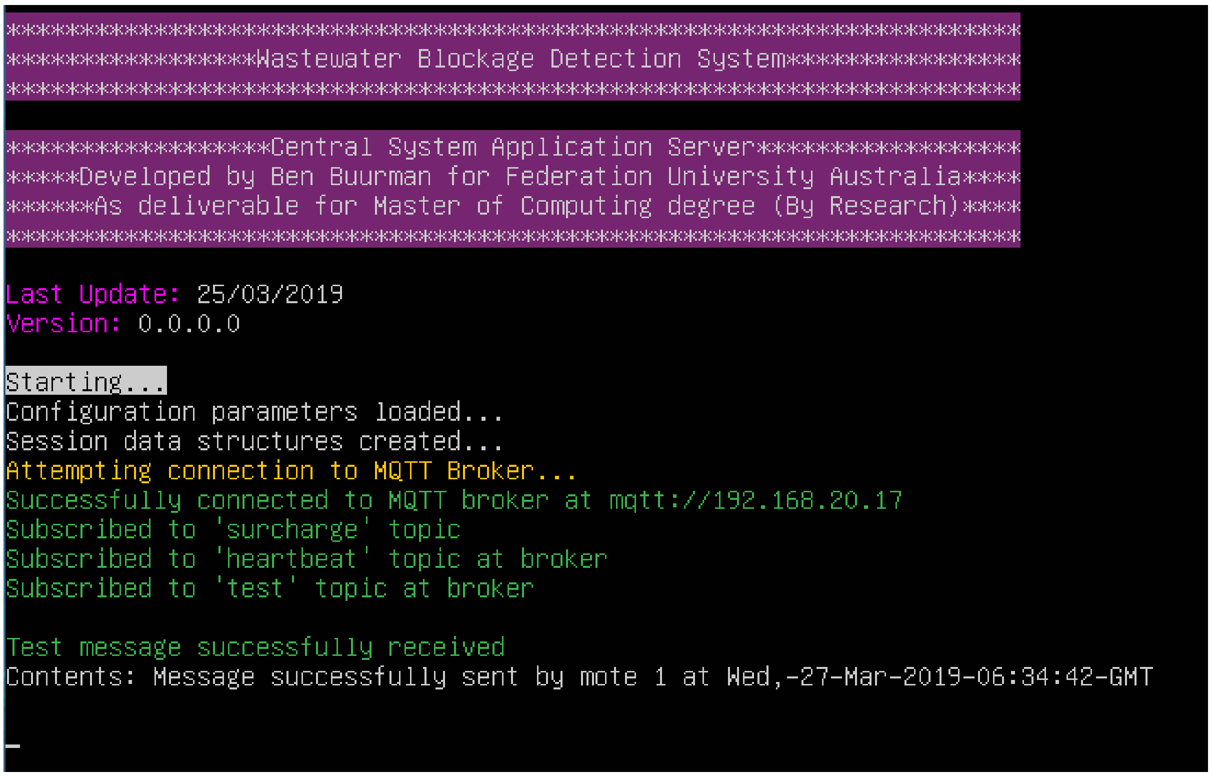
We loaded the script named mote_test shown in Code 4-3 on to our prototype mote and executed it. By doing this, we not only established the mote could access the MQTT infrastructure, but also that it is capable of communicating with the application server.

Figure 4-12 shows the mote's output when executing Code 4-3, while Figure 4-13 shows the output produced by the application server. This confirmed that not only could motes connect to the MQTT infrastructure, but they could also successfully publish messages to the application server – therefore, implementing our design's scope of data communications.



```
pi@raspberrypi:/usr/bin/mote $ node mote_test.js
***** Wastewater Blockage Detection System - MQTT Test *****
Developed by Ben Buurman for Federation University Australia
As deliverable for Master of Computing degree (By Research)

Successfully connected to broker at mqtt://192.168.20.17
Message 'Message successfully sent by mote 1 at Wed,-27-Mar-2019-06:34:42-GMT' o
f topic 'test' published successfully


^M
```

Figure 4-12: A screenshot of the output produced by the mote when running Code 4-3.
This shows that the mote is successfully able to establish a connection with the MQTT broker and can subsequently *publish* a message to the broker. This utilises the same standard PUBLISH message seen in the previous test, however, is from a physical device as opposed to a virtual machine.



```
*********************************************************************
*****************Wastewater Blockage Detection System****************
*********************************************************************

******************Central System Application Server******************
*****Developed by Ben Buurman for Federation University Australia****
*******As deliverable for Master of Computing degree (By Research)****
*********************************************************************

Last Update: 25/03/2019
Version: 0.0.0.0

Starting...
Configuration parameters loaded...
Session data structures created...
Attempting connection to MQTT Broker...
Successfully connected to MQTT broker at mqtt://192.168.20.17
Subscribed to 'surcharge' topic
Subscribed to 'heartbeat' topic at broker
Subscribed to 'test' topic at broker

Test message successfully received
Contents: Message successfully sent by mote 1 at Wed,-27-Mar-2019-06:34:42-GMT

_
```

Figure 4-13: The application server’s output after Code 4-3 is executed and the mote produces the output seen in Figure 4-12.

This shows the message is successfully received and its contents are not corrupted or altered during transmission. Note the fact that these results are identical to those seen in Figure 4-11despite the mote being a completely different device, CPU architecture, and operating system. This shows that;

i - our code is capable of running on many platforms but still producing the same results.

ii – the application server will consistently process code from different platforms.

Code 4-2: mote_test.js

```
//mote_test.js - Tests that mote can connect to central system with MQTT

//Import required modules. 'fs' and 'chalk' are in lower-case as these are de-
facto standards/expected

const GPIO = require("onoff");

const SHELL = require("child_process");

const RASPBERRY_PI = require("systeminformation");

const MQTT = require("mqtt");

const fs = require("fs");


//Import configuration parameters from config.json

let BROKER_ADDRESS = "mqtt://192.168.20.17";

let BROKER_PORT = 1883;

let MOTE_ID = "1";


//Get date and time in string format readable by application server

function currentDate()

{

    let raw = new Date();

    let utcString = raw.toUTCString();

    //Replace spaces with hyphens for processing at server. These will be
replaced with spaces again once this processing is complete.

    return utcString.replace(/ /g, "-");

}


//Publishes an MQTT message of a given topic, using the mqtt library.

function mqttPublish(topic, message)

{

    let conn = MQTT.connect(BROKER_ADDRESS);

    conn.on("connect", () =>

    {

        console.log("Successfully connected to broker at " + BROKER_ADDRESS);


        conn.publish(topic, message, (err)=>

        {
            if(err)
                console.log(err);
            else
                console.log("Message '" + message + "' of topic '" + topic + "'
published successfully");

            console.log("");
        });
    });
}
// 'main' and entry function for firmware.
function main()
{
    console.log("***** Wastewater Blockage Detection System - MQTT Test *****");
    console.log("Developed by Ben Buurman for Federation University Australia");
    console.log("As deliverable for Master of Computing degree (By Research)");
    console.log("");

    mqttPublish("test", "Message successfully sent by mote 1 at " +
currentDate())
}
//Begin processing - execute main method
main();
```

4.3.3 Testing Surcharge Detection

Confident in our mote's ability to send surcharge messages to the central system, the next step was to test whether it could actually detect surcharges and send these to the central system. This simultaneously evaluated the central system's ability to process surcharge messages; we had demonstrated it was capable of receiving messages, but these were simple strings with no advanced processing.

By testing the process of a mote detecting a surcharge, sending it to the central system, and having the central system analyse and classify it, we tested the entire system and its process. With this test complete we could confidently state our design was viable, as we had observed the entire system process operating from start to finish.

Two actions were required to perform this test. First, we developed an environment in our lab to accurately simulate a surcharge. This environment was essential for testing the system, as it allowed us to observe how motes react to an actual surcharge. Next, we created a SQL database record for the mote used during testing. The simulation environment is discussed in Section 4.3.3.1; however, we will discuss the SQL record task below.

Without adding a database record, the application server would not be able to identify the mote and could therefore not process the surcharge. We created the record shown in Table 4-13, which was assigned a primary key of 1. The mote's config.json file was updated to the values shown in Table 4-14, ensuring it would send a client ID matching the primary key of its database record. When the mote's messages arrive at the application server, the unique ID will be used to confirm the mote's identity and retrieve Table 4-13's record.

Table 4-13: Database record created for mote.	
Field/Column	**Value**
id (PK)	1
physicalAddress	080027C9DA32
streetAddress	10 Test Street
downstream	null
batteryLevel	100
surchargeStatus	0
lastCommunication	null

Table 4-14: Configuration parameter values on mote.	
Parameter	**Value**
moteID	“1”
debounceTime	5000 (5 seconds)
gatewayAddress	192.168.20.17
gatewayPort	1883
surchargeAttempts	1
heartbeatAttempts	1

4.3.3.1 Surcharge Simulation

To effectively test the mote's ability to detect surcharges, the obvious solution was to expose the mote to a surcharge and observe whether it was detected - therefore, we constructed a simple 'test bed' to install the mote in. The test bed simulated a surcharge by creating the same conditions (although with sanitary fresh water as opposed to wastewater), allowing us to observe the mote's response.

Our test bed's design is shown in Figure 4-14, and Figure 4-15 shows a photograph of the completed construct. The test bed was simple to develop, with its main body consisting of a short horizontal length of pipe and longer length of vertical pipe protruding upwards from its centre. This created a *perpendicular* shape best described as 'an upside-down T'. A hose fitting was added to one end of the horizontal pipe so the test bed could be filled with water, eventually creating a rising surface in the vertical pipe – a simulated surcharge.

The most significant component of our test bed, however, was a customised *pipe cap* that allowed the mote and float switch to be safely installed in the test bed and detect the simulated surcharge. Our cap had a small hole drilled in it, and a rigid plastic tube attached to its underside with the hollow centre placed over the hole. With this, the float switch could be placed in the shaft and attached to the end of the tube, while its wires were threaded up to emerge from the hole in the cap. The rigid plastic tube is an essential component, as it prevents the float switch from being freely suspended and moving with the surcharge surface. A design for this cap is shown in Figure 4-16, and Figures 4-17 and 4-18 show photographs of the actual product.

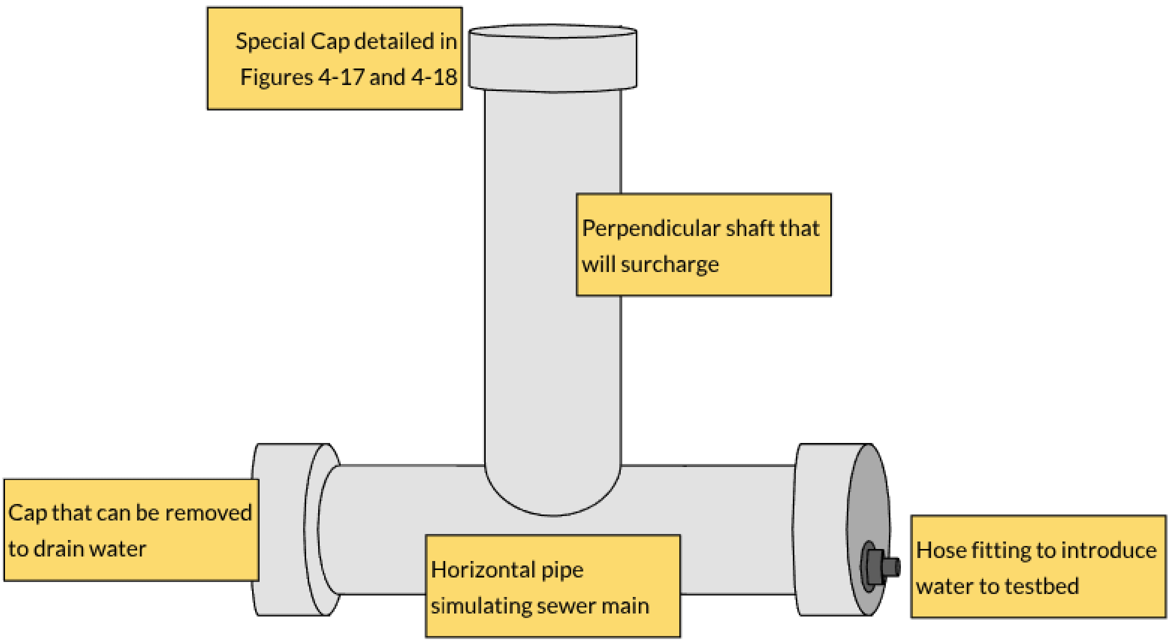

Special Cap detailed in Figures 4-17 and 4-18
Perpendicular shaft that will surcharge
Cap that can be removed to drain water
Horizontal pipe simulating sewer main
Hose fitting to introduce water to testbed

Figure 4-14: The design for our testbed. The float switch is connected using the special cap detailed in Figures 4-16 to 4-18. Water is introduced to the testbed through the hose-fitting and will fill the pipes, beginning to rise up the perpendicular shaft. This will produce identical results to a surcharge and ideally trigger the float switch.

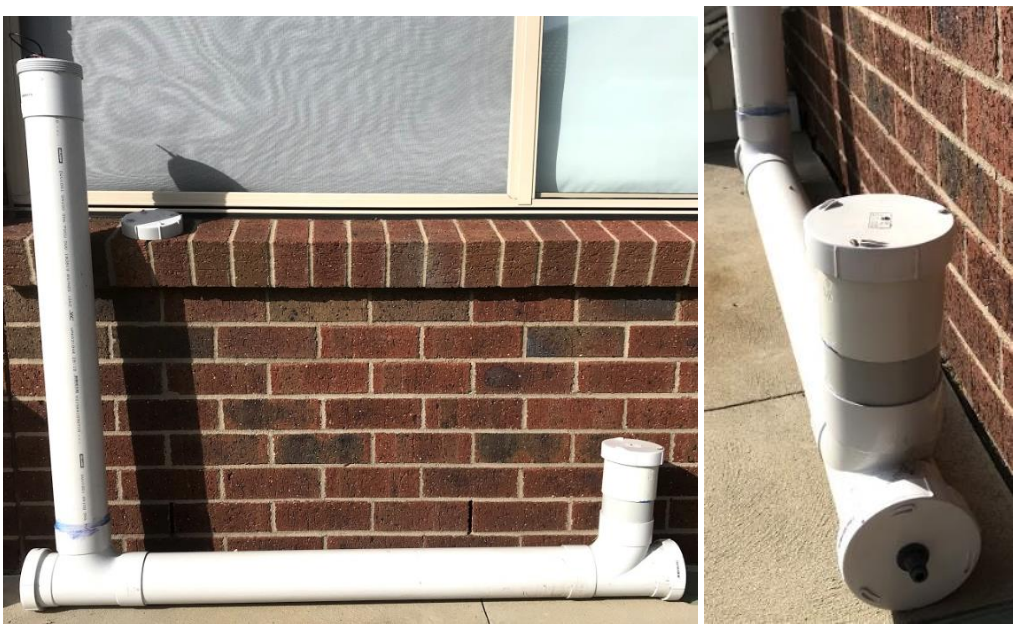


Figure 4-15: Photographs of the completed test bed from varying perspectives. Aside from an additional, shorter perpendicular pipe, this confirms to the general design laid down in the previous figure. All components specified in the design are shown in the left image, while the right image has been added to show the hose fitting for allowing water in. When water is introduced with that hose fitting, the perpendicular pipes will begin to fill with water. The mote will be placed in the tall pipe in the left image, and the float switch will activate when the rising surface reaches it.

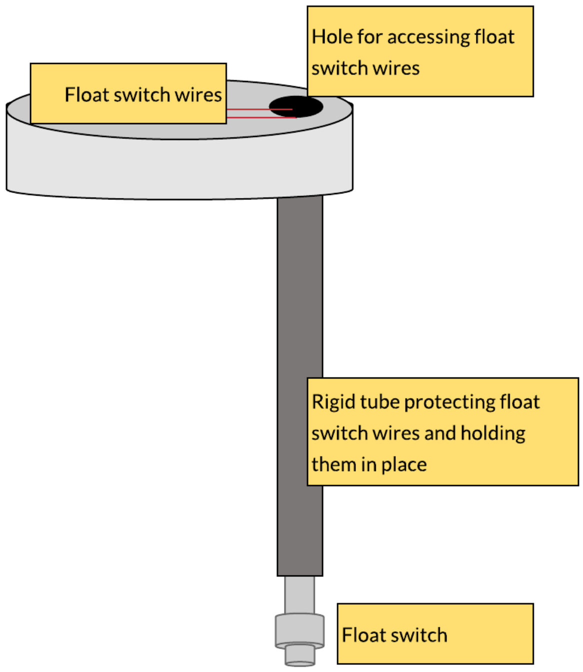

Hole for accessing float switch wires
Float switch wires
Rigid tube protecting float switch wires and holding them in place
Float switch

Figure 4-16: The design for our custom pipe cap. This sits on top of the vertical pipe in Figures 4-14 and 4-15, allowing the float switch to be correctly added to testbed and capable of detecting surcharges. Waterproofing also prevents electronic damage to the prototype mote.

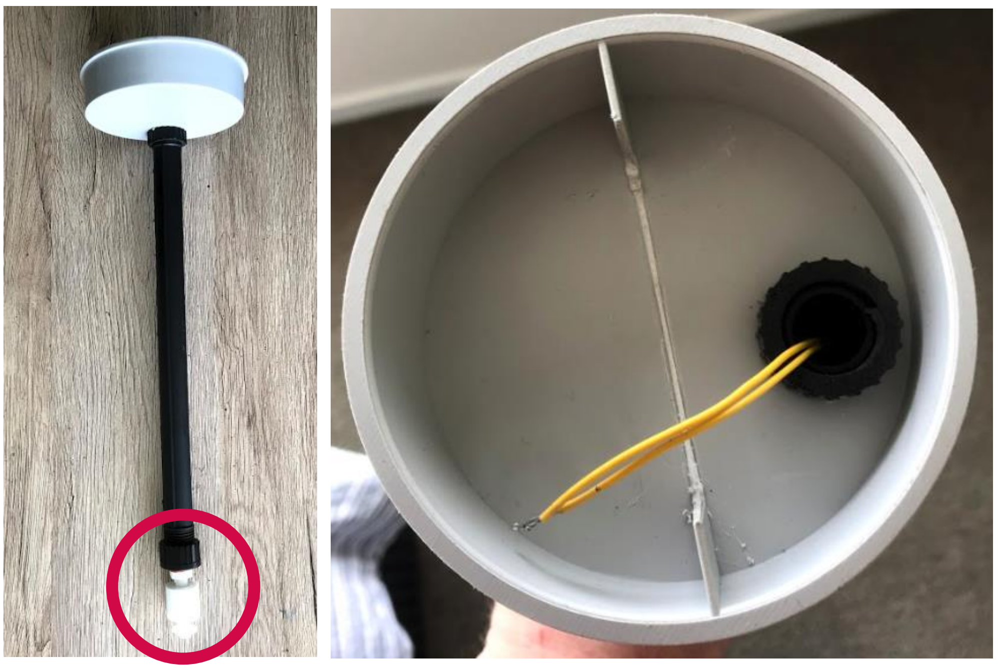


Figure 4-17: The custom pipe cap produced from our design, shown from a variety of perspectives. The left image shows how the cap looks as it will be suspended in an inspection shaft, with the float switch at the bottom activating when a rising surface pushes its contacts up. The float switch is outlined with a red circle. Conversely, the right image shows the cap's hollow space where the mote will sit. Wires from the Float switch protrude through the hole in the cap, where they can be connected to the mote.

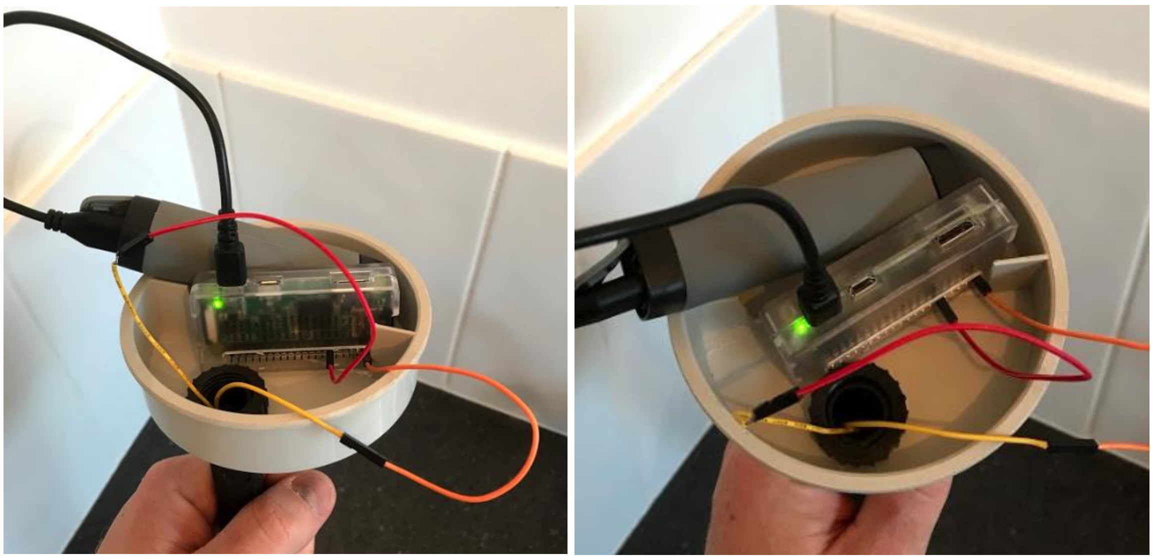


Figure 4-18: The mote placed inside our custom pipe cap. Unfortunately, as the USB battery we utilised was too large to properly fit inside the pipe cap, the mote system protruded from the cap. However, as this device is only a prototype, we were less concerned with the relative size and more with ensuring the device had power. Dedicated, small batteries are available for our purpose, however we were unable to procure one in time for testing.
Future implementations of our design will likely use a different power source or even device, and therefore the size of our battery is irrelevant for optimised testing of the developed system.

4.3.3.2 *Mote Testing and Results*

With the mote installed in our test bed as shown in the previous section, we executed *Mote.js* and began filling the test bed with water – a photograph of this process is shown in Figure 4-19. Considering that surcharge was being measured for a single shaft with consistently rising water levels, we were simulating the events of a *full property connection blockage*. As a result, we expected our application server to report a full property connection blockage. Upon examination of Table 4-13, we can elaborate further by expecting a full property connection blockage to be reported at *10 Test Street*. This is because the mote's unique ID matches the primary key of that database record, which will be retrieved by the central server after extracting the mote ID.

Our application server also utilises several *timers* to classify each surcharge event. As we have discussed several times, these will run where appropriate to compare event state before and after their countdown. We set these values unusually low for the test so it could be performed more efficiently and with less waiting. The timer lengths we used, alongside other configuration parameters, are shown in Table 4-15. These values were chosen to expedite testing while still remaining accurate. Accuracy was ensued by keeping each timer's length relative to the others consistent with expectations of real-world phenomena. For example, the *falseAlarm* timer was the shortest as real-world false alarms will be much shorter in duration. The only exception to this was the *inactive* timer, which was given an arbitrary short value to expediate testing.

To begin the test, we filled the test bed with water to cause rising levels in the vertical pipe. After enough time had passed, the surface of this rising water pushed the float switch's contacts together and simulated the surcharge threshold being reached. Following this, the application server produced the output shown in Figure 4-20. This shows that not only was the device successful in identifying and sending a surcharge, but the application server was capable of receiving and correctly classifying it.

Following this, we drained the test bed of all water as shown in Figure 4-21 to simulate a surcharge ceasing or being resolved. Almost immediately after the water was drained, the level dropped below the surcharge threshold, resulting in the application server producing the output shown in Figure 4-22. This showed that not only could the system correctly detect and identify a blockage, but it was also capable of determining when a blockage had ceased or fluctuated.

Notably, classifying the blockage took some time, while detecting that the blockage had stopped was almost instantaneous. Reviewing the algorithms used in blockage classification explains why this is the case; initial classification requires the *false alarm* and *property connection or blockage* timers in Table 4-15 to elapse - however, processing the first surcharge cessation requires no timers.

Despite the overall success, two issues were identified during this test. Notably, even after the blockage had ceased, the central server did not mark it as *resolved*. We also observed that mote battery level was incorrectly being reported as 0 – this is likely an issue with the Node package we utilised for reading system information. Still, we will attempt to resolve these issues in future implementations of our design and take note of their potential cause.

Table 4-15: Configuration parameters used on Application Server during testing.

Field	**Value**
brokerAddress	192.168.20.19
falseAlarm	20000
propertyConnectionOrMain	100000
partialOrFull	300000
inactive	25000
batteryThreshold	999
dbAddress	localhost
dbPort	3306
dbUser	ben
dbPassword	Pa$$w0rd
dbInitial	appserver

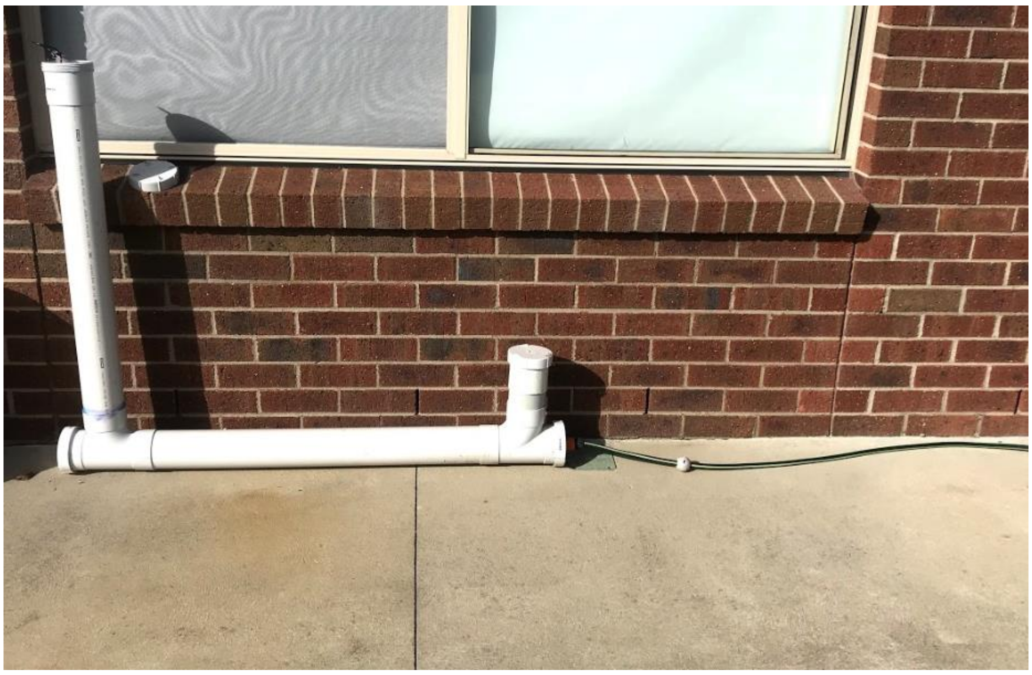
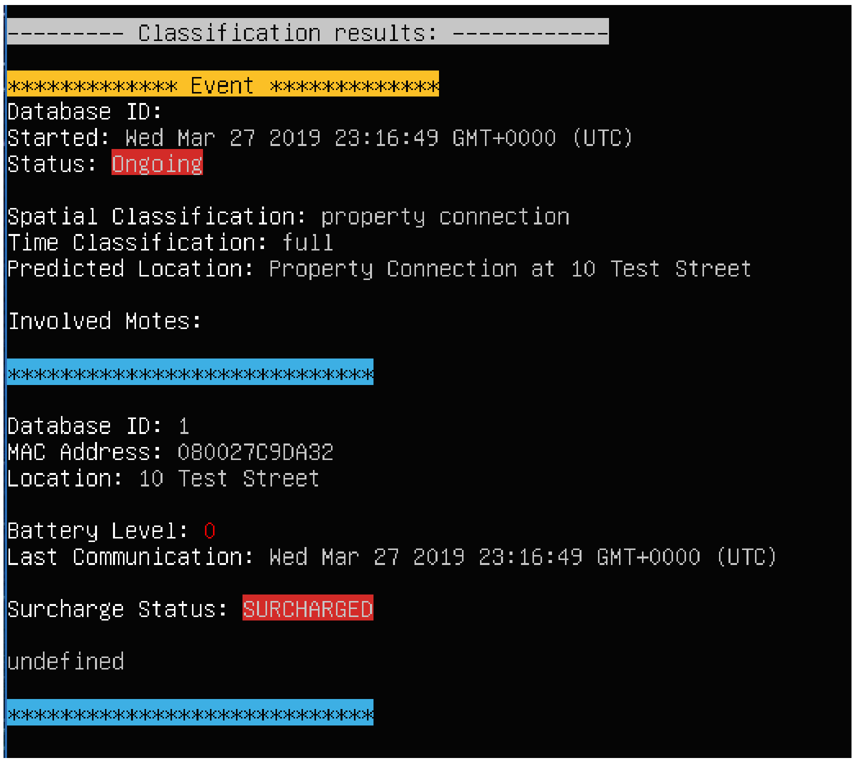


Figure 4-19: The test bed being filled with water. A standard garden hose is attached to the fitting and turned on, allowing water to fill the pipe. Our mote can be seen at the top of the tall pipe.



```
--------- Classification results: ------------

************* Event *************
Database ID:
Started: Wed Mar 27 2019 23:16:49 GMT+0000 (UTC)
Status: Ongoing

Spatial Classification: property connection
Time Classification: full
Predicted Location: Property Connection at 10 Test Street

Involved Motes:

****************************

Database ID: 1
MAC Address: 080027C9DA32
Location: 10 Test Street

Battery Level: 0
Last Communication: Wed Mar 27 2019 23:16:49 GMT+0000 (UTC)

Surcharge Status: SURCHARGED

undefined

****************************
```

Figure 4-20: The application server output following the float switch being activated as seen in Figure 4-19. Note that the Event has no Database ID – this is because we are not writing Events to the database in this prototype. Results clearly show that a Full Property Connection blockage has been detected at the false address 10 Test Street, and the involved mote is currently surcharged.

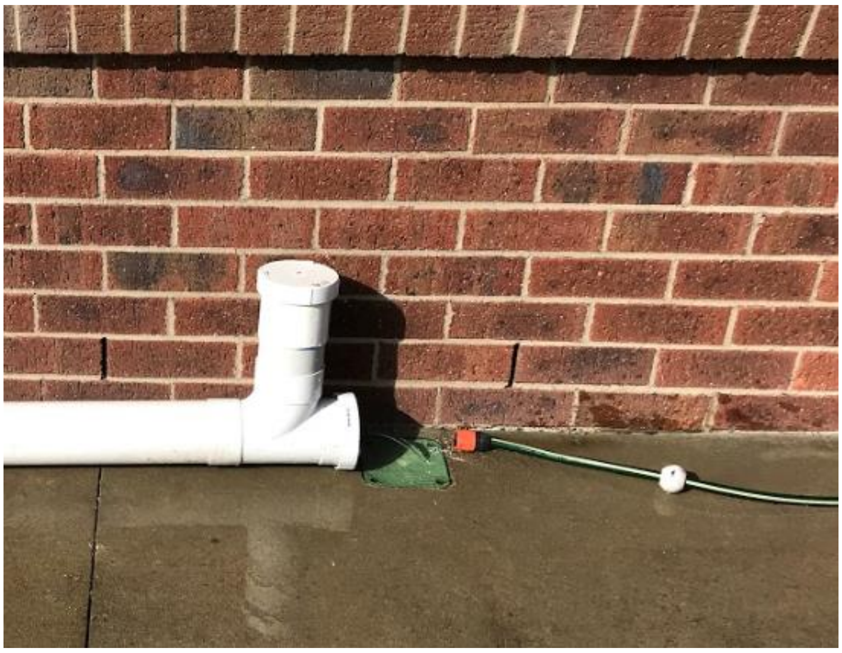
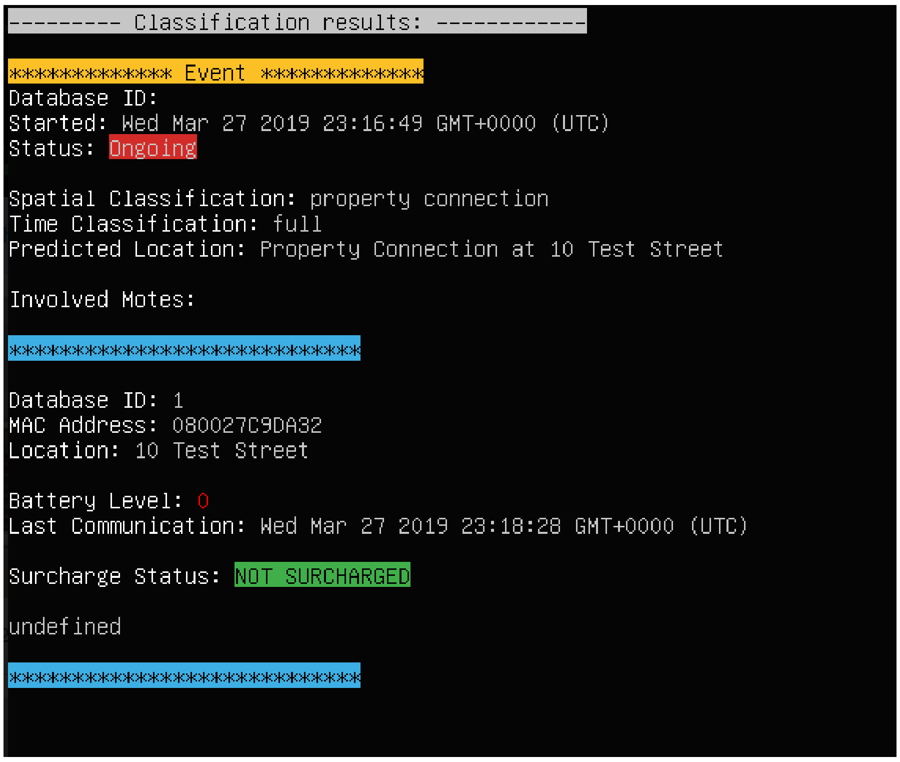


Figure 4-21: Water being drained from the test bed by removing the hose. Consequentially, the water level in the tall pipe dropped quickly and the float switch's contacts came apart. This simulates a surcharge ceasing, likely from a blockage no longer being present or from natural fluctuations in a partial blockage.



```
---------- Classification results: -------------

************** Event **************
Database ID:
Started: Wed Mar 27 2019 23:16:49 GMT+0000 (UTC)
Status: Ongoing

Spatial Classification: property connection
Time Classification: full
Predicted Location: Property Connection at 10 Test Street

Involved Motes:

******************************

Database ID: 1
MAC Address: 080027C9DA32
Location: 10 Test Street

Battery Level: 0
Last Communication: Wed Mar 27 2019 23:18:28 GMT+0000 (UTC)

Surcharge Status: NOT SURCHARGED

undefined

******************************
```

Figure 4-22: The output produced by the application server following Figure 4-21. Note that the *Surcharge Status* field now displays *NOT SURCHARGED* as its value.

4.3.4 Testing Blockage Classification

A key component of our design is its ability to classify different types of blockage – this includes false alarms, spatial classification, and time classification. While we have demonstrated our system can detect and process surcharges, we have not yet evaluated the full scope of its classification abilities. Demonstrating all classifications logically requires simulating all types of blockage, some of which cause surcharges across multiple motes. For the reasons discussed at the start of Section 4.1 limiting our ability to test motes in a real-world deployment we will perform these tests in virtual simulated environment. This environment is delivered through a *virtual* model of a sewer main with three property connections, each of which have an installed mote, and is displayed in Figure 4-23.

Three property connections were modelled as this is the minimum number required to test the entire range of blockages the system can classify. If two connections are present, there will be no observable difference between two blockages at neighbouring property connections or a main blockage. However, if three connections are present, surcharges can be simulated at both the first and last connection with no surcharge in the middle one. This produces a distinct observable result to a main blockage.

To successfully model a mote for each connection in the virtual model, three *Mote* records needed to be added to the application server database. The *Mote* record from the previous real-world test has been reused, while two additional records have been added as detailed in Tables 16-17. The *downstream* value provides a link between each Mote as discussed throughout our research.

A Node.js script was written for each simulated event, publishing MQTT surcharge messages identical to those produced by the actual event simulated. Each message will contain the appropriate mote ID, mapping it to one of the database records in Tables 4-13, 4-16 and 4-27. From the application server's perspective, these will be indistinguishable to messages sent from actual motes with real surcharges occurring. We have also ensured these scripts produce verbose output, allowing us to observe the messages published and verify their content and timing are correct.

These simulation tests are detailed in the following sub-sections with each providing a table detailing the test, a copy of the code used to perform the simulation, and screenshots of results. The code included in this section has comments and standard output print commands removed, however full versions of each code can be found in the appendices. In addition, the *time* column is relative and starts when the first message is sent. It also cannot be guaranteed that events will occur at the exact times stated, so these can be thought of as *best-effort*. Battery level values in the published MQTT messages were also been fabricated for these tests, as the script is running on a virtual machine.

Table 4-16: Database record created for Mote 2.

Field/Column	**Value**
id (PK)	2
physicalAddress	40cc1cd26e64 (Randomly generated)
streetAddress	9 Test Street
downstream	1
batteryLevel	70
surchargeStatus	0
lastCommunication	null

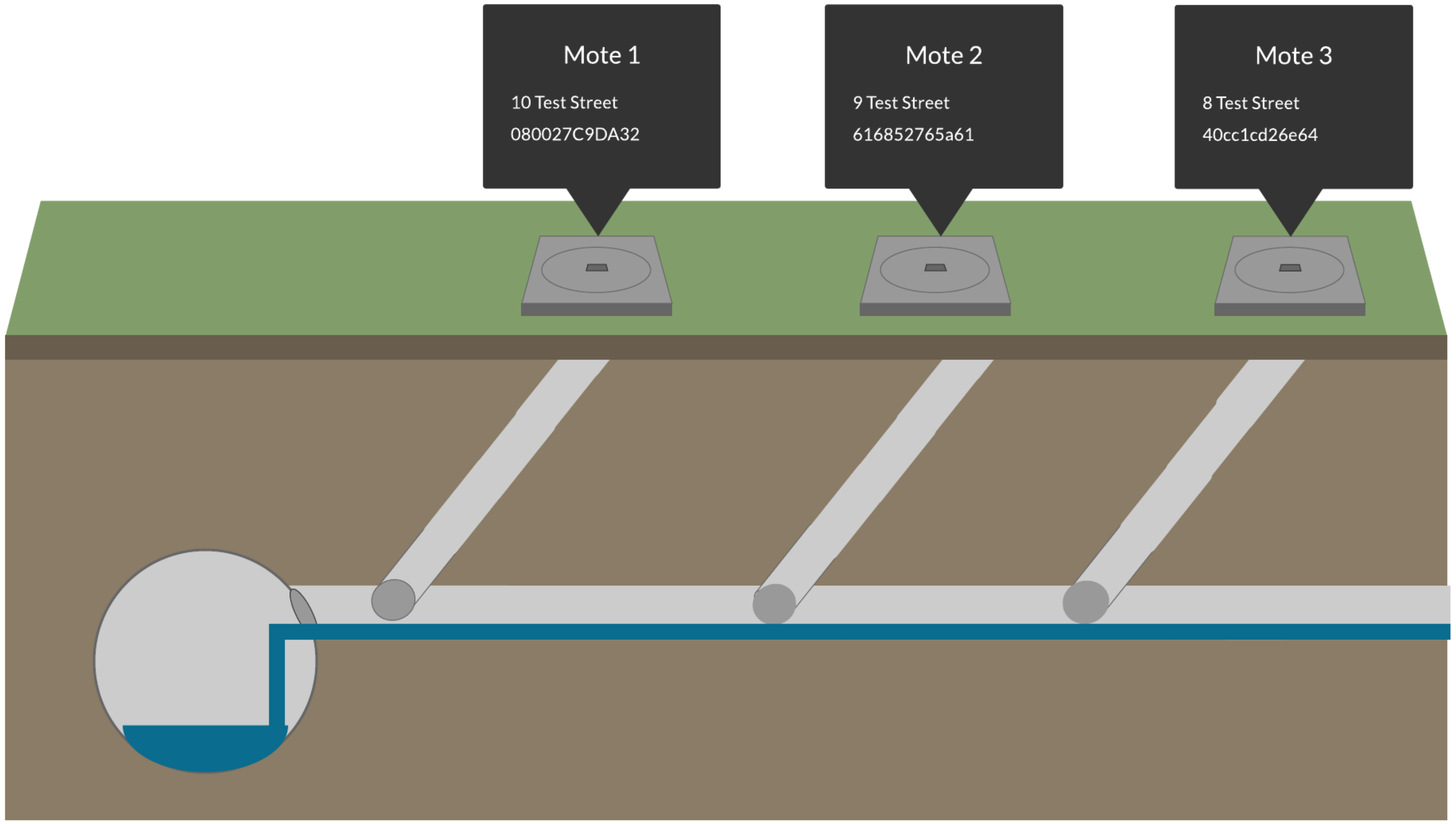


Figure 4-23: The virtual model produced by our simulation scripts. Each mote’s ID and MAC address is shown above the simulated property connection, as this is where the mote would theoretically be installed.

Table 4-17: Database record created for Mote 3.

Field/Column	Value
id (PK)	3
physicalAddress	616852765a61 (Randomly generated)
streetAddress	8 Test Street
downstream	2
batteryLevel	30
surchargeStatus	0
lastCommunication	null

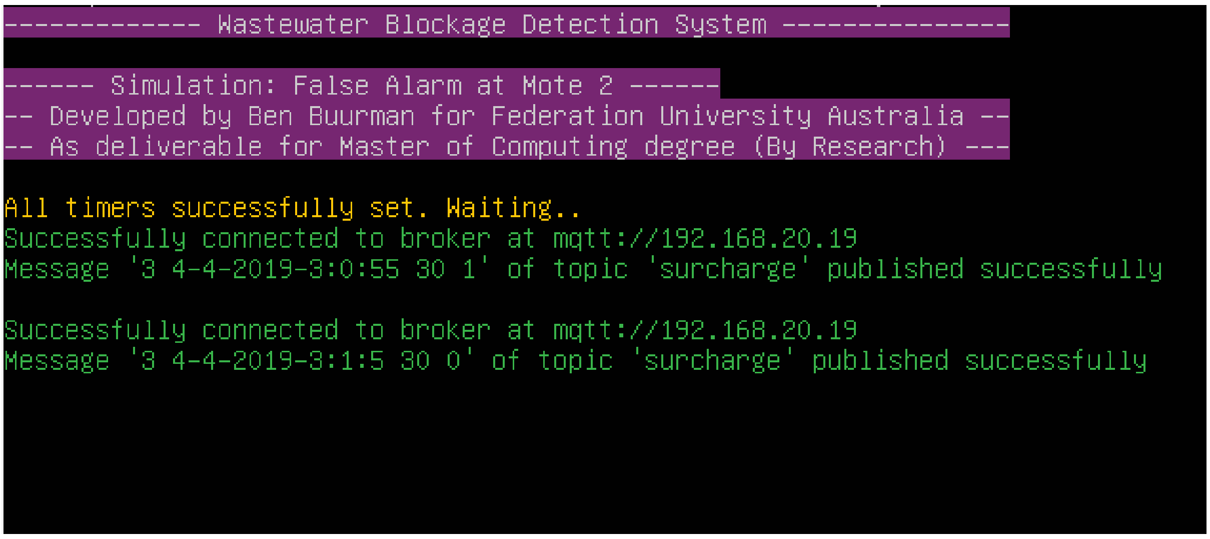
Having designed the simulation scenario, in the following sub-section we test false alarms alongside blockage classifications using the simulation data shown in the above details. The simulation scenario is constructed by utilising this data.

4.3.4.1 *False Alarm Detection*

The ability to differentiate *false alarms* from genuine blockages is a critical requirement of our system, having been discussed since Chapter 1 and elaborated throughout the remainder of this thesis. Consequentially, we will perform this test first by simulating a false alarm event. This simulation is carried out when *false_alarm.js* delivers an initial *surcharge* message for Mote 1, followed by a second one 10 seconds later. As this time falls below the false alarm timer of 20 seconds we specified, this should be detected as a false alarm.

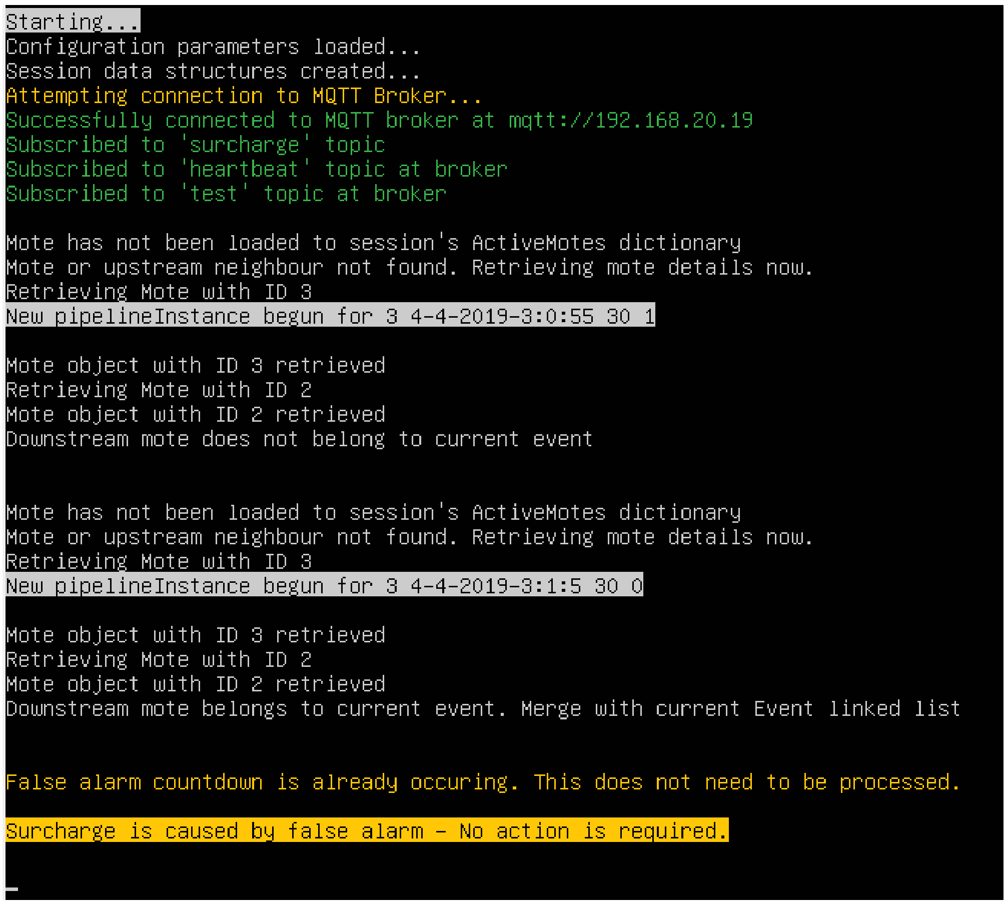
Figure 4-24 shows the output produced by *false_alarm.js* when it has completed execution, and Figure 2-25 shows the application server's output. These collectively demonstrate that the application server is capable of differentiating false alarms from genuine blockages if the timer is set correctly, which fulfils a key requirement of our research.

Table 4-18: Test plan for verifying false alarm classification.			
Event Simulated:	False Alarm at Mote 1		
Script Used:	**false_alarm.js**		
Messages Sent:			
Time:	**Mote ID:**	**Battery Level:**	**Surcharge Status:**
-	2	30%	1
0:10	2	30%	0



```
------------- Wastewater Blockage Detection System ---------------

------ Simulation: False Alarm at Mote 2 ------
-- Developed by Ben Buurman for Federation University Australia --
-- As deliverable for Master of Computing degree (By Research) ---

All timers successfully set. Waiting..
Successfully connected to broker at mqtt://192.168.20.19
Message '3 4-4-2019-3:0:55 30 1' of topic 'surcharge' published successfully

Successfully connected to broker at mqtt://192.168.20.19
Message '3 4-4-2019-3:1:5 30 0' of topic 'surcharge' published successfully
```

Figure 4-24: The output produced by **false_alarm.js** when all messages to simulate a false alarm's rapid fluctuations have been sent. The message strings show the 10 second difference between the two being sent.



```
Starting...
Configuration parameters loaded...
Session data structures created...
Attempting connection to MQTT Broker...
Successfully connected to MQTT broker at mqtt://192.168.20.19
Subscribed to 'surcharge' topic
Subscribed to 'heartbeat' topic at broker
Subscribed to 'test' topic at broker

Mote has not been loaded to session's ActiveMotes dictionary
Mote or upstream neighbour not found. Retrieving mote details now.
Retrieving Mote with ID 3
New pipelineInstance begun for 3 4-4-2019-3:0:55 30 1

Mote object with ID 3 retrieved
Retrieving Mote with ID 2
Mote object with ID 2 retrieved
Downstream mote does not belong to current event


Mote has not been loaded to session's ActiveMotes dictionary
Mote or upstream neighbour not found. Retrieving mote details now.
Retrieving Mote with ID 3
New pipelineInstance begun for 3 4-4-2019-3:1:5 30 0

Mote object with ID 3 retrieved
Retrieving Mote with ID 2
Mote object with ID 2 retrieved
Downstream mote belongs to current event. Merge with current Event linked list


False alarm countdown is already occuring. This does not need to be processed.

Surcharge is caused by false alarm - No action is required.

_
```

Figure 4-25: The application server’s output after receiving the second message. This shows that the simulation has correctly been classified as a false alarm. The *False alarm countdown is already occurring* message appears when a message arrives while the false alarm countdown is elapsing and is only used for our own debugging.

Code 4-3: false_alarm.js

```
const MQTT = require("mqtt");

const chalk = require("chalk");


const BROKER_ADDRESS = "";


const MOTE_2 = ["", "30"];


//Get date and time in string format readable by application server

function currentDate()

{

    let raw = new Date();
    let yy = raw.getFullYear().toString();
    let MM = (raw.getMonth() + 1).toString();
    let dd = raw.getDate().toString();
    let hh = raw.getHours().toString();
    let mm = raw.getMinutes().toString();
    let ss = raw.getSeconds().toString();

    return (dd + "-" + MM + "-" + yy + "-" + hh + ":" + mm + ":" + ss);
}

//Publishes an MQTT message of a given topic, using the mqtt library.
function mqttPublish(topic, message)
{
    let conn = MQTT.connect(BROKER_ADDRESS);

    conn.on("connect", () =>
    {
        console.log(chalk.green("Successfully connected to broker at " +
BROKER_ADDRESS));

        conn.publish(topic, message, (err)=>
        {
            if(err)
                console.error(err);
            else
                console.log(chalk.green("Message '" + message + "' of topic '" +
topic + "' published successfully"));

            console.log("");
        });
    });
}

// 'main' and entry function for firmware.
function main()
```

```
{
    console.log(chalk.bgMagenta("------------- Wastewater Blockage Detection
System ---------------"));
    console.log("");
    console.log(chalk.bgMagenta("------ Simulation: False Alarm at Mote 2 ------
"));
    console.log(chalk.bgMagenta("-- Developed by Ben Buurman for Federation
University Australia --"));
    console.log(chalk.bgMagenta("-- As deliverable for Master of Computing
degree (By Research) ---"));
    console.log("");

    //Send first message
    mqttPublish(getMoteString(MOTE_2, 1));

    //Wait 1 minute (600000 milliseconds) and send next message
    setTimeout( ()=>
    {
        mqttPublish(getMoteString(MOTE_1, 0));

    }, 300000);

    console.log(chalk.yellow("All timers successfully set. Waiting.."));

}

//Begin processing - execute main method
main();
```

4.3.4.2 Full Main Blockages

Our testing with the mote prototype proved the application server's ability to classify full blockages at the property connection. This consequentially demonstrated its abilities to both spatially classify property connection blockages, and temporally classify *full* blockages. Following this, we verified its ability to classify *full* blockages located at the main. This involves the retrieval of multiple *main* records, and the ability to determine their connections with each other.

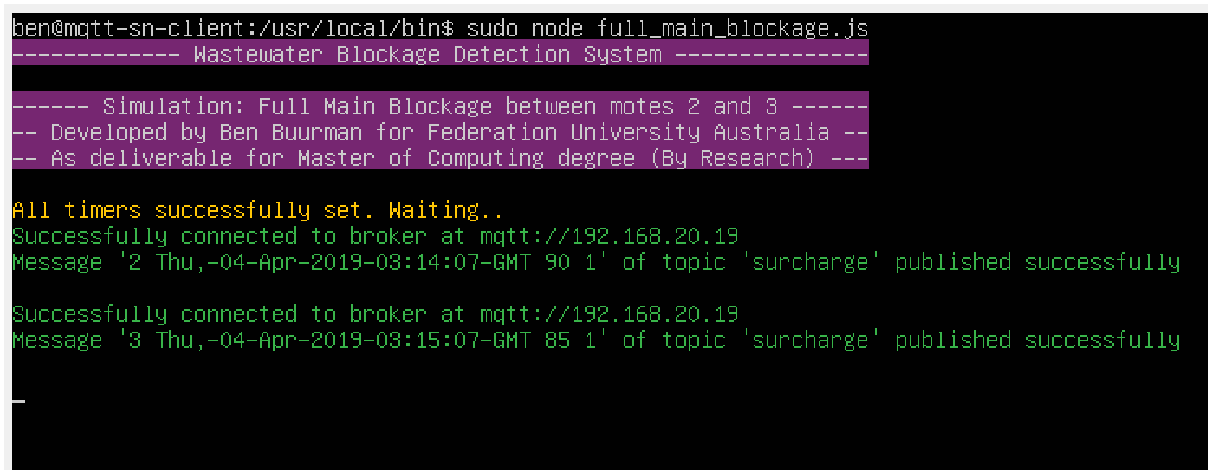
This simulation was carried out by a script named *full_main_blockage.js*, which sent two *surcharge* messages; first for Mote 2, and then for Mote 3. These were sent one minute apart, as while this is a very short time it is still longer than the *false alarm* counter. In addition, it is also shorter than the spatial classification timer of 2 minutes. Being shorter than this second countdown means that we can test what happens when a message arrives while the countdown is elapsing – ideally, it will cause that countdown to cancel and only display a single result.

Figure 4-26 shows the output produced by *full_main_blockage.js* upon completing execution. This confirms that both messages were sent a minute apart, meaning that results at the application server are a realistic approximation of our system's ability to handle full main blockages.

Following this, Figure 4-27 shows the output produced at the application server when finished classification. This is the best possible outcome, as it has correctly classified the two surcharges as a *main blockage* and determined their location. Not only this, only a single output was produced. This shows that the spatial classification timer started by the first message was cancelled, and processing for that message stopped in favour of the second. In combination with previous results, this shows our system is very competent with handling parallel processing and synchronisation.

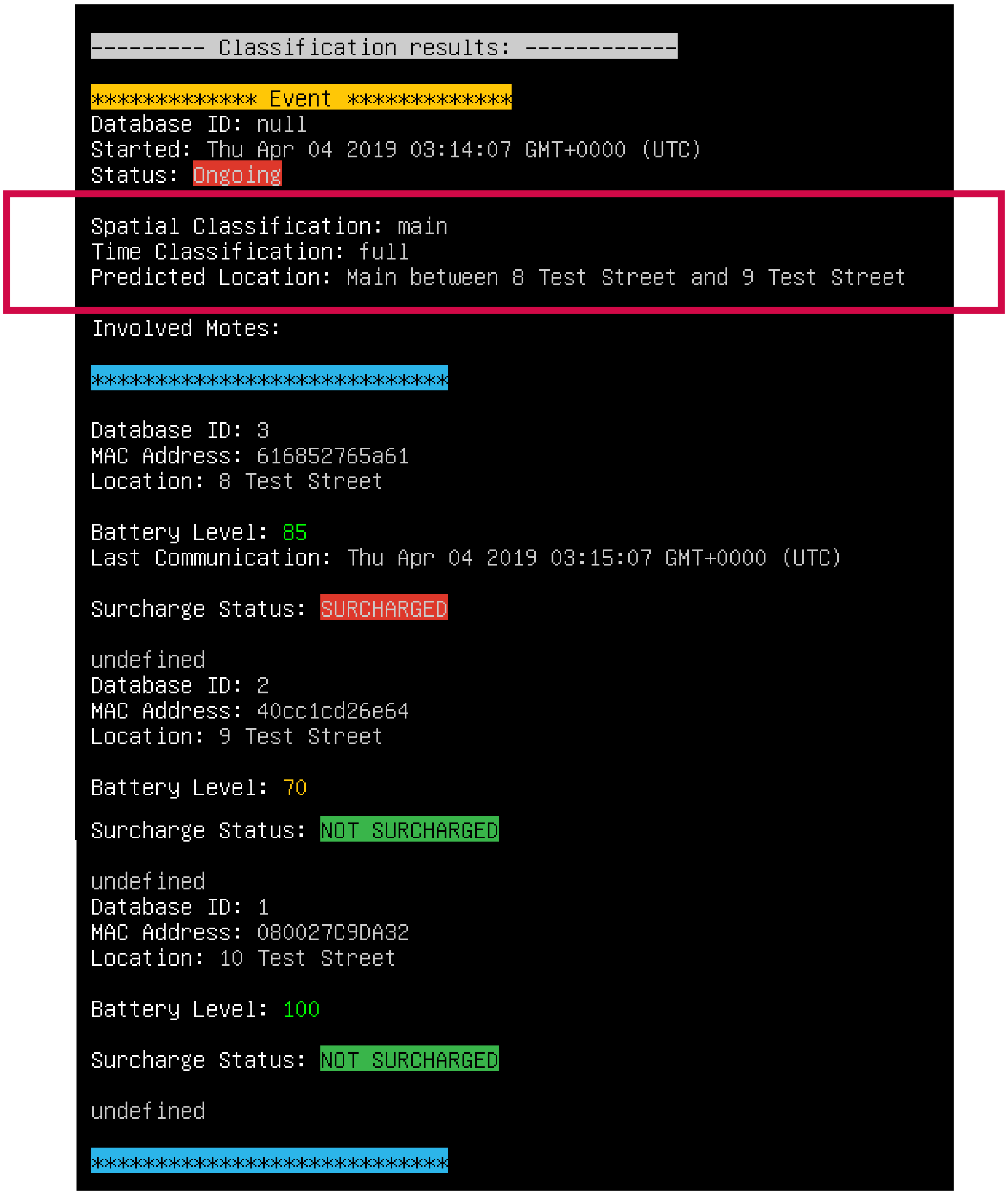
Table 4-19: Test plan for verifying classification of full main blockages.

Event Simulated:	Full main blockage between Motes 2 and 3.		
Script Used:	**full_main_blockage.js**		
Messages Sent:			
Time:	**Mote ID:**	**Battery Level:**	**Surcharge Status:**
-	2	90%	1
1:00	1	85%	1



```
ben@mqtt-sn-client:/usr/local/bin$ sudo node full_main_blockage.js
------------- Wastewater Blockage Detection System ---------------

------ Simulation: Full Main Blockage between motes 2 and 3 ------
-- Developed by Ben Buurman for Federation University Australia --
-- As deliverable for Master of Computing degree (By Research) ---

All timers successfully set. Waiting..
Successfully connected to broker at mqtt://192.168.20.19
Message '2 Thu,-04-Apr-2019-03:14:07-GMT 90 1' of topic 'surcharge' published successfully

Successfully connected to broker at mqtt://192.168.20.19
Message '3 Thu,-04-Apr-2019-03:15:07-GMT 85 1' of topic 'surcharge' published successfully

_
```

Figure 4-26: Output produced by **full_main_blockage.js** when both messages have been sent. Observing the message strings shows that messages are sent exactly one minute apart.



```
--------- Classification results: ------------

************** Event **************
Database ID: null
Started: Thu Apr 04 2019 03:14:07 GMT+0000 (UTC)
Status: Ongoing

Spatial Classification: main
Time Classification: full
Predicted Location: Main between 8 Test Street and 9 Test Street

Involved Motes:

******************************

Database ID: 3
MAC Address: 616852765a61
Location: 8 Test Street

Battery Level: 85
Last Communication: Thu Apr 04 2019 03:15:07 GMT+0000 (UTC)

Surcharge Status: SURCHARGED

undefined
Database ID: 2
MAC Address: 40cc1cd26e64
Location: 9 Test Street

Battery Level: 70

Surcharge Status: NOT SURCHARGED

undefined
Database ID: 1
MAC Address: 080027C9DA32
Location: 10 Test Street

Battery Level: 100

Surcharge Status: NOT SURCHARGED

undefined

******************************
```

Figure 4-27: Classification produced by the application server – output displaying the classification itself is highlighted.

This correctly predicts that the simulated blockage is a *main* blockage, located in the main between virtual properties 8 and 9 Test Street. Below the classification, all mote records retrieved during this process are listed. Mote 2 and 3 have been retrieved as they both reported surcharges, while Mote 1 was retrieved because it is downstream from Mote 2.

Code 4-4: full_main_blockage.js

```
const MQTT = require("mqtt");
const chalk = require("chalk");


const BROKER_ADDRESS = "";


const MOTE_2 = ["", "90"];
const MOTE_1 = ["", "85"];


//Get date and time in string format readable by application server
function currentDate()
{
    let raw = new Date();
    let utcString = raw.toUTCString();
    //Replace spaces with hyphens for processing at server. These will be
replaced with spaces again once this processing is complete.
    return utcString.replace(/ /g, "-");
}
//Publishes an MQTT message of a given topic, using the mqtt library.
function mqttPublish(topic, message)
{
    let conn = MQTT.connect(BROKER_ADDRESS);


    conn.on("connect", () =>
    {
        console.log(chalk.green("Successfully connected to broker at " +
BROKER_ADDRESS));


        conn.publish(topic, message, (err)=>
        {
            if(err)
                console.error(err);
            else
                console.log(chalk.green("Message '" + message + "' of topic '" +
topic + "' published successfully"));
```

```
            console.log("");

        });

    });

}// 'main' and entry function for firmware.

function main()

{

    console.log(chalk.bgMagenta("------------- Wastewater Blockage Detection
System ---------------"));

    console.log("");

    console.log(chalk.bgMagenta("------ Simulation: Full Main Blockage between
motes 2 and 3 ------"));

    console.log(chalk.bgMagenta("-- Developed by Ben Buurman for Federation
University Australia --"));

    console.log(chalk.bgMagenta("-- As deliverable for Master of Computing
degree (By Research) ---"));

    console.log("");


    //Send first message

    mqttPublish(getMoteString(MOTE_2, 1));


    //Wait 5 minutes (300000 milliseconds) and send second message

    setTimeout( ()=>

    {

        mqttPublish(getMoteString(MOTE_1, 1));


    }, 300000);


    //Wait 15 minutes (900000 milliseconds) and send third message

    setTimeout( ()=>

    {

        mqttPublish(getMoteString(MOTE_1, 0));


    }, 900000);


    //Wait 20 minutes (1200000 milliseconds) and send final message

    setTimeout( ()=>
    {
        mqttPublish(getMoteString(MOTE_2, 0));

    }, 1200000);

    console.log(chalk.yellow("All timers successfully set. Waiting.."));
}

//Begin processing - execute main method
main();
```

4.3.4.3 Partial Main Blockages

Having proven the application server can successfully perform both spatial classifications and classify a *full blockage*, the only remaining classification to test is the *partial blockage*. The same temporal classification algorithm is used to classify both *main* and *property connection* partial blockage, however classifying a *partial main* blockage involves more complex parameters. Consequentially, we chose to test a partial main blockage, as the success of this test will confirm the algorithm operates as expected and imply successful classification of a partial property connection blockage.

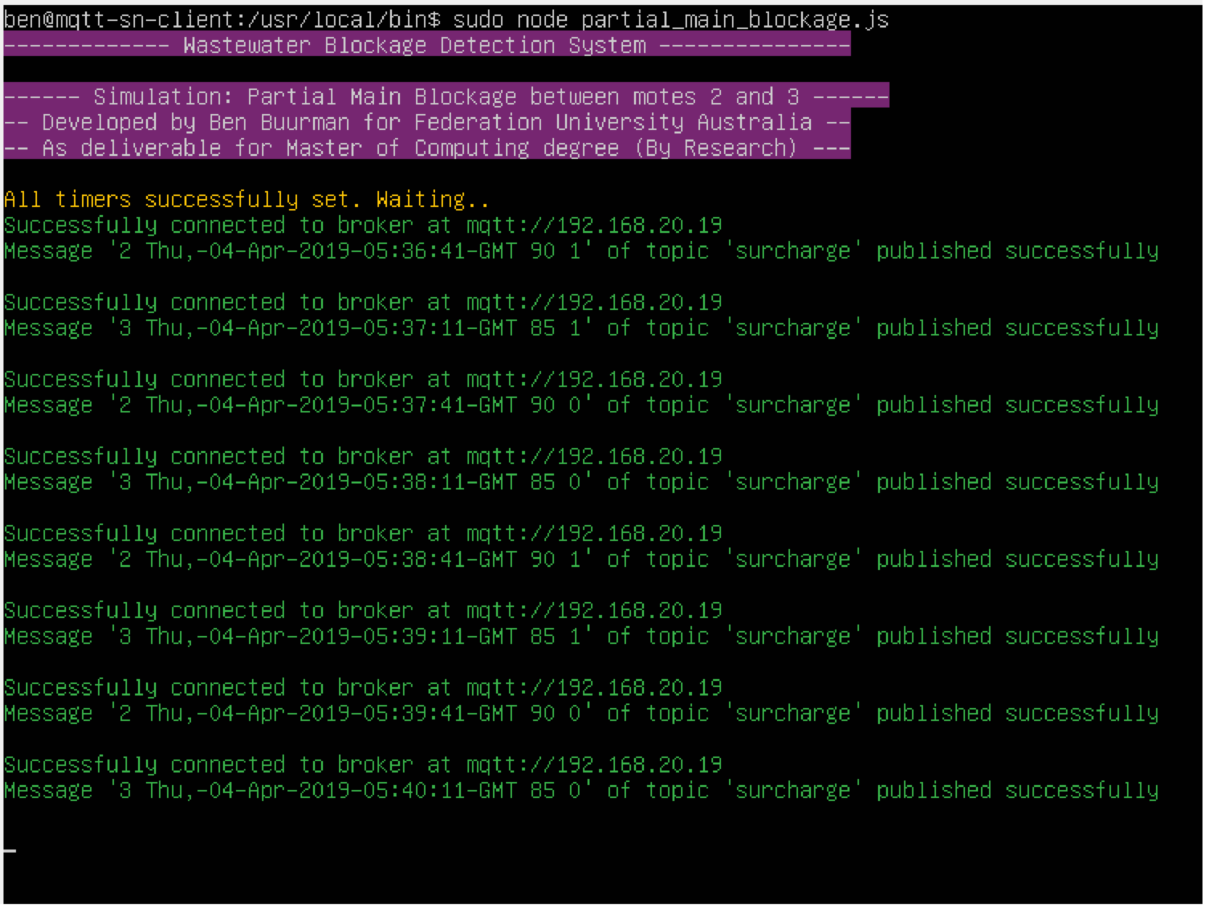
A script named *partial_main_blockage.js* simulated a partial blockage at the main between motes 2 and 3. This was achieved by sending messages that simulate a fluctuation at two neighbouring property connections; every thirty seconds, an alternate mote would send a state change. From the perspective of a single mote, this resulted in each mote sending a state change every minute. Once all timers had been executed and all messages sent, this script had produced the output shown in Figure 4-28.

Upon receiving the third-last message, the application server classified the event as shown in Figure 4.-29. This is the expected outcome – the application server estimated a partial blockage in the main between Motes 2 and 3. The *started* field shown in Figure 4-29 also matches the time the first message was sent, proving that all messages were processed as the same event. Our application server was therefore successfully able to classify partial blockages, whether occurring at the main or a property connection. We were also able to have complete confidence in our server's ability to handle the asynchronous and parallel processing required for IoT systems.

One final issue was also addressed; in Section 4.3.3.2, we discussed that the application server could not determine that the blockage was resolved. While this was not critical, we made some small changes to the code in hope of fixing this bug. Our efforts were fruitful, and after the final message was received and the appropriate time had passed, the server produced the output shown in Figure 4-30. This notifies users that the blockage is now resolved.

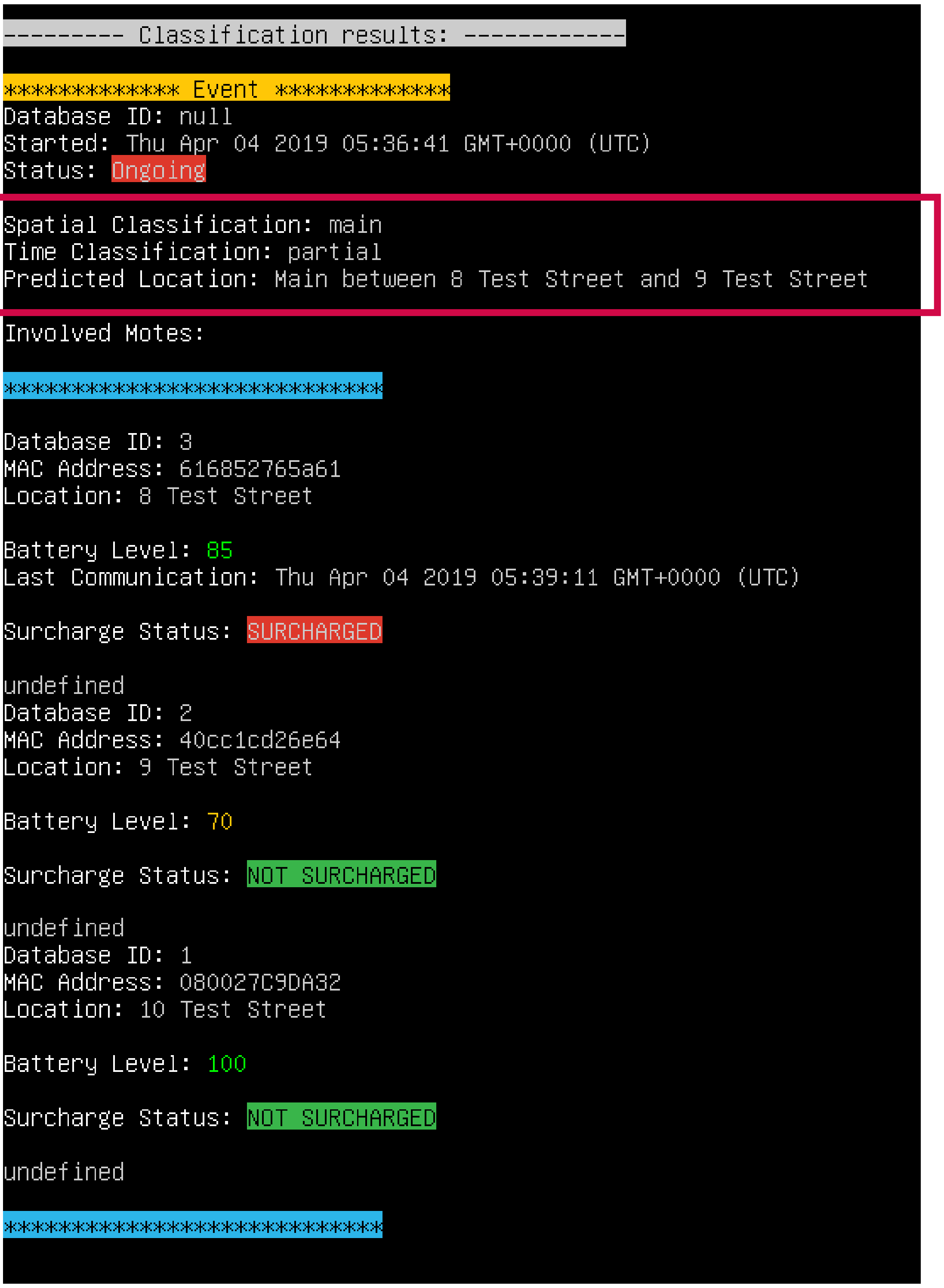
Table 4-20: Our test to verify classification of partial main blockages.

Event Simulated:	Partial main blockage between Motes 2 and 3		
Script Used:	**partial_main_blockage.js**		
Messages Sent:			
Time:	**Mote ID:**	**Battery Level:**	**Surcharge Status:**
-	2	90%	1
0:30	3	85%	1
1:00	2	90%	0
1:30	3	85%	0
2:00	2	90%	1
2:50	3	85%	1
03:00	2	90%	0
03:50	3	85%	0



```
ben@mqtt-sn-client:/usr/local/bin$ sudo node partial_main_blockage.js
------------- Wastewater Blockage Detection System ---------------

------ Simulation: Partial Main Blockage between motes 2 and 3 ------
-- Developed by Ben Buurman for Federation University Australia --
-- As deliverable for Master of Computing degree (By Research) ---

All timers successfully set. Waiting..
Successfully connected to broker at mqtt://192.168.20.19
Message '2 Thu,-04-Apr-2019-05:36:41-GMT 90 1' of topic 'surcharge' published successfully

Successfully connected to broker at mqtt://192.168.20.19
Message '3 Thu,-04-Apr-2019-05:37:11-GMT 85 1' of topic 'surcharge' published successfully

Successfully connected to broker at mqtt://192.168.20.19
Message '2 Thu,-04-Apr-2019-05:37:41-GMT 90 0' of topic 'surcharge' published successfully

Successfully connected to broker at mqtt://192.168.20.19
Message '3 Thu,-04-Apr-2019-05:38:11-GMT 85 0' of topic 'surcharge' published successfully

Successfully connected to broker at mqtt://192.168.20.19
Message '2 Thu,-04-Apr-2019-05:38:41-GMT 90 1' of topic 'surcharge' published successfully

Successfully connected to broker at mqtt://192.168.20.19
Message '3 Thu,-04-Apr-2019-05:39:11-GMT 85 1' of topic 'surcharge' published successfully

Successfully connected to broker at mqtt://192.168.20.19
Message '2 Thu,-04-Apr-2019-05:39:41-GMT 90 0' of topic 'surcharge' published successfully

Successfully connected to broker at mqtt://192.168.20.19
Message '3 Thu,-04-Apr-2019-05:40:11-GMT 85 0' of topic 'surcharge' published successfully

_
```

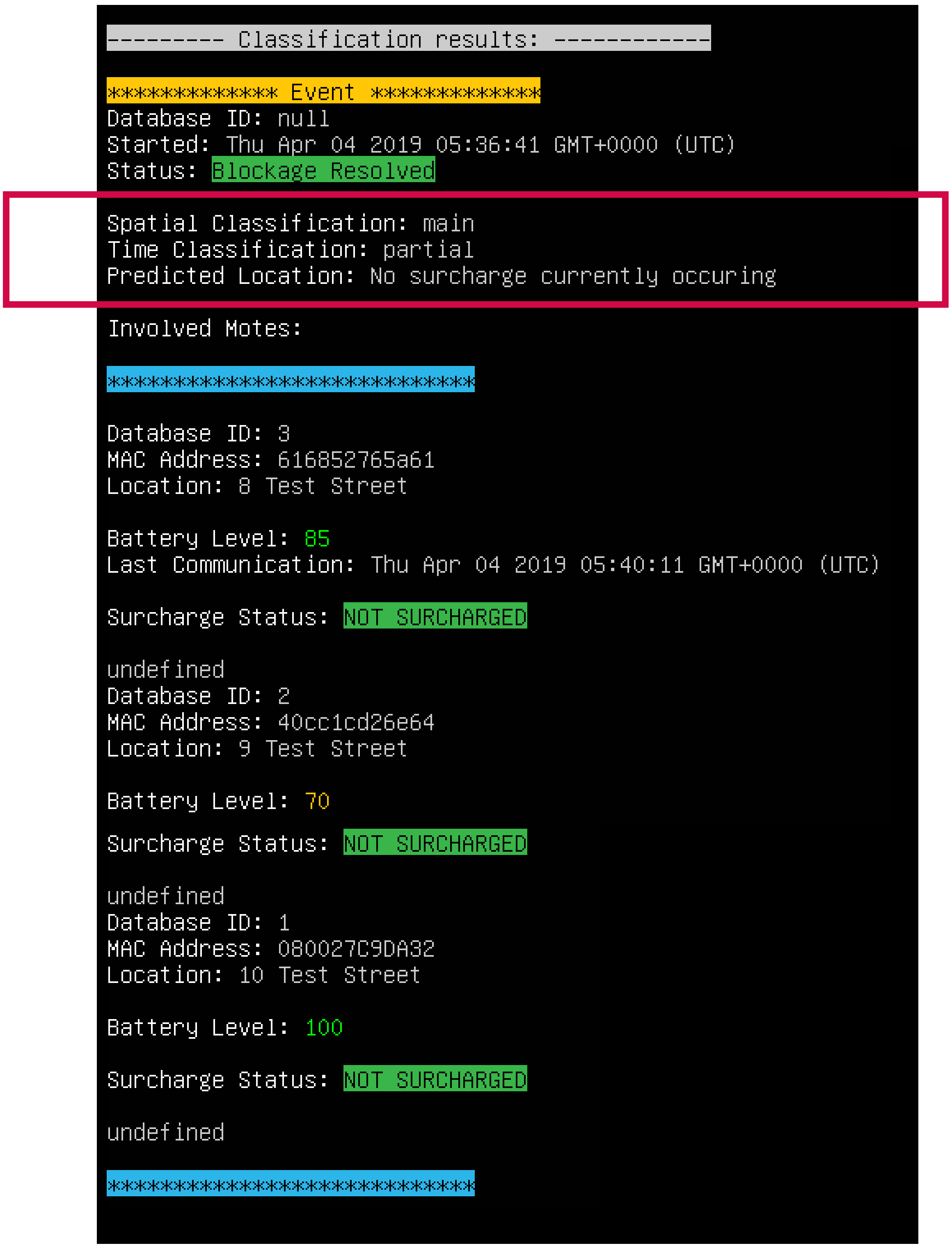
Figure 4-28: Output produced by *partial_main_blockage.js* when once all messages to simulate a partial main blockage have been sent. It can be observed that all messages were sent roughly 30 seconds apart, with each individual mote having a message sent every 60 seconds.



```
---------- Classification results: -------------

************** Event **************
Database ID: null
Started: Thu Apr 04 2019 05:36:41 GMT+0000 (UTC)
Status: Ongoing

Spatial Classification: main
Time Classification: partial
Predicted Location: Main between 8 Test Street and 9 Test Street

Involved Motes:

******************************

Database ID: 3
MAC Address: 616852765a61
Location: 8 Test Street

Battery Level: 85
Last Communication: Thu Apr 04 2019 05:39:11 GMT+0000 (UTC)

Surcharge Status: SURCHARGED

undefined
Database ID: 2
MAC Address: 40cc1cd26e64
Location: 9 Test Street

Battery Level: 70

Surcharge Status: NOT SURCHARGED

undefined
Database ID: 1
MAC Address: 080027C9DA32
Location: 10 Test Street

Battery Level: 100

Surcharge Status: NOT SURCHARGED

undefined

******************************
```

Figure 4-29: Upon receiving the third-last message shown in Figure 4-27, the server performed the classification shown - output stating the classification results is highlighted.

This shows us that the blockage is still ongoing and has been classified as a partial blockage in the main between 8 and 9 test street. The *Involved Motes* output demonstrates that Mote 3 is currently surcharging, while Motes 2 and 1 are not. Mote 1 has been retrieved as it is downstream from Mote 2.



```
--------- Classification results: ------------

************* Event *************
Database ID: null
Started: Thu Apr 04 2019 05:36:41 GMT+0000 (UTC)
Status: Blockage Resolved

Spatial Classification: main
Time Classification: partial
Predicted Location: No surcharge currently occuring

Involved Motes:

****************************

Database ID: 3
MAC Address: 616852765a61
Location: 8 Test Street

Battery Level: 85
Last Communication: Thu Apr 04 2019 05:40:11 GMT+0000 (UTC)

Surcharge Status: NOT SURCHARGED

undefined
Database ID: 2
MAC Address: 40cc1cd26e64
Location: 9 Test Street

Battery Level: 70

Surcharge Status: NOT SURCHARGED

undefined
Database ID: 1
MAC Address: 080027C9DA32
Location: 10 Test Street

Battery Level: 100

Surcharge Status: NOT SURCHARGED

undefined

****************************
```

Figure 4-30: Output produced some time after the final message in Figure 4-26 was received – lines showing the updated status are highlighted.

Note that the time listed in *Started* is still the same as the first message, meaning that this refers to the same event. These results show that the blockage is *resolved*, and all motes have stopped surcharging.

Code 4-5: partial_main_blockage.js

```
const MQTT = require("mqtt");
const chalk = require("chalk");

const BROKER_ADDRESS = "mqtt://192.168.20.19";

//Contains mote ID and simulated battery level - mote details for sending to
application server
const MOTE_1 = ["2", "90"];
const MOTE_2 = ["3", "85"];

//Get date and time in string format readable by application server
function currentDate()
{
    let raw = new Date();
    let utcString = raw.toUTCString();

    //Replace spaces with hyphens for processing at server. These will be
replaced with spaces again once this processing is complete.
    return utcString.replace(/ /g, "-");
}

//Publishes an MQTT message of a given topic, using the mqtt library.
function mqttPublish(topic, message)
{
    let conn = MQTT.connect(BROKER_ADDRESS);

    conn.on("connect", () =>
    {
        console.log(chalk.green("Successfully connected to broker at " +
BROKER_ADDRESS));

        conn.publish(topic, message, (err)=>
        {
            if(err)
```

```
                console.error(err);
            else
                console.log(chalk.green("Message '" + message + "' of topic '" +
topic + "' published successfully"));

            console.log("");
        });
    });
}



//For a specified mote and surcharge status, returns a string for sending to the
application server.
function getMoteString(moteObject, surchargeStatus)
{
    return moteObject[0] + " " + currentDate() + " " + moteObject[1] + " " +
surchargeStatus.toString();
}



// 'main' and entry function for firmware.
function main()
{
    console.log(chalk.bgMagenta("------------- Wastewater Blockage Detection
System ---------------"));
    console.log("");
    console.log(chalk.bgMagenta("------ Simulation: Partial Main Blockage
between motes 2 and 3 ------"));
    console.log(chalk.bgMagenta("-- Developed by Ben Buurman for Federation
University Australia --"));
    console.log(chalk.bgMagenta("-- As deliverable for Master of Computing
degree (By Research) ---"));
    console.log("");

    //Send first message
    mqttPublish("surcharge", getMoteString(MOTE_1, 1));

    //Send second message
```

```
setTimeout( ()=>
{
    mqttPublish("surcharge", getMoteString(MOTE_2, 1));

}, 30000);

//Send third message
setTimeout( ()=>
{
    mqttPublish("surcharge", getMoteString(MOTE_1, 0));

}, 60000);

//Send fourth message
setTimeout( ()=>
{
    mqttPublish("surcharge", getMoteString(MOTE_2, 0));

}, 90000);

//Send fifth message
setTimeout( ()=>
{
    mqttPublish("surcharge", getMoteString(MOTE_1, 1));

}, 120000);

//Send sixth message
setTimeout( ()=>
{
    mqttPublish("surcharge", getMoteString(MOTE_2, 1));

}, 150000);

//Send sixth message
```

```
    setTimeout( ()=>
    {
        mqttPublish("surcharge", getMoteString(MOTE_1, 0));

    }, 180000);

    //Send sixth message
    setTimeout( ()=>
    {
        mqttPublish("surcharge", getMoteString(MOTE_2, 0));

    }, 210000);



    console.log(chalk.yellow("All timers successfully set. Waiting.."));

}

//Begin processing - execute main method
main();
```

4.4 Conclusion

Using the design produced in Chapter 3, this chapter detailed the development of a prototype system capable of carrying out our research goals. The mote design was followed to build a mote from a Raspberry Pi Zero (RPZ) capable of connecting to a central system using Wi-Fi. Using virtual machines to host an MQTT broker and application server, we were successfully able to create the central system and its required infrastructure. However, because of limitations with both our time and available resources, we were unable to implement MQTT-SN protocol. Finally, we created a prototype application server using the corresponding design and following its algorithms.

Testing was performed to evaluate whether our design was capable of realistically carrying out our research goals. First, we tested whether the MQTT protocol was working as expected. This was conducted by testing communications between a virtual client and the broker, and then between our prototype mote and the broker. On both occasions, communications were fully operational.

Next, we tested the mote's ability to detect actual surcharges. Surcharges were simulated in a laboratory environment using a testbed that structurally emulated a sewer main and attached property connection. The mote was deployed in the simulation environment's property connection in the same way it would be deployed in a real-world inspection shaft. The testbed was filled with water to simulate a surcharge, and output from the application server showed that the mote was correctly detecting surcharges starting and stopping.

Finally, other surcharge events the application server was required to detect were simulated in a completely virtual environment. These were false alarms, full main blockages, and partial main blockages. Partial property connection blockages were not tested, as successful classification of a partial main blockage infers they can also be detected. For each simulation, the application server correctly classified the surcharge.

Test results confirmed that we have produced a design and derived prototype capable of detecting wastewater blockages across a wide area, alongside correctly classifying them, while remaining practical and inexpensive. In the next chapter we will discuss the greater implications of this and provide areas for future works.

5 Conclusion

5.1 Summary

Wastewater blockages are a relatively common occurrence resulting from normal phenomena such as improper use of home fixtures and growth of tree roots into a pipe. Blockages left unresolved will eventually completely obstruct the flow of effluent through the pipe, which subsequently leads to effluent breaching the surface through pipe shafts or fixtures. These breaches not only present significant risk of financial loss to wastewater providers, but can cause illness, disability, or even death in exposed humans. Despite this, techniques used for detection and management of blockages are mostly archaic and inefficient, involving routine inspections of each asset or response to customer complaints. Routine inspections incur significant person-hours and cost, and even then, are unlikely to detect most blockages. By the time customers make a complaint, it is often too late, and effluent has already breached. Our research originated from a simple concept - simultaneously monitoring an entire wastewater system for blockages and notifying relevant parties as soon as possible.

Current blockage detection methods function by examining an asset at each property named an *Inspection Shaft*. Blockages cause effluent levels across all assets to rise, and therefore increased levels in an inspection shaft can indicate a blockage. This increased level is known as a *surcharge*, and technicians can gather a surprisingly large amount of information from observing it. A constantly rising surcharge is likely the result of a *full* blockage, while a slowly fluctuating surcharge likely originates from a *partial blockage*. Surcharges isolated to one property connection imply the blockage is located in that same connection, while multiple affected connections imply the blockage is located at the connected main. In addition, surcharges consisting of relatively few short fluctuations are often everyday activity requiring no action from utilities providers – we refer to these as *false alarms.* Our research now had a modus operandi – monitoring surcharge levels across property connections would be theoretically capable of detecting and locating blockages. Along with this, observing the fluctuations of surcharges and their speed would allow us to *classify* blockages according to the previously mentioned types.

Considering the severe consequences of wastewater blockages and inefficiency of current detection methods, there has been surprisingly little past research into alternative and more efficient solutions. Past research made several worthy contributions to the field, however solutions proposed or delivered were impractical for real deployment. Some of these solutions were unacceptably expensive, overly complicated, or impractical due to their own limited software and hardware. For example, some solutions used sensors with very short-range wireless communications, requiring an unreasonably high number of repeaters and gateways. When previous solutions were practical, they were only concerned with monitoring very few assets for blockages. In response, we have leveraged the lessons learned by this research and more general studies into wireless networks and smart cities to produce an alternative solution.

In this thesis, we presented a detailed design for our solution capable of monitoring sewer blockages across an entire urban infrastructure while remaining inexpensive, simple, and reliable. Surcharges are detected using wireless *motes* consisting of a float switch sensor, microcontroller, power supply and two wireless transceivers. Our prototype mote cost $85.35 AUD including the Raspberry Pi Zero mote with built-in Wi-Fi transceiver, float switch, and power supply. For large scale production, this cost is expected to significantly decrease, further demonstrating the affordability and practicality of our system.

Both LoRa and Wi-Fi transceivers will be available, offering two independent transmission media for connecting to the system's backend. LoRa is useful for long-distance communication in urban areas, while the shorter-range Wi-Fi allows the system to leverage off home or public Wi-Fi. Motes will spend most of their time in a low-power *sleep* mode, however, will wake when the float switch detects a surcharge. Following surcharge detection, motes will activate the correct wireless transceiver and send a message detailing this surcharge to a *central system* using the public Internet. We have selected the MQTT-SN application-layer *publish-subscribe* protocol, a variant of the MQTT protocol commonly utilised in IoT systems, to facilitate all communications. MQTT-SN is functionally the same as MQTT, however is designed especially for sensor networks and consumes significantly less network, processing, and power resources.

The central system is a collection of backend servers accessible on the public Internet, collectively responsible for managing network communications, receiving messages from motes, and performing intelligent processing on surcharge messages. Intelligent processing first determines whether the surcharge is caused by a genuine blockage or false alarm. No further action is required for false alarms, however if the surcharge is genuine a blockage it will be classified according to the previously mentioned blockage types. This process is broken down into *spatial* (property connection or main) and *temporal* (partial or full) classifications. Following spatial classification, the central system is also capable of determining the exact property or length of sewer main between properties where the blockage is located.

Following the finalisation of our design, we developed a prototype to evaluate its performance and compliance with our research goals. This prototype consists of a mote, central server, and the network infrastructure required for them to communicate. The prototype mote was constructed using a Raspberry Pi Zero mini-computer and float switch sensor, while the prototype central system consisted of an MQTT broker and application server deployed on virtual machines. We were unable to use MQTT-SN for our prototype as no reputable code libraries were available, and there was insufficient time to develop our own. Despite this, the MQTT Broker we utilised is compatible with MQTT-SN for future works. Each of these virtual machines run the Ubuntu Server operating system, and all software was developed in the Node.js language.

Testing conducted on our prototype system showed that the prototype operated as expected and fulfilled all research goals, further proving our design is both viable and practical. The first test was performed in laboratory environment, with a physical testbed set up to emulate a property connection and attached inspection shaft. Our prototype mote was installed in the test-bed's inspection shaft, before we simulated a surcharge by filling the testbed with water. Results were confirmatory as the mote successfully detected the surcharge, before successfully transmitting a notification to the central system where it was correctly classified. This was followed by several simulation tests evaluating the central system's ability to classify and locate blockages, using a virtual model of three property connections on the same main. Results were again confirmatory, as all tests produced results as expected and demonstrated the application server and classification algorithms was operating correctly. With our design proved both practical, viable, and operational, we have developed the first wastewater blockage detection system practically capable of monitoring an entire urban infrastructure. This has huge potential for improving services worldwide, reducing environmental pollution, and even saving lives.

5.2 Future Works

Despite our successful delivery, the current research can be extended in a number of ways, some of which are discussed below.

5.2.1 Further testing

While we verified that our delivered design is capable of practical deployment, time and resource constraints prevented us from conducting further testing that could prove beneficial. We proved that the solution works but could only undertake limited testing on *how well* it works. As a result, we lack quantitative measures of performance and classification accuracy. In addition, non-functional requirements including battery life and signal performance should be tested to further measure practicality.

This additional testing can be achieved by developing several prototype motes and deploying them throughout an actual wastewater infrastructure. While this requires collaboration with a utility provider, many would be eager to contribute with our research now proven through this thesis. Deployment would last for a predetermined period of time, and during this time data would be collected to be analysed upon conclusion. Primarily, surcharge detection and blockage classification should be compared with actual surcharges and blockages that have occurred.

50% of motes deployed should utilise the LoRa protocol while the other 50% utilise home or public Wi-Fi networks. Mote programming should be updated to log each attempt at sending a message and following testing these attempts can be compared to the amount successfully received. This will not only give an overall metric for network reliability but allow researchers to compare the performance of LoRa and Wi-Fi deployment.

5.2.2 MQTT-SN Implementation

While our solution is compatible with MQTT-SN, we were unable to test this protocol during our research as no reputable software libraries were available. We conducted testing with the closely related and more resource-hungry MQTT protocol, and our success implies that any MQTT-SN implementation will also operate correctly. However, the degree of resource consumption saved between MQTT and MQTT-SN is still unknown. While MQTT-SN consumes less resources, the benefits are not often as obvious for systems with small messages, and processing at an MQTT-SN Gateway incurs additional overhead.

Future research should develop a robust and practical MQTT-SN library for Node.js or similar languages, contributing significantly to both academia and the computing industry. Once this library has been developed, it should be implemented by a subsequent version of our solution and its performance compared with standard MQTT. Observed differences in performance will not only apply to our system, but to any IoT systems with similar scale and message size. This information will prove invaluable to future research and development across many domains involving IoT.

5.2.3 Predictive Modelling

Future research should determine if any independent environmental variables such as weather, asset condition, and time have an effect on surcharge probability and blockage type. This can be done by deploying motes at a variety of real-world locations and measuring these variables each time a surcharge is classified by the application server. If these relationships exist, they can be used to build a statistical model determining the probability of classification depending on these factors. Weather, particularly rainfall, should be examined closely as stormwater entering sewer assets has been known to cause surcharges.

With a statistical model developed, it will be possible to introduce a predictive element to the system. If the environmental variables are known each time a surcharge is detected, the application server can perform a more accurate classification scheme. Furthermore, if the environmental variables are regularly detected, the application server will be able to predict surcharges and specific blockage types at different locations before they occur. History of surcharges and blockage occurrences at particular localities and pipe sections can be considered to embed local content into such a protection This could be a significant development for the industry and change blockage resolution from *reactive* to *proactive*.

Appendix A mote.js

```
//Import required modules. 'fs' and 'chalk' are in lower-case as these are de-
facto standards/expected
const GPIO = require("onoff");
const SHELL = require("child_process");
const RASPBERRY_PI = require("systeminformation");
const MQTT = require("mqtt");
const fs = require("fs");

//Import configuration parameters from config.json
let configJson = JSON.parse(fs.readFileSync("config.json", "utf-8"));
let DEBOUNCE_TIME = parseInt(configJson.debounceTime);
let BROKER_ADDRESS = configJson.gatewayAddress;
let BROKER_PORT = parseInt(configJson.gatewayPort);
let SURCHARGE_ATTEMPTS = parseInt(configJson.surchargeAttempts);
let HEARTBEAT_ATTEMPTS = parseInt(configJson.heartbeatAttempts);
let MOTE_ID = configJson.moteID;

//Commands to execute power-saving functionality
const POWER_SAVING_COMMANDS =
[
    "/usr/bin/tvservice -o",
    "echo none | sudo tee /sys/class/leds/led0/trigger",
    "echo 1 | sudo tee /sys/class/leds/led0/brightness"
];

let previousValue = 0;
let heartbeatTimer = null;


//Get date and time in string format readable by application server
function currentDate()
{
    let raw = new Date();
    let utcString = raw.toUTCString();

    //Replace spaces with hyphens for processing at server. These will be
replaced with spaces again once this processing is complete.
    return utcString.replace(/ /g, "-");
}


//Publishes an MQTT message of a given topic, using the mqtt library.
function mqttPublish(topic, message)
{
    let conn = MQTT.connect(BROKER_ADDRESS);

    conn.on("connect", () =>
    {
        console.log("Successfully connected to broker at " + BROKER_ADDRESS);

        conn.publish(topic, message, (err)=>
        {
            if(err)
                console.log(err);
            else
                console.log("Message '" + message + "' of topic '" + topic +
"' published successfully");

            console.log("");
        });
    });
}


//Declare handler method for reacting to float switch state change. Value is
current value of float switch.
let floatSwitch_onchange = function(err, value)
{
    if(value != previousValue)
    {
        let batteryLevel = RASPBERRY_PI.battery();

        batteryLevel.then((result) =>
        {
            let messageString = MOTE_ID + " " + currentDate + " " +
result.percent.toString() + " " + value.toString();

            let messageSuccess = false;
            let attemptCtr = 0;

            while(!messageSuccess && attemptCtr < SURCHARGE_ATTEMPTS)
            {
                messageSuccess = mqttPublish("surcharge", messageString);
                attemptCtr++;
            }

            //Set previousValue to current value
            previousValue = value;

            //Reset heartbeat countdown by clearing timeout and starting again
            clearInterval(heartbeatTimer);
        heartbeatTimer = setInterval(heartbeat_ontimeout, 86483647)
        });
    }
}
```

```
//Declare handler method for reacting to heartbeat timer elapsing.
let heartbeat_ontimeout = function()
{
    let messageString = MOTE_ID + " " + currentDate + " " + batteryLevel;

    let messageSuccess = false;
    let attemptCtr = 0;

    while(!messageSuccess && attemptCtr < HEARTBEAT_ATTEMPTS)
    {
        messageSuccess = mqttPublish("heartbeat", messageString);
        attemptCtr++;
    }
}


// 'main' and entry function for firmware.
function main()
{
    console.log("***** Wastewater Blockage Detection System - Mote Software
*****");
    console.log("Developed by Ben Buurman for Federation University
Australia");
    console.log("As deliverable for Master of Computing degree (By
Research)");
    console.log("");

    //Declare float switch input pin and event listener
    let floatSwitch = new GPIO(17, "in", "both", {debounceTimeout:
DEBOUNCE_TIME});
    floatSwitch.watch(floatSwitch_onchange);

    //Declare pin for activating/deactivating power-save options
    let powerSave = new GPIO(1, "in", "both");

    //Declare heartbeat timer and event listener
    heartbeatTimer = setInterval(heartbeat_ontimeout, 86483647);

    //If power-saving functionality enabled, Execute power-saving measures
asynchronously
    if(powerSave.readSync() == 1)
    for(let i = 0; i < POWER_SAVING_COMMANDS.length; i++)
        SHELL.spawn(POWER_SAVING_COMMANDS[i]);

    //Node event loop will now run indefinitely while waiting for event
listeners
    console.log("Initialisation complete. Now waiting for surcharge or
heartbeat elapse..")
    console.log("");
}

//Begin processing - execute main method
main();
```

Appendix B run_appserver.js

```
//Import required packages
let mqtt = require("mqtt");
let chalk = require("chalk");
let processSurcharge = require("surcharge_pipeline.js");
let processHeartbeat = require("heartbeat_pipeline.js");

console.log(chalk.inverse("Starting..."))

//Load global configuration
let config = new processSurcharge.ApplicationParams();
console.log("Configuration parameters loaded...");

//Create Session object required for sharing data structures across parallel
executions of each pipeline
let theSession = new processSurcharge.AppServerSession();
console.log("Session data structures created...")

//Attempt connection to MQTT Broker and MySQL Server
let mqtt_conn = mqtt.connect(config.BROKER_IP);
console.log(chalk.yellow("Attempting connection to MQTT Broker..."));


//Executes when the system establishes a connection to the MQTT broker.
//mqt__conn.on("connect", function()
mqtt_conn.on("connect", () =>
{
    console.log(chalk.green("Successfully connected to MQTT broker at " +
config.BROKER_IP));

    mqtt_conn.subscribe("surcharge", 0, function(err, granted)
    {
        if(err)
            console.log(chalk.bgRed("Could not subscribe to 'surcharge'
topic"));


        console.log(chalk.green("Subscribed to 'surcharge' topic"));
    });

    mqtt_conn.subscribe("heartbeat", 0, function(err, granted)
    {
        if(err)
            console.log(chalk.bgRed("Could not subscribe to 'heartbeat'
topic"));

        console.log(chalk.green("Subscribed to 'heartbeat' topic at broker"));
        console.log("");
    });

});


//Executes when an MQTT message is received from the broker
//mqtt_conn.on("message", function (topic, message)
mqtt_conn.on("message", (topic, message) =>
{
    //React to different topics of messages
    if(topic.toString() == "surcharge")
    {
        //Pass message received, session data structures, and configuration
parameters into Surcharge Pipeline
        let pipelineInstance = new processSurcharge.surchargePipeline(message,
theSession, config);

        console.log(chalk.inverse("New pipelineInstance begun for " +
message));
        console.log("");

        pipelineInstance.then( (classifiedEvent) =>
        {
            if(classifiedEvent.isFalseAlarm == true)
            {
                console.log(chalk.bgYellow("Surcharge is caused by false alarm
- no action is required."));
                console.log("")
            }
            else
            {
                console.log("Surcharge pipeline processed");
            }
        },
        (rejected) =>
        {
            console.log("");
            console.log(chalk.bgRed(" !!!!!!!!!!!! "));
            console.log(chalk.bgRed("Error processing surcharge at
pipeline"));
            console.log(chalk.bgRed(rejected));
            console.log("");

        });
    }
    else if (topic.toString() == "heartbeat")
    {
        //Pass message received and configuration parameters into Heartbeat
Pipeline
```

```
        processHeartbeat.heartbeatPipeline(message, config);
    }
    else if(topic.toString() == "test")
    {
        console.log(chalk.green("Test message successfully received"));
        console.log("Contents: " + message);
        console.log("");
    }
    else
    {
        console.warn(chalk.bgYellow("MQTT Message of non-standard topic
received. Please check system security and take any required measures."));
        console.warn("Message topic: " + topic);
        console.warn("Message Contents: ");
        console.warn(message);
        console.warn("");
    }

});
```

Appendix C classes.js

```
let fs = require("fs");
let chalk = require("chalk");

//Stores information, data structures, and interface/package objects required
throughout all parallel executions of Surcharge and Heartbeat pipelines
class AppServerSession
{
    constructor()
    {
        //Instantiate all Data Structures shared across pipelines
        this.activeMotes = new Array();
        this.activeEvents = new Array();
        this.resolvedEvents = new Array();
    }


    /*
        Adds a new Event to the AppServerSession's activeEvents dictionary.
Performs all needed processing - this includes adding the Mote to the Event
and incrementing the
        Event's surcharging counter if appropriate.
    */
    addEvent(theMote)
    {
        let theEvent = new WastewaterEvent(theMote.lastCommunication);
        theEvent.involvedMotes = theMote;

        if(theMote.surchargeStatus == 1)
            theEvent.surchargingMotes++;

        this.activeEvents.push(theEvent);

        if(theMote != null && theMote.id != null)
            this.activeMotes[theMote.id] = theMote;

        return theEvent;
    }


    //Searches for any active events involving a given mote. Returns the event
if a match is found, or null if no match is found.
    searchEvent(theMote)
    {
        for(let i = 0; i < this.activeEvents.length; i++)
        {
            let next = this.activeEvents[i];
            if(next.involvedMotes.getDownstreamNeighbourByID(theMote.id) !=
null)
                return next;
        }

        //If this point is reached, no results are found. Return null.
        return null;
    }
}


/*
 Represents all runtime parameters/environment variables used by this instance
of the application server. These are able to be set by system administrators
using the configuration file.
*/
class ApplicationParams
{
    //Reads parameters/configuration information from the  file at the
provided path
    constructor(filepath = "config.json")
    {
        //Declare global structures used across application server
        this.currentEvents = new Array();
        this.loadedMotes = new Array();
        this.resolvedEvents = new Array();

        //Load configuration parameters from JSON file
        let theFile = fs.readFileSync(filepath, "utf-8");
        let configJSON = JSON.parse(theFile);

        this.BROKER_IP = configJSON.brokerAddress;

        this.FALSE_ALARM_TIMER = configJSON.falseAlarm;
        this.PROPERTY_CONNECTION_TIMER = configJSON.propertyConnectionOrMain;
        this.PARTIAL_TO_FULL_TIMER = configJSON.partialOrFull;
        this.INACTIVE_TIMER = configJSON.inactive;

        this.BATTERY_LEVEL_THRESHOLD = configJSON.batteryThreshold;

        this.SQL_LOCATION =  configJSON.dbAddress;
        this.SQL_PORT = configJSON.dbPort;
        this.SQL_USER = configJSON.dbUser;
        this.SQL_PASS = configJSON.dbPassword;
        this.INITIAL_DB = configJSON.dbInitial;
    }
}
```

```
/*
 Represents any message recieved from a mote or other MQTT client with the
same message signature.
*/
class ClientMessage
{
    //Note - a battery level of -1 indicates that the battery level is
unknown.
    constructor(initID, initClientID, initTimestamp, initBatteryLevel = -1)
    {
        this.id = initID;
        this.clientID = initClientID;
        this.timestamp = initTimestamp;
        this.batteryLevel = initBatteryLevel;

        //Initialise mote value
        this.involvedMote = null;
    }


    //Prints a detailed summary of this ClientMessage and its fields to
Standard Output.
    print()
    {
        console.log(chalk.bgCyan.blue("*********ClientMessage*********"));
        console.log(chalk.bold("Mote: ") + this.clientID);
        console.log(chalk.bold("Timestamp :") + this.timestamp.toString());

        if(this.batteryLevel > 70)
            console.log(chalk.bold("Battery Level: ") +
chalk.greenBright(this.batteryLevel));
        else if (this.batteryLevel < 30)
            console.log(chalk.bold("Battery Level: ") +
chalk.redBright(this.batteryLevel));
        else
            console.log(chalk.bold("Battery Level: ") +
chalk.yellow(this.batteryLevel));

        console.log("");
    }
}


/*
 An extension of ClientMessage that represents messages representing a
surcharge state change. This stores all information that a standard
ClientMessage does, along with an additional boolean
 field for the new state.
*/
```

```
class SurchargeMessage extends ClientMessage
{
    constructor(initID, initClientID, initTimestamp, initSurchargeStatus,
initBatteryLevel = -1)
    {
        super(initID, initClientID, initTimestamp, initBatteryLevel);
        this.surchargeStatus = initSurchargeStatus;
    }


    //Prints a detailed summary of this SurchargeMessage and its fields to
Standard Output.
    print()
    {
        console.log(chalk.bgBlue.white("*********SurchargeMessage*********"));
        console.log(chalk.bold("Mote: ") + this.clientID);
        console.log(chalk.bold("Timestamp: ") + this.timestamp.toString());

        if(this.batteryLevel > 70)
            console.log(chalk.bold("Battery Level: ") +
chalk.greenBright(this.batteryLevel));
        else if (this.batteryLevel < 30)
            console.log(chalk.bold("Battery Level: ") +
chalk.redBright(this.batteryLevel));
        else
            console.log(chalk.bold("Battery Level: ") +
chalk.yellow(this.batteryLevel));


        if(this.surchargeStatus == true)
            console.log(chalk.bold("Surcharge Status: ") +
chalk.bgRed("SURCHARGED"));
        else
            console.log(chalk.bold("Surcharge Status: ") +
chalk.bgGreen.black("NOT SURCHARGED"));


        console.log("");
    }
}


/*
 Represents a mote placed in a property's inspection shaft to detect
surcharges. This holds information about the Mote, along with any currently
occuring events.
*/
class Mote
{
    constructor(initID, initPhysicalAddress, initStreetAddress,
initDownstreamID, initDownstream, initBatteryLevel, initSurchargeStatus = 0,
initLastCommunication = null)
    {
        this.id = initID;
        this.physicalAddress = initPhysicalAddress;
        this.streetAddress = initStreetAddress;

        this.batteryLevel = initBatteryLevel;
        this.lastCommunication = initLastCommunication;
        this.surchargeStatus = initSurchargeStatus;

        this.downstreamID = initDownstreamID;
        this.downstream = initDownstream;

        //Initially set the Mote's current Event to null. This references any
event currently occuring for the mote, forming a two way reference as events
also reference all motes involved.
        this.event = null;

        //In JavaScript, stacks can be represented as arrays given their
push/pop methods. Initialise as new, empty array.
        this.stateChanges = new Array();

    }


    //Prints a detailed summary of this Mote and its fields to Standard
Output.
    print()
    {
        console.log(chalk.bold("Database ID: ") + this.id);
        console.log(chalk.bold("MAC Address: ") + this.physicalAddress);
        console.log(chalk.bold("Location: ") + this.streetAddress);
        console.log("");

        if(this.batteryLevel > 70)
            console.log(chalk.bold("Battery Level: ") +
chalk.greenBright(this.batteryLevel));
        else if (this.batteryLevel < 30)
            console.log(chalk.bold("Battery Level: ") +
chalk.redBright(this.batteryLevel));
        else
            console.log(chalk.bold("Battery Level: ") +
chalk.yellow(this.batteryLevel));

        console.log(chalk.bold("Last Communication: ") +
this.lastCommunication.toString());
```

```
        console.log("");

        if(this.surchargeStatus == true)
            console.log(chalk.bold("Surcharge Status: ") +
chalk.bgRed("SURCHARGED"));

        else
            console.log(chalk.bold("Surcharge Status: ") +
chalk.bgGreen.black("NOT SURCHARGED"));

        console.log("");
    }


    //Uses a surcharge message received from this mote's physical hardware to
update the object
    updateDetails(theMsg)
    {
        this.batteryLevel = theMsg.batteryLevel;
        this.surchargeStatus = theMsg.surchargeStatus;
        this.lastCommunication = theMsg.timestamp;

        this.stateChanges.push(new StateChange(theMsg.timestamp,
theMsg.surchargeStatus));
    }


    //Returns any mote downstream with the specified ID. If none can be found,
will return null. This is the 'wrapper' for the recursive
__traverseDownstreamList method.
    getDownstreamNeighbourByID(idToGet)
    {
        if(this.id == idToGet)
            return this;
        else
            return this.__traverseDownstreamList(this, idToGet);
    }


    /*
     Recursive method that checks if a given mote (next)'s downstream
neighbour has the ID specified. If so, it will return that downstream
neighbour. If not, it will go to the next.
     If the mote has no downstream neighbour at all, it will return null as
this means the end of the 'main' has been reached.
    */
    __traverseDownstreamList(next, idToGet)
    {
        //Failure base case
        if(next.downstream == undefined || next.downstream == null)
            return null;
        //Success base case
        else if (next.downstream.id == idToGet)
            return next.downstream;
        //Continue recursion
        else
            return this.__traverseDownstreamList(next.downstream, idToGet);

    }


    /*
     Checks if this Mote and any of its downstream neighbours have
'fluctuated' - meaning that it has more than three state changes (0->1->0).
     This is a 'wrapper' for the hidden recursive method
__recursivelyCheckFluctuation.

     Returns true if a fluctuation is detected, and false if none are detected
    */
    isMoteFluctuating()
    {
        return this.__recursivelyCheckFluctuation(this);
    }


    /*
     Recursively checks if a mote or its downstream neighbours are
fluctuating. Returns true if a fluctuation is detected, and false if none are
detected.
     Fluctuation means more than three state changes (0->1->0)
    */
    __recursivelyCheckFluctuation(next)
    {
        //Failure base case - this mote is fluctuating. Return true
        if (next.stateChanges.length >= 3)
        {
            return true;
        }
        else
        {
            //If the next node has no downstream neighbours, this is the
'end'. Return false, as it has made it this far without returning true.
            if(next.downstream == null)
                return false;
            else
                return this.__recursivelyCheckFluctuation(next.downstream);
        }
    }
    /*
     Returns true if all motes have a surcharge status of false, and false if
one or motes have surcharge statuses of true. This essentially determines if
any main blockage has been resolved
     This is the 'wrapper' for the hidden recursive method
__recursivelyCheckSurcharge
    */
    isMainResolved()
    {
        return this.__recursivelyCheckSurcharge(this)
    }


    /*
     Recursive method that determines if the next mote and all its downstream
neighbours, are undergoing surcharge.
     Return false if a surcharge can be detected here or downstream, and true
otherwise.
    */
    __recursivelyCheckSurcharge(next)
    {
        //Failure base case - this mote is surcharging
        if (next.surchargeStatus == true)
        {
            return false;
        }
        else
        {
            //If the next node has no downstream neighbours, this is the
'end'. Return true, as it has made it this far without returning false.
            if(next.downstream == null)
                return true;
            else
                return this.__recursivelyCheckSurcharge(next.downstream);
        }
    }



    /*
     Searches this mote's downstream linked-list for a Mote matching
newState's unique ID. If a match is found, that mote's mutable values will be
overwritten with those of newState. The mutable
     values are batteryLevel, lastCommunication, surchargeStatus, and
stateChanges.
    */
    updateMoteState(newState)
    {
        //If this mote is the one to update, overwrite mutable values and
return true;
        if(this.id == newState.id)
        {
            this.batteryLevel = newState.batteryLevel;
            this.surchargeStatus = newState.surchargeStatus;
            this.lastCommunication = newState.lastCommunication;
            this.stateChanges.push(new StateChange(newState.lastCommunication,
newState.surchargeStatus));
        }
        else
        {
            this.__traverseDownstreamAndUpdate(this, newState);
        }
    }


    /*
     Recursive method that checks whether a mote's downstream neighbour's ID
matches the newState object's ID (newState is a mote record). If so, that
neighbour's mutable values will be
     overwritten with newState's mutable values. If not, it will go to the
next.
    */
    __traverseDownstreamAndUpdate(next, newState)
    {
        //Failure base case
        if(next.downstream == undefined || next.downstream == null)
        {
            return false;
        }
        //Success base case
        else if (next.downstream.id == newState.id)
        {
            //Update downstream mote's values
            next.downstream.batteryLevel = newState.batteryLevel;
            next.downstream.surchargeStatus = newState.surchargeStatus;
            next.downstream.lastCommunication = newState.lastCommunication;
            next.downstream.stateChanges.push(new
StateChange(newState.lastCommunication, newState.surchargeStatus));

            return true;
        }
        //Continue recursion
        else
        {
            return this.__traverseDownstreamAndUpdate(next.downstream,
newState);
```

```
        }
    }


    //Return all downstream motes in a downstream-order array.
    getDownstreamNeighboursAsArray()
    {
        return this.__buildDownstreamArray([this]);
    }


    /*
     Recursive method that builds returns an array of the current mote's
downstream neighbours. If there are no downstream neighbours, it will meet the
base case and return itself.
     The parameter 'next' represents the current state of the array -
    */
    __buildDownstreamArray(next)
    {
        //Base case - end of LinkedList / 'main' has been reached
        if (next[next.length -1].downstream == undefined || next[next.length -
1].downstream == null)
            return next;
        else
            return next.concat(this.__buildDownstreamArray([next[next.length -
1].downstream]));

    }


}


/*
 Represents any state change that occurs at a Mote. This is when a mote goes
from surcharged to not surcharged, or vice-versa.
*/
class StateChange
{
    constructor(initTime, initStatus)
    {
        this.time = initTime;
        this.status = initStatus;
    }
}


/*
```

```
 Represents any Event that occurs across the wastewater network. This includes
all types of  blockages and false alarms, and stores the mote involved.
*/
class WastewaterEvent
{
    constructor(initOccurred, initID = null)
    {
        this.id = initID;
        this.occurred = initOccurred;
        this.involvedMotes = null;
        this.surchargingMotes = 0;

        //Initialise event fields to default values. As no classification has
occured yet, these will be 'empty'.
        this.falseAlarmCount = 0;
        this.isFalseAlarm = null;
        this.spatialClassification = "";
        this.timeClassification = "";

        //Initialise status fields to default values.
        this.resolved = false;
        this.location = "";
        this.latestSurchargeStatus = null;

        //Initialise notifications sent value as new array. This will be empty
as no notifications can be sent yet.
        this.notifcationsSent = new Array();

        //Initiate values used for classification timers
        this.__falseAlarmCountdownRunning = false;
        this.__propertyConnectionTimerRunning = false;
        this.__partialToFullCountdownTimerRunning = false;
        this.__inactiveCountdownTimerRunning = false;

        //Initialise Event timers. These will be 'null' and will populate with
timers following declaration. Once finished, the timers will re-set to null.
        let falseAlarmTimer = null;
        let propertyConnectionTimer = null;
        let partialToFullTimer = null;
        let inactiveTimer = null;
    }


    //Prints a detailed summary of this Event and its fields to Standard
Output.
    print()
    {
        console.log(chalk.bgYellow.black("************* Event
*************"));
```

```
        console.log(chalk.bold("Database ID: ") + this.id);
        console.log(chalk.bold("Started: ") + this.occurred);

        if(this.resolved == true)
            console.log(chalk.bold("Status: ") + chalk.bgGreen.black("Blockage 
Resolved"));
        else
            console.log(chalk.bold("Status: ") + chalk.bgRed("Ongoing"));

        console.log("");

        if(this.falseAlarmCount == 1 || this.falseAlarmCount == 2)
        {
            console.log(chalk.bold.yellow("Event is False Alarm"));
        }
        else
        {
            console.log(chalk.bold("Spatial Classification: ") + 
this.spatialClassification);
            console.log(chalk.bold("Time Classification: ") + 
this.timeClassification);
            console.log(chalk.bold("Predicted Location: ") + this.location);
        }

        console.log("");
        console.log(chalk.bold("Involved Motes:"));
        console.log("");
        console.log(chalk.bgCyan.black("******************************"));
        console.log("");

        let motesArray = this.involvedMotes.getDownstreamNeighboursAsArray();

        for(let i = 0; i < motesArray.length; i++)
            if(motesArray[i] != null && motesArray[i] != undefined)
                console.log(motesArray[i].print());

        console.log("");
        console.log(chalk.bgCyan.black("******************************"));
        console.log("");
    }

}


exports.AppServerSession = AppServerSession;
exports.ApplicationParams = ApplicationParams;
exports.ClientMessage = ClientMessage;
exports.SurchargeMessage = SurchargeMessage;
exports.Mote = Mote;
exports.StateChange = StateChange;
exports.Event = WastewaterEvent;
```

Appendix D surcharge_pipeline.js

```
let app_server = require("data_conn.js");
let mysql = require("mysql");
let chalk = require("chalk");

surchargePipeline = function surchargePipeline(messageString, session, config)
{
    return new Promise( (resolve, reject) =>
    {
        //Declare database connection for use during this pipeline
        let db = mysql.createConnection(
            {host: config.SQL_LOCATION,
            port: config.SQL_PORT,
            user: config.SQL_USER,
            password: config.SQL_PASS,
            database: config.INITIAL_DB});

        db.connect((err) =>
        {
            if(err)
            {
                console.log("");
                console.log(chalk.bgRed.black(" COULD NOT CONNECT TO MYSQL
DATABASE "));
                console.log(err);
                console.log("");
            }

        });

        //Convert the message string into a SurchargeMessage object.
        let theMessage = __decodeSurchargeString(messageString);

        if(theMessage != null)
        {
            let moteData = __loadMoteData(theMessage, session, db);

            moteData.then((step1) =>
            {
                let alarmVeracity = __checkAlarmVeracity(step1, config);

                alarmVeracity.then((step2) =>
                {
                    if(step2.isFalseAlarm)
                    {
                        resolve(step2);
                    }
                    else
                    {
                        let spatialClassification =
__alarmSpatialClassification(step2, config);

                        spatialClassification.then((step3) =>
                        {
                            step3.location = locateBlockage(step3);
                            let temporalClassification =
__alarmTimeClassification(step3, config);

                            temporalClassification.then((step4) =>
                            {
                                session.activeEvents[step4.id] = null;
                                session.resolvedEvents[step4.id] = step4;

                                __prototypePrintResults(theMessage, step4,
db);

                                resolve(step4);
                            });

                        });
                    }

                }, (rejected) =>
                {
                    console.log(chalk.yellow("False alarm countdown is already
occuring. This does not need to be processed."));
                });
            });
        }
        else
        {
            reject(messageString);
        }
    });
}


//Converts a space-encoded string sent from a mote to a valid
SurchargeMesssage object.
function __decodeSurchargeString(stringToDecode)
{
    //Heartbeat string structure is shown in Section 3.6.3 of thesis
    let surchargeStringSplit = (stringToDecode.toString()).split(" ");
    let formattedDate = surchargeStringSplit[1].replace(/-/g, " ");

    if(surchargeStringSplit.length != 4)
    {
        console.log("Error - Malformed surcharge alert string");
        return null;
    }

    let decodedClientID = parseInt(surchargeStringSplit[0]);
    let decodedTimestamp = new Date(formattedDate);
    let decodedBatteryLevel = parseInt(surchargeStringSplit[2]);
    let decodedSurchargeStatus = (parseInt(surchargeStringSplit[3]) == 1) ?
true : false;

    let toReturn = new app_server.SurchargeMessage(null, decodedClientID,
decodedTimestamp, decodedSurchargeStatus, decodedBatteryLevel);

    return toReturn;
}



//Checks to see if the effected mote is currently represented in the active
motes dictionary.
function __loadMoteData(involvedSurchargeMessage, session, db)
{
    return new Promise( (resolve, reject) =>
    {
        let theMote = null;
        let theEvent = null;

        //This mote is in session's activeMotes dictionary
        if(involvedSurchargeMessage.clientID in session.activeMotes)
        {
            console.log("Mote has already been loaded to session's ActiveMotes
dictionary");

            theMote = session.activeMotes[involvedSurchargeMessage.clientID];
            theEvent = session.searchEvent(theMote);

            //If mote has surcharge status of 0, and this has surcharge status
of 1, increment event's surcharge counter
            if(theMote.surchargeStatus == 0 &&
involvedSurchargeMessage.surchargeStatus == 1)
                theEvent.surchargingMotes++;

            //If the mote has a surcharge status of 1, and this has a
surcharge status of 0, increment event's surcharge counter
            else if (theMote.surchargeStatus == 1 &&
involvedSurchargeMessage.surchargeStatus == 0)
                theEvent.surchagingMotes--;
            theMote.updateDetails(involvedSurchargeMessage);

            if(theEvent != null)
            {
                resolve(theEvent)
            }
            else
            {
                console.log(chalk.red("No event could be retrieved for
previously loaded mote."));
                reject(null);
            }
        }
        else
        {
            console.log("Mote has not been loaded to session's ActiveMotes
dictionary");

            //Determine if mote's upstream neighbour has been loaded into
session ActiveMotes dictionary
            for(let i = 0; i < session.activeMotes.length; i++)
            {
                let next = session.activeMotes[i];

                if((next != null && next != undefined) && next.downstreamID ==
involvedSurchargeMessage.clientID)
                {
                    theMote = next;
                    break;
                }
            }

            //Determine if mote was found in previous loop to search for
upstream neighbour. If not, theMote will still be null.
            if(theMote != null)
            {
                console.log("Mote's upstream neighbour found in dictionary.
Retrieving mote's downstream neighbour if applicable.")

                //Get downstream mote
                if(theMote.downstreamID != null && theMote.downstream == null)
                {
                    getMoteRecord(theMote.downstreamID).then((nbr) =>
                    {
                        theMote.downstream = nbr;
                        theEvent = session.addEvent(nbr); //MIGHT CHANGE BACK
                        resolve(theEvent);
                    });
                }
            }
            else
            {
                console.log("Mote or upstream neighbour not found. Retrieving 
mote details now. ");

                getMoteAndNeighbour(involvedSurchargeMessage.clientID, 
involvedSurchargeMessage, db).then((mote) =>
                {
                        //Check if downstream Mote belongs to current event
                        if(mote.downstream != null && 
session.searchEvent(mote.downstream) != null)
                        {
                            theEvent = session.searchEvent(mote.downstream);

                            console.log("Downstream mote belongs to current 
event. Merge with current Event linked list");

                            if(theEvent.involvedMotes.downstream != null)
                                mote.downstream.downstream = 
theEvent.involvedMotes.downstream;

                            theEvent.involvedMotes = mote;
                            session.activeMotes[mote.id] = mote;
                            theEvent.surchargingMotes += mote.surchargeStatus;
                        }
                        else
                        {
                            console.log("Downstream mote does not belong to 
current event");

                            theEvent = session.addEvent(mote);
                        }


                    resolve(theEvent);
                });
            }

        }
    });
}


/*
    Retrieves a Mote's record from the database, along with that of any 
downstream neighbour. Updates the mote record from any included surcharge 
message, before
```

```
    adding the downstream neighbour to the mote.
*/
function getMoteAndNeighbour(moteID, surchargeMessage, db)
{
    return new Promise ((resolve, reject) =>
    {
        getMoteRecord(moteID, db).then( (mote) =>
        {
            mote.updateDetails(surchargeMessage);

            if(mote.downstreamID != null)
            {
                getMoteRecord(mote.downstreamID, db).then((neighbour) =>
                {
                    mote.downstream = neighbour;
                    resolve(mote);
                });
            }
            else
            {
                resolve(mote);
            }
        });
    });
}



//Returns promise to retrieve a given Mote record based on ID. This is the
'inner' hidden method used by getMoteAndNeighbour().
function getMoteRecord(IDtoGet, db)
{
    return new Promise( (resolve, reject) =>
    {
        console.log("Retrieving Mote with ID " + IDtoGet);

        db.query("SELECT * FROM mote WHERE id = ?", [IDtoGet], function(error,
results, fields)
        {
            if(error)
            {
                console.log(chalk.red("Could not retrive Mote with ID " +
IDtoGet));
                console.log(error);
                console.log("");
                reject(null);
            }

            if(results.length > 0 && results[0] != null)
            {
                let involvedMote = new app_server.Mote(results[0].id,
results[0].physicalAddress, results[0].streetAddress, results[0].downstream,
null,
                    results[0].batteryLevel, 0, results[0].lastCommunication);

                console.log("Mote object with ID " + IDtoGet + " retrieved");
                resolve(involvedMote);

            }
            else
            {
                console.log(chalk.red("No results found for Mote"));
                reject(null);
            }
        });

    });
}



 //Determines if an Event is caused by a false alarm/regular daily occurrence.
function __checkAlarmVeracity(theEvent, config)
{
    return new Promise(function(resolve, reject)
    {
        console.log("");
        console.log(chalk.yellow.black("Beginning Alarm Veracity check"));

        //If it has already been proven event is not false alarm, do not run
this test.
        if(theEvent.isFalseAlarm == false)
            resolve(theEvent);

        //If false alarm countdown is NOT running on Event, and event has no
false alarm flags raised, begin false alarm countdown
        if( (theEvent.__falseAlarmCountdownRunning == false) &&
theEvent.falseAlarmCount == 0)
        {

            //Set __falseAlarmCountdownRunning to true, and set it back to
false when timer is elapsed
            theEvent.__falseAlarmCountdownRunning = true;

            //This code will execute after timer has finished running
            theEvent.falseAlarmTimer = setTimeout(() =>
            {
                theEvent.__falseAlarmCountdownRunning = false;
                if(theEvent.falseAlarmCount == 1  || theEvent.falseAlarmCount
== 2)
                    theEvent.isFalseAlarm = true;
                else
                    theEvent.isFalseAlarm = false;

                //Resolve Event. Number of false alarms will have been updated
by any other surcharge notifications.
                resolve(theEvent);

            }, 20000);

        }
        //If the false alarm countdown IS running on an event, increment false
alarm count
        else if (theEvent.__falseAlarmCountdownRunning)
        {
            theEvent.falseAlarmCount++;

            //Reject this event, as an earlier Surcharge notification already
has countdown running for the Event.
            reject(theEvent);
        }
    });
}



 //Spatially classifies an event based on where it is occuring - this produces
either a main blockage or property connection blockage.
function __alarmSpatialClassification(theEvent, config)
{
    return new Promise(function(resolve, reject)
    {
        console.log("");
        console.log(chalk.yellow.black("Beginning Spatial Classification"));

        //If the Event has multiple involved motes, it is a main blockage.
Resolve it as a main blockage.
        if(theEvent.surchargingMotes > 1)
        {
            console.log(chalk.yellow("Event has multiple motes surcharging"));

            //If the Event's propertyConnectioon timer is currently counting
down, cancel it as it is no longer needed.
            if (theEvent.__propertyConnectionTimerRunning)
            {
                clearTimeout(theEvent.propertyConnectionTimer);
                theEvent.__propertyConnectionTimerRunning = false;
            }

            console.log(chalk.green("Event is main blockage"));

            theEvent.spatialClassification = "main";
            resolve(theEvent);
        }
        else if (theEvent.surchargingMotes == 1 &&
theEvent.spatialClassification == "property connection")
        {
            //There is no need to reclassify the event, so resolve it
            resolve(theEvent);
        }
        else
        {
            console.log(chalk.yellow("Event has one or less motes
surcharging"));


             //If Event property conneciton timer is already running, another
mote surcharge occurence is 'waiting' on this Event.
            if(theEvent.__propertyConnectionTimerRunning == true)
                reject(theEvent);


            //If the event has one involved mote, it could be a property
connection blockage.
            theEvent.__propertyConnectionTimerRunning = true;

            console.log(chalk.yellow("Begin property connection timer"));

            theEvent.propertyConnectionTimer = setTimeout(function()
            {
                console.log(chalk.yellow("Property connection timer has
elapsed"));

                theEvent.__falseAlarmCountdownRunning = false;

                //Once timeout has elapsed, make sure other motes have not
surcharged for this event in meantime.
                if(theEvent.surchargingMotes > 1)
                    theEvent.spatialClassification = "main";
                else
                    theEvent.spatialClassification = "property connection";

                resolve(theEvent);
            }, 100000);
        }
    });
}


//Accepts an Event object (Representing a blockage) and returns a string
stating its street address/location
function locateBlockage(theEvent)
{
    //Determine blockage location
    if(theEvent.spatialClassification == "property connection")
    {
        return "Property Connection at " +
theEvent.involvedMotes.streetAddress;
    }
    else
    {
        //Get all motes in event
        let downstreamAsArray =
theEvent.involvedMotes.getDownstreamNeighboursAsArray();

        //Remove all non-surcharging motes from downstream as array
        let filtered = downstreamAsArray.filter(function(value, index, arr)
        {
           return (value.surchargeStatus == 1);
        });

        if(filtered.length > 0)
        {
            let lastValue = filtered[filtered.length - 1];

            if(lastValue.downstream != null && lastValue.downstream !=
undefined)
                return "Main between " + lastValue.streetAddress + " and " +
lastValue.downstream.streetAddress;

            else
                return "Main between " + lastValue.streetAddress + " and end
of main";
        }
        else
        {
            //Do not change location
            return theEvent.location;
        }

    }
```

```
}



 //Classifies an event based on whether it is caused by a full or parital
blockage (Time classification)
function __alarmTimeClassification(theEvent, config)
{
    return new Promise(function(resolve, reject)
    {
        console.log("");
        console.log(chalk.yellow.black("Beginning Time Classification"));

        //If the latest surcharge status is false
        if(theEvent.latestSurchargeStatus == false)
        {
            //Determine if all motes are 'inactive' - surcharge status of 0.
This method will return true if no motes are surcharging
            if (theEvent.involvedMotes.isMainResolved())
            {
                theEvent.__inactiveCountdownTimerRunning = true;
                //Begin inactive countdown timer

                setTimeout(function()
                {
                    theEvent.__inactiveCountdownTimerRunning = false;

                    //If all motes are still inactive
                    if(theEvent.involvedMotes.isMainResolved())
                    {
                        theEvent.resolved = true;
                        resolve(theEvent);
                    }

                }, 25000);

            }
            else
            {
                //If latest surcharge status is 0, do nothing for now.
                reject(theEvent);
            }

        }
        else //Latest surcharge status is true
        {
            //If the inactive timer was running, one or motes has fluctuated
from false to true when all were previously false. This is clearly a partial
blockage.
            if(theEvent.__inactiveCountdownTimerRunning ||
theEvent.__partialToFullCountdownTimerRunning)
            {
                theEvent.__inactiveCountdownTimerRunning = false;
                theEvent.__partialToFullCountdownTimerRunning = false;

                clearTimeout(theEvent.partialToFullTimer);
                clearTimeout(theEvent.inactiveTimer);

                theEvent.timeClassification = "partial";
                resolve(theEvent);
            }
            else
            {
                //Check if any motes have cycled between surcharged, not
surcharged, and then surcharged again. If so, this indicates a likely partial
blockage.
                if(theEvent.involvedMotes.isMoteFluctuating())
                {
                    theEvent.timeClassification = "partial";
                    theEvent.__partialToFullCountdownTimerRunning = true;

                    //Begin the partial_to_full countdown timer. If this is
elapses with no interruption, the event is a full blockage as fluctuations
have 'stopped'.
                    setTimeout(function()
                    {
                        theEvent.__partialToFullCountdownTimerRunning = false;

                        if(theEvent.involvedMotes.isMainResolved())
                        {
                            theEvent.resolved = true;
                        }
                        else
                        {
                            theEvent.timeClassification = "full";
                        }

                        resolve(theEvent);

                    }, 300000);

                }
                else //If not, this likely indicates a full blockage
                {
                    theEvent.timeClassification = "full";
                    resolve(theEvent);
                }
            }
        }
    });
}

    console.log("");
    console.log(chalk.inverse("--------- Classification results: ------------
"));
    console.log("");
    theEvent.print();
}

exports.surchargePipeline = surchargePipeline;
exports.ApplicationParams = app_server.ApplicationParams;
exports.AppServerSession = app_server.AppServerSession;
```

Bibliography

[1] D. Yagain, S. Chennapnoor and H. Yagain, “Design and implementation of high-speed, low-area switch debouncer ASIC for deep submicron technology,” in *2011 Annual IEEE India Conference*, Hyderabad, India, 2011.

[2] Z. Zhou, W. Nie, Z. Xi and X. Wang, “A High-Electrical-Reliability MEMS Inertial Switch Based on Latching Mechanism and Debounce Circuit,” *IEEE Sensors Journal,* vol. 16, no. 7, pp. 1918 - 1925, 2016.

[3] A. Nayyar and V. Puri, “A review of Arduino board's, Lilypad's & Arduino shields,” in *2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom)*, New Dehli, India, 2016.

[4] Y. A. Badamasi, “The working principle of an Arduino,” in *2014 11th International Conference on Electronics, Computer and Computation (ICECCO)*, Abuja, Nigeria, 2014.

[5] Z. Cheng, Y. Li and R. West, “Demo abstract: A multithreaded arduino system for embedded computing,” in *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, Seattle, USA, 2015.

[6] J. P. Pawar, A. S, A. S and S. K. B, “Real time energy measurement using smart meter,” in *2016 Online International Conference on Green Engineering and Technologies (IC-GET)*, Coimbatore, India, 2016.

[7] J. G. Ganssle, “A Guide to Debouncing,” 06 2008. [Online]. Available: https://pubweb.eng.utah.edu/~cs5780/debouncing.pdf. [Accessed 25 09 2018].

[8] Australian Department of Health, “Disease from sewage,” Australian Department of Health, 11 2010. [Online]. Available: http://www.health.gov.au/internet/publications/publishing.nsf/Content/ohp-enhealth-manual-atsi-cnt-l~ohp-enhealth-manual-atsi-cnt-l-ch2~ohp-enhealth-manual-atsi-cnt-l-ch2.3. [Accessed 25 09 2018].

[9] Centers for Disease Control and Prevention, “Salmonella,” Centers for Disease Control and Prevention , 12 09 2018. [Online]. Available: https://www.cdc.gov/salmonella/index.html. [Accessed 25 09 2018].

[10] B. Hibbert, C. Costiniuk, R. Hibbert, P. Joseph, H. Alanazi, T. Simard, C. Dennie, J. B. Angel and E. R. O'Brien, “Cardiovascular complications of Salmonella enteritidis infection,” *Canadian Journal of Cardiology,* vol. 26, pp. 323-325, 2010.

[11] C. P. D. MD, “Salmonella,” eMedicineHealth, 24 07 2018. [Online]. Available: https://www.emedicinehealth.com/salmonella/article_em.htm. [Accessed 25 09 2018].

[12] P. Bibby, “KFC ordered to pay $8m to brain-damaged girl,” Sydney Morning Herald, 27 04 2012. [Online]. Available: https://www.smh.com.au/national/nsw/kfc-ordered-to-pay-8m-to-brain-damaged-girl-20120427-1xpkc.html. [Accessed 25 09 2018].

[13] W. H. Organization, “Trachoma,” World Health Organization, 16 02 2018. [Online]. Available: http://www.who.int/news-room/fact-sheets/detail/trachoma. [Accessed 25 09 2018].

[14] The Fred Hollows Foundation, “Trachoma,” The Fred Hollows Foundation, [Online]. Available: https://www.hollows.org/au/eye-health/trachoma. [Accessed 25 09 2018].

[15] S. Toze, “Microbial Pathogens in Wastewater,” Commonwealth Scientific and Industrial Research Organisation, Canberra, Australia, 1997.

[16] M. B. Ali, K. Horoshenkov and S. Tait, “Rapid detection of sewer defects and blockages using acoustic-based instrumentation,” *Water Science & Technology,* vol. 64, no. 8, pp. 1700-1707, 2011.

[17] World Health Organization, “Sanitation,” World Health Organization, 19 02 2018. [Online]. Available: http://www.who.int/news-room/fact-sheets/detail/sanitation. [Accessed 25 09 2018].

[18] M. Safi, “'Manual scavenging': death toll of Indian sewer cleaners revealed,” The Guardian, 18 09 2018. [Online]. Available:

https://www.theguardian.com/world/2018/sep/19/death-toll-of-indian-sewer-cleaners-revealed-for-first-time. [Accessed 25 09 2018].

[19] S. Nair, “One manual scavenging death every five days: Official data,” Indian Express, 18 09 2018. [Online]. Available: https://indianexpress.com/article/india/official-data-shows-one-manual-scavenging-death-every-five-days-5361531/. [Accessed 25 09 2019].

[20] Environmental Protection Authority Victoria, “Stormwater,” Environmental Protection Authority Victoria, 26 06 2012. [Online]. Available: https://www.epa.vic.gov.au/your-environment/water/stormwater. [Accessed 26 09 2018].

[21] New South Wales Office of Environment and Heritage, “Stormwater,” New South Wales Office of Environment and Heritage, 04 07 2018. [Online]. Available: https://www.environment.nsw.gov.au/stormwater/. [Accessed 26 09 2018].

[22] N. C. f. G. R. a. Training, “What is groundwater?,” National Centre for Groundwater Research and Training, [Online]. Available: http://www.groundwater.com.au/pages/what-is-groundwater. [Accessed 26 09 2018].

[23] T. N. Pham, M.-F. Tsai, D. B. Nguyen, C.-R. Dow and D.-J. Deng, “A Cloud-Based Smart-Parking System Based on Internet-of-Things Technologies,” *IEEE Access,* vol. 3, pp. 1581-1591, 2015.

[24] A. R. Biswas and R. Giaffreda, “IoT and cloud convergence: Opportunities and challenges,” in *2014 IEEE World Forum on Internet of Things (WF-IoT)*, Seoul, South Korea, 2014.

[25] M. Singh, A. Singh and S. Kim, “Blockchain: A game changer for securing IoT data,” in *2018 IEEE 4th World Forum on Internet of Things (WF-IoT)*, Singapore, Singapore, 2018.

[26] W. Shi, J. Cao, Q. Zhang, Y. Li and L. Xu, “Edge Computing: Vision and Challenges,” *IEEE Internet of Things Journal,* vol. 3, no. 5, pp. 637-646, 2016.

[27] J. E. Siegel, S. Kumar and S. E. Sarma, “The Future Internet of Things: Secure, Efficient, and Model-Based,” *IEEE Things of Things Journal,* vol. 5, no. 4, pp. 2386-2398, 2018.

[28] J. A. Stankovic, “Research Directions for the Internet of Things,” *IEEE Internet of Things Journal,* vol. 1, no. 1, pp. 3-9, 2014.

[29] M. Chernyshev, Z. Baig, O. Bello and S. Zeadally, “Internet of Things (IoT): Research, Simulators, and Testbeds,” *IEEE Internet of Things Journal,* vol. 5, no. 3, pp. 1637-1647, 2018.

[30] S. Duangsuwan, A. Takam, R. Nujankaew and P. Jamjareegulgarn, “A Study of Air Pollution Smart Sensors LPWAN via NB-IoT for Thailand Smart Cities 4.0,” in *2018 10th International Conference on Knowledge and Smart Technology (KST)*, Chiang Mai, Thailand, 2018.

[31] D. Patel and M. Won, “Experimental Study on Low Power Wide Area Networks (LPWAN) for Mobile Internet of Things,” in *2017 IEEE 85th Vehicular Technology Conference (VTC Spring)*, Sydney, Australia, 2017.

[32] W. Guibene, J. Nowack, N. Chalikias, K. Fitzgibbon, M. Kelly and D. Prendergast, “Evaluation of LPWAN Technologies for Smart Cities: River Monitoring Use-Case,” in *2017 IEEE Wireless Communications and Networking Conference Workshops (WCNCW)*, San Francisco, USA, 2017.

[33] K. Mikhaylov, M. Stusek, P. Masek, V. Petrov, J. Petajajarvi, S. Andreev, J. Pokorny, J. Hosek, A. Pouttu and Y. Koucheryavy, “Multi-RAT LPWAN in Smart Cities: Trial of LoRaWAN and NB-IoT Integration,” in *2018 IEEE International Conference on Communications (ICC)*, Kansas City, USA, 2018.

[34] N. Saravanan, A. Das and V. Iyer, “Smart water grid management using LPWAN IoT technology,” in *2017 Global Internet of Things Summit (GIoTS)*, Geneva, Switzerland, 2017.

[35] I. Ganchev, Z. Ji and M. O'Droma, “A generic IoT architecture for smart cities,” in *25th IET Irish Signals & Systems Conference 2014 and 2014 China-Ireland International Conference on Information and Communications Technologies (ISSC 2014/CIICT 2014)*, Limerick, Ireland, 2013.

[36] X. Chen, J. Liu, X. Li, L. Sun and Y. Zhen, “Integration of IoT with smart grid,” in *IET International Conference on Communication Technology and Application (ICCTA 2011)*, Beijing, China, 2011.

[37] J. Jin, J. Gubbi and M. Palaniswami, “An Information Framework for Creating a Smart City Through Internet of Things,” *IEEE Internet of Things Journal ,* vol. 1, no. 2, pp. 112 - 121, 2014.

[38] O. Jo, Y.-K. Kim and J. Kim, “Internet of Things for Smart Railway: Feasibility and Applications,” *IEEE Internet of Things Journal,* vol. 5, no. 2, pp. 482-490, 2017.

[39] A. Zanella, N. Bui, A. Castellani, L. Vangelista and M. Zorzi, “Internet of Things for Smart Cities,” *IEEE Internet of Things Journal,* vol. 1, no. 1, pp. 22-32, 2014.

[40] M. Wollschlaeger, T. Sauter and J. Jasperneite, “The Future of Industrial Communication: Automation Networks in the Era of the Internet of Things and Industry 4.0,” *IEEE Industrial Electronics Magazine,* vol. 11, no. 1, pp. 17-27, 2017.

[41] M. D. V. Pena, J. J. Rodriguez-Andina and M. Manic, “The Internet of Things: The Role of Reconfigurable Platforms,” *IEEE Industrial Electronics Magazine,* vol. 11, no. 3, pp. 6-19, 2017.

[42] U. Raza, P. Kulkarni and M. Sooriyabandara, “Low Power Wide Area Networks: An Overview,” *IEEE Communications Surveys and Tutorials,* vol. 19, no. 2, pp. 855-873, 2017.

[43] K. Mekki, E. Bajic, F. Chaxel and F. Meyer, “A comparative study of LPWAN technologies for large-scale IoT deployment,” *ICT Express,* 2018.

[44] D. Vakula and Y. K. Kolli, “Waste water management for smart cities,” in *2017 International Conference on Intelligent Sustainable Systems (ICISS)*, Palladam, India, 2017.

[45] M. Sonrani, M. Abbatangelo, E. Carmona, G. Duina, M. Malgaretti, E. Comini, V. Sberveglieri, M. P. Bhandari, D. Bolpagni and G. Sberveglieri, “Array of Semiconductor

Nanowires Gas Sensor for IoT in Wastewater Management," in *2018 Workshop on Metrology for Industry 4.0 and IoT*, Brescia, Italy, 2018.

[46] N. Dlodlo, O. Gcaba and A. Smith, "Internet of things technologies in smart cities," in *2016 IST-Africa Week Conference*, Durban, South Africa, 2016.

[47] Z. Li, Q. Shi, W. Hu and Y. Li, "A sewer sensor monitoring system based on embedded system," in *2018 13th IEEE Conference on Industrial Electronics and Applications (ICIEA)*, Wuhan, China, 2018.

[48] A. Shrivastava, "Development of robotic sewerage blockage detector controlled by embedded systems," in *IEEE-International Conference On Advances In Engineering, Science And Management (ICAESM -2012)*, Nagapattinam, Tamil Nadu, India, 2012.

[49] I. Vaani, S. J. Sushil, U. V. Kunjamma, A. Ramachandran, V. T. Bai and B. Thyla, "BhrtyArtana (A pipe cleaning and inspection robot)," in *2017 Third International Conference on Sensing, Signal Processing and Security (ICSSS)*, Chennai, India, 2017.

[50] M. S. Khan, "Empirical Modeling of Acoustic Signal Attenuation in Municipal Sewer Pipes for Condition Monitoring Applications," in *2018 IEEE Green Technologies Conference (GreenTech)*, Austin, TX, USA, 2018.

[51] M. S. Khan and R. Patil, "Statistical Analysis of Acoustic Response of PVC Pipes for Crack Detection," in *SoutheastCon 2018*, St. Petersburg, FL, USA, 2018.

[52] C. See, K. Horoshenkov, S. Tait, R. Abd-Alhameed, Y. Hu, E. Elkhazmi and J. Gardiner, "A Zigbee based wireless sensor network for sewerage monitoring," in *2009 Asia Pacific Microwave Conference*, Singapore, Singapore, 2009.

[53] I. Stoianov, L. Nachman, S. Madden, T. Tokmouline and M. Csail, "PIPENET: A Wireless Sensor Network for Pipeline Monitoring," in *2007 6th International Symposium on Information Processing in Sensor Networks*, Cambridge, MA, USA, 2007.

[54] J. Yan, Z. Feng, J. Wu and J. Ma, "Research on identifying drainage pipeline blockage based on multi-feature fusion," in *2017 29th Chinese Control And Decision Conference (CCDC)*, Chongqing, China, 2017.

[55] Q. Hongrong, L. Jianzhong, Z. Guohui and L. Luexhuan, “Study of problems and corrective actions of urban drainage network,” in *2011 International Conference on Electric Technology and Civil Engineering (ICETCE)*, Lushan, China, 2011.

[56] W. Tsang, P. Carey, G. O'Connor and P. Connaughton, “Bluetooth Terminology,” Trinity College Dublin School of Computer Science and Statistics, [Online]. Available: http://ntrg.cs.tcd.ie/undergrad/4ba2.01/group3/terminology.html. [Accessed 10 10 2018].

[57] R. Fielding, “Fielding Dissertation: CHAPTER 5 Representational State Transfer (REST),” University of California, Irvine, 2000. [Online]. Available: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm. [Accessed 16 10 2018].

[58] T. K. D. P. R. K. John Schneider, “Efficient XML Interchange (EXI) Format 1.0 (Second Edition),” 11 02 2014. [Online]. Available: https://www.w3.org/TR/exi/. [Accessed 16 10 2018].

[59] H. Wajahat and H. S. Kim, “Efficient XML interchange for automated demand response in smart grid networks,” in *2014 14th International Symposium on Communications and Information Technologies (ISCIT)*, Incheon, South Korea, 2014.

[60] D. Peintner, H. Kosch and J. Heur, “Efficient XML Interchange for rich internet applications,” in *2009 IEEE International Conference on Multimedia and Expo*, New York, NY, USA, 2009.

[61] A. P. Castellani, M. Gheda, N. Bui, M. Rossi and M. Zorzi, “Web Services for the Internet of Things through CoAP and EXI,” in *2011 IEEE International Conference on Communications Workshops (ICC)*, Kyoto, Japan, 2011.

[62] Z. Shelby, K. Hartke and C. Bormann, “The Constrained Application Protocol (CoAP),” 06 2014. [Online]. Available: https://tools.ietf.org/html/rfc7252. [Accessed 06 10 2018].

[63] A. Castellani, S. Loreto, A. Rahman, T. Fossati and E. Dijk, “Best Practises for HTTP-CoAP Mapping Implementation,” Internet Engineering Task Force, 03 05 2012. [Online].

Available: https://tools.ietf.org/html/draft-castellani-core-http-mapping-02. [Accessed 17 10 2018].

[64] J. Ganssle, “A Guide to Debouncing, or, How to Debounce a Contact in Two Easy Pages,” The Ganssle Group, 03 2014. [Online]. Available: http://www.ganssle.com/debouncing.htm. [Accessed 08 11 2018].

[65] B. Huang, “Logic Levels,” Sparkfun, [Online]. Available: https://learn.sparkfun.com/tutorials/logic-levels/all. [Accessed 08 11 2018].

[66] “Defining Logical Levels: true and false (Boolean Constants),” Arduino, 2018. [Online]. Available: https://www.arduino.cc/reference/en/language/variables/constants/constants/. [Accessed 08 11 2018].

[67] J. Finnegan and S. Brown, “A Comparative Survey of LPWA Networking,” Maynooth University Department of Computer Science, Maynooth, Ireland, 2018.

[68] A. Augustin, J. Yi, T. Clausen and W. M. Townsley, “A Study of LoRa: Long Range & Low Power Networks for the Internet of Things,” *Sensors - Open Access Journal,* vol. 16, 2016.

[69] A. Lavric and V. Popa, “Internet of Things and LoRa Low-Power Wide-Area Networks: A Survey,” in *2017 International Symposium on Signals, Circuits and Systems (ISSCS)*`, Iasi, Romania, 2017.

[70] L. Gregora, L. Vojtech and M. Neruda, “Indoor Signal Propagation of LoRa Technology,” in *2016 17th International Conference on Mechatronics - Mechatronika (ME)*, Prague, Czech Republic, 2016.

[71] A. Lavric and V. Popa, “LoRa™ wide-area networks from an Internet of Things perspective,” in *2017 9th International Conference on Electronics, Computers and Artificial Intelligence (ECAI)*, Targoviste, Romania, 2017.

[72] S. Devalal and A. Karthikeyan, “LoRa Technology - An Overview,” in *2018 Second International Conference on Electronics, Communication and Aerospace Technology (ICECA)*, Coimbatore, India, 2018.

[73] L. Vangelista, “Modulation, Frequency Shift Chirp Modulation: The LoRa Modulation,” *IEEE Signal Processing Letters,* vol. 24, no. 12, pp. 1818-1821, 2017.

[74] L. Trinh, V. Bui, F. Ferrero, T. Nguyen and M. Le, “Signal propagation of LoRa technology using for smart building applications,” in *2017 IEEE Conference on Antenna Measurements & Applications (CAMA)*, Tsukuba, Japan, 2017.

[75] A. Lavric and V. Popa, “Internet of Things and LoRa™ low-power wide-area networks challenges,” in *2017 9th International Conference on Electronics, Computers and Artificial Intelligence (ECAI)*, Targoviste, Romania, 2017.

[76] U. Noreen, A. Bounceur and L. Clavier, “A study of LoRa low power and wide area network technology,” in *2017 International Conference on Advanced Technologies for Signal and Image Processing (ATSIP)*, Fez, Morocco, 2017.

[77] A. Sahadevan, D. Mathew, J. Mookathana and B. A. Jose, “An Offline Online Strategy for IoT Using MQTT,” in *2017 IEEE 4th International Conference on Cyber Security and Cloud Computing (CSCloud)*, New York, NY, USA, 2017.

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

[79] M. Loriot, A. Aljer and I. Shahrour, “Analysis of the use of LoRaWan technology in a large-scale smart city demonstrator,” in *2017 Sensors Networks Smart and Emerging Technologies (SENSET)*, Beirut, Lebanon, 2017.

[80] B. Oniga, V. Dadarlat, E. D. Poorter and A. Munteanu, “A secure LoRaWAN sensor network architecture,” in *2017 IEEE SENSORS*, Glasgow, UK, 2017.

[81] B. Oniga, V. Dadarlat, E. D. Poorter and A. Munteanu, “Analysis, design and implementation of secure LoRaWAN sensor networks,” in *2017 13th IEEE International*

Conference on Intelligent Computer Communication and Processing (ICCP), Cluj-Napoca, Romania, 2017.

[82] R. J. Cohn, R. J. Coppen, A. Banks and R. Gupta, “MQTT Version 3.1.1 - OASIS Standard,” 29 08 2014. [Online]. Available: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.pdf. [Accessed 05 12 2018].

[83] A. Stanford-Clark and H. L. Truong, “MQTT For Sensor Networks (MQTT-SN) Protocol Specification,” 14 11 2013. [Online]. Available: http://mqtt.org/new/wp-content/uploads/2009/06/MQTT-SN_spec_v1.2.pdf. [Accessed 09 12 2018].

[84] M. Masirap, M. H. Amaran, Y. M. Yussoff, R. A. Rahman and H. Hashim, “Evaluation of reliable UDP-based transport protocols for Internet of Things (IoT),” in *2016 IEEE Symposium on Computer Applications & Industrial Electronics (ISCAIE)*, Batu Feringghi, Malaysia, 2016.

[85] The Things Network, “Network Architecture,” The Things Network, 2019. [Online]. Available: https://www.thethingsnetwork.org/docs/network/architecture.html. [Accessed 13 01 2019].

[86] S. Spinsante, G. Ciattaglia, A. D. Campo, D. Perla, D. Pigini, G. Cancellieri and E. Gambi, “A LoRa enabled building automation architecture based on MQTT,” in *2017 AEIT International Annual Conference*, Cagliari, Italy, 2017.

[87] H.-K. Wu, T.-W. Hung, S.-H. Wang and J.-W. Wang, “Development of a shoe-based dementia patient tracking and rescue system,” in *2018 IEEE International Conference on Applied System Invention (ICASI)*, Chiba, Japan, 2018.

[88] J.-H. Kim, S.-H. Hong, S.-H. Yang and J.-H. Kim, “Design of Universal Broker Architecture for Edge Networking,” in *2017 International Conference on Networking, Architecture, and Storage (NAS)*, Shenzen, China, 2017.

[89] S. Penkov, A. Taneva, V. Kalkov and S. Ahmed, “Industrial network design using Low-Power Wide-Area Network,” in *2017 4th International Conference on Systems and Informatics (ICSAI)*, Hangzhou, China, 2017.

[90] Pinout, “The comprehensive GPIO Pinout guide for the Raspberry Pi.,” Pinout, [Online]. Available: The comprehensive GPIO Pinout guide for the Raspberry Pi.. [Accessed 22 01 2019].

[91] G. (. Unknown), “Raspberry Pi GPIO Tutorial: The Basics Explained,” PiMyLifeUp, 09 09 2015. [Online]. Available: https://pimylifeup.com/raspberry-pi-gpio/. [Accessed 22 01 2019].

[92] M. T. Jim Lindblom, “Raspberry gPIo,” Sparkfun, 29 10 2015. [Online]. Available: https://learn.sparkfun.com/tutorials/raspberry-gpio/all. [Accessed 22 01 2019].

[93] S. Mischie, “On teaching Raspberry Pi for undergraduate university programmes,” in *2016 12th IEEE International Symposium on Electronics and Telecommunications (ISETC)*, Timisoara, Romania, 2016.

[94] J. Geerling, “Raspberry Pi Zero - Power Consumption Comparison,” 27 11 2015. [Online]. Available: https://www.jeffgeerling.com/blogs/jeff-geerling/raspberry-pi-zero-power. [Accessed 23 01 2019].

[95] R. Light, “Mosquitto: server and client implementation of the MQTT protocol,” *The Journal of Open Source Software,* vol. 2, no. 13, 2017.

[96] Eclipse Foundation, “Eclipse Paho - MQTT-SN Transparent Gateway,” Eclipse Foundation, [Online]. Available: https://www.eclipse.org/paho/components/mqtt-sn-transparent-gateway/. [Accessed 23 01 2019].

[97] T. Yamaguchi, “MQTT-SN Transparent / Aggrigating Gateway,” GitHub Inc, [Online]. Available: https://github.com/eclipse/paho.mqtt-sn.embedded-c/tree/master/MQTTSNGateway. [Accessed 23 01 2019].

[98] Eclipse Foundation, “Paho,” Eclipse Foundation, [Online]. Available: https://www.eclipse.org/paho/. [Accessed 23 01 2019].

[99] R. K. Kodali and S. Soratkal, “MQTT based home automation system using ESP8266,” in *2016 IEEE Region 10 Humanitarian Technology Conference (R10-HTC)*, Agra, India, 2016.

[100] R. K. Kodali and V. S. K. Gorantla, “Weather tracking system using MQTT and SQLite,” in *2017 3rd International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT)*, Tumkur, India, 2018.

[101] A. Panwar, A. Singh, R. Kumawat, S. Jaidka and K. Garg, “Eyrie smart home automation using Internet of Things,” in *2017 Computing Conference*, London, UK, 2017.

[102] J. Zambada, R. Quintero, R. Isijara, R. Galeana and L. Santillan, “n IoT based scholar bus monitoring system,” in *2015 IEEE First International Smart Cities Conference (ISC2)*, Guadalajara, Mexico, 2015.

[103] Y. Upadhyay, A. Borole and D. Dileepan, “MQTT based secured home automation system,” in *2016 Symposium on Colossal Data Analysis and Networking (CDAN)*, Indore, India, 2016.

[104] L. P. Chitra and R. Satapathy, “Performance comparison and evaluation of Node.js and traditional web server (IIS),” in *2017 International Conference on Algorithms, Methodology, Models and Applications in Emerging Technologies (ICAMMAET)*, Chennai, India, 2017.

[105] A. J. Poulter, S. J. Johnston and S. J. Cox, “Using the MEAN stack to implement a RESTful service for an Internet of Things application,” in *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*, Milan, Italy, 2015.

[106] M. Sutiono, H. Nugroho and K. Karyono, “ApplianceHub: A wireless communication system for smart devices (case study: Smart Rice Cooker),” in *2016 International Conference on Radar, Antenna, Microwave, Electronics, and Telecommunications (ICRAMET)*, Jakarta, Indonesia, 2016.

[107] M. Moness, A. M. Moustafa, A.-R. H. Muhammad and A.-S. A. Younis, “Hybrid controller for a software-defined architecture of industrial internet lab-scale process,” in *2017 12th International Conference on Computer Engineering and Systems (ICCES)*, Cairo, Egypt, 2017.

[108] O. Chieochan, A. Saokaew and E. Boonchieng, “IOT for smart farm: A case study of the Lingzhi mushroom farm at Maejo University,” in *2017 14th International Joint*

Conference on Computer Science and Software Engineering (JCSSE), Nakhon Si Thammarat, Thailand, 2017.

[109] J. Geerling, “Raspberry Pi Zero - Conserve power and reduce draw to 80mA,” Jeff Geerling, 29 11 2015. [Online]. Available: https://www.jeffgeerling.com/blogs/jeff-geerling/raspberry-pi-zero-conserve-energy. [Accessed 24 01 2019].

[110] L. Fried, “A Tour of the Pi Zero,” Adafruit Industries, 06 08 2017. [Online]. Available: https://learn.adafruit.com/introducing-the-raspberry-pi-zero/a-tour-of-the-pi-zero. [Accessed 07 02 2019].

[111] L. Fried, “Introducing the Raspberry Pi Zero - Video Outputs,” Adafruit Industries, 26 11 2015. [Online]. Available: https://learn.adafruit.com/introducing-the-raspberry-pi-zero/video-outputs. [Accessed 07 02 2018].

[112] NPM Enterprise, “MQTT.js,” NPM Enterprise, [Online]. Available: https://www.npmjs.com/package/mqtt. [Accessed 2019 02 27].

[113] NPM Enterprise, “onoff.js,” NPM Enterprise, 05 03 2019. [Online]. Available: https://www.npmjs.com/package/onoff. [Accessed 08 03 2019].

[114] M. F. G. M. '. (. Nicholas Humfrey, “MQTT-SN Tools,” GitHub, 16 06 2018. [Online]. Available: https://github.com/njh/mqtt-sn-tools. [Accessed 08 03 2019].

[115] J. P. Talusan, “Trying MQTT-SN on Raspberry Pi,” 08 02 2018. [Online]. Available: https://jpinjpblog.wordpress.com/2018/02/08/trying-mqtt-sn-on-raspberry-pi/. [Accessed 08 03 2019].

[116] M. Foksa, “rsmb MQTT and MQTT-SN Broker,” GitHub Inc, 04 12 2015. [Online]. Available: https://github.com/MichalFoksa/rsmb. [Accessed 09 03 2019].

[117] N. Humfrey, R. Light and I. Craggs, “Mosquitto RSMB (Really Small Message Broker),” GitHub, 26 06 2017. [Online]. Available: https://github.com/eclipse/mosquitto.rsmb. [Accessed 09 03 2019].

[118] S. S. Josh Junon, “NPM - Chalk,” NPM Enterprise, 01 2019. [Online]. Available: https://www.npmjs.com/package/chalk. [Accessed 11 03 2019].

[119] H. Arasteh, V. Hosseinnezhad, V. Loia, A. Tommaseti, O. Troisi, M. Shafie-Khah and P. Siano, “Iot-based smart cities: A survey,” in *2016 IEEE 16th International Conference on Environment and Electrical Engineering (EEEIC)*, Florence, Italy, 2016.

[120] K. Avijit and D. R. Chinnaiyan, “IOT for Smart Cities,” *International Journal of Scientific Research in Computer Science, Engineering and Information Technology ,* vol. 3, no. 4, pp. 1126-1139, 2018.

[121] S. P. Mohanty, U. Choppali and E. Kougianos, “Everything you wanted to know about smart cities: The Internet of things is the backbone,” *IEEE Consumer Electronics Magazine,* vol. 5, no. 3, pp. 60-70, 2016.

[122] M. B. Rodger Lea, “Smart Cities: an IoT-centric Approach,” in *Proceedings of the 2014 International Workshop on Web Intelligence and Smart Sensing*, Saint Etienne, France, 2014.

[123] H. A. Patawala, P. B. Navnath, P. B. Yogesh and P. S. Z. Ashwini, “IOT Based Water Management System for Smart City,” *International Journal of Advance Research, Ideas, and Innovations in Technology,* vol. 3, no. 2, pp. 379-383, 2017.

[124] M. H. Miraz, M. Ali, P. S. Excell and R. Picking, “A Review on Internet of Things (loT), Internet of Everything and Internet of Nano Things (IoNT),” in *2015 Internet Technologies and Applications (ITA)*, Wrexham, UK, 2015.

[125] M. Nagakannan, C. J. Inbaraj, K. M. Kannan and S. Ramkumar, “A RECENT REVIEW ON IOT BASED TECHNIQUES AND APPLICATIONS,” in *2018 2nd International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC)I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC), 2018 2nd International Conference on*, Palladam, India, 2018.

[126] S. Al-Sarawi, M. Anbar, K. Alieyan and M. Alzubaidi, “Internet of Things (IoT) communication protocols: Review,” in *2017 8th International Conference on Information Technology (ICIT)*, Amman, Jordan, 2017.

[127] K. Routh and T. Pal, “A survey on technological, business and societal aspects of Internet of Things by Q3, 2017,” in *2018 3rd International Conference On Internet of Things: Smart Innovation and Usages (IoT-SIU)*, Bhimtal, India, 2018.

[128] D. E. Boyle, D. C. Yates and E. M. Yeatman, “Urban Sensor Data Streams: London 2013,” *IEEE Internet Computing,* vol. 17, no. 6, pp. 12-20, 2013.

[129] F. Zhou and Q. Li, “Parking Guidance System Based on ZigBee and Geomagnetic Sensor Technology,” in *2014 13th International Symposium on Distributed Computing and Applications to Business, Engineering and Science*, Xian Ning, China, 2014.

[130] East Gippsland Water Corporation, “East Gippsland Water - About Us,” Scribblevision, 04 2019. [Online]. Available: https://www.egwater.vic.gov.au/about-us/. [Accessed 13 04 2019].

[131] Gecko Gear Australia Pty Ltd, “Gecko Tradie Tough Portable Power Pack 2200 mAh - Black/Grey,” Gecko Gear Australia Pty Ltd, 2019. [Online]. Available: http://geckogear.com.au/gecko-tradie-tough-powerup-2200-mah-black-grey-gg900053/. [Accessed 28 03 2019].