

**Honours Project (CCIS) Final Report**

**2018-2019**

**Submitted for the Degree of:**

**BSc Computing**

**Project Title: Using MQTT to provide reliable messaging for a mobile Point of Sale application in unreliable network environments.**

**Name:**

**Programme : BSc (Hons) Computing**

**Matriculation Number:**

**Project Supervisor:**

**Second Marker:**

**Word count : 10,986**

**“Except where explicitly stated, all work in this report, including the appendices, is my own original work and has not been submitted elsewhere in fulfilment of the requirement of this or any other award”**

**Signed by Student:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Date: \_\_\_\_\_\_\_\_\_\_\_\_**

Abstract

Trends in retail industries currently suggest that mobile Point of Sale systems are becoming increasingly popular, usually in the form of an application installed on a smartphone or tablet device. These mobile applications are fundamentally responsible for generating transactions between a customer and a business in a physical location and therefor revenue for the business. As a result, it is of crucial importance that they can communicate with any necessary resources such as remote servers to complete transactions. This communication can prove difficult as wireless networks are by their nature often less reliable than wired equivalents. Systems are now required to ensure that transactions created by a mobile Point of Sale application reach the server responsible for processing that transaction, regardless of the reliability of the network.

The objective of this project is to establish whether Message Queue Telemetry Transport (MQTT) can be used to ensure reliable messaging between a mobile Point of Sale application and a receiving server in an unreliable network environment, so that the transaction may be processed appropriately.

This objective was completed by designing and building a mobile Point of Sale application typical of that which exists in the retail industry, which was required to connect to a server to complete transactions. MQTT was deployed as a means of sending the transaction data to the server and its ability to ensure that the data arrives at the server in a simulated unreliable network environment was tested.

The development and subsequent testing of this prototype found MQTT to be a successful candidate for providing reliable messaging of transaction data from a mobile application to a server but may not be entirely suited to this use case and would be better deployed in support of a broader technology stack to aid in reliable messaging rather than to provide it in its entirety.

Acknowledgements

I’d like to acknowledge Glasgow Caledonian University for providing me with the skills required to fully realise this project. My sincere thanks go to for both inspiring and supervising this project; without his support and insight it could not have been completed.

Lastly but by no means least, I thank my family and friends for their continual encouragement and for putting up with me at various stages of stress and frustration throughout! All told it has been a pleasure to complete this project and I hope it finds some use somewhere.

Table of Contents

[1 Introduction 6](#_Toc6395221)

[1.1 Background 6](#_Toc6395222)

[1.2 Aims and Methodology 7](#_Toc6395223)

[1.3 Structure of the rest of the report 8](#_Toc6395224)

[2 Literature Review and Technology Assessment 9](#_Toc6395225)

[2.1 Mobile PoS 9](#_Toc6395226)

[2.2 Message Oriented Middleware 10](#_Toc6395227)

[2.3 MQ Telemetry Transport 11](#_Toc6395228)

[3 Execution 14](#_Toc6395229)

[3.1 Analysis 14](#_Toc6395230)

[3.1.1 The scenario 14](#_Toc6395231)

[3.1.2 Lifecycle 14](#_Toc6395232)

[3.1.3 Requirements 15](#_Toc6395233)

[3.2 Design 17](#_Toc6395234)

[3.2.1 Context Diagram 17](#_Toc6395235)

[3.2.2 Class Diagram 17](#_Toc6395236)

[3.3 Implementation 18](#_Toc6395237)

[3.3.1 Components 18](#_Toc6395238)

[3.3.2 Technologies and Implementation 19](#_Toc6395239)

[3.3.3 System Architecture 22](#_Toc6395240)

[3.3.4 Developer Tools 22](#_Toc6395241)

[3.4 Testing 24](#_Toc6395242)

[3.4.1 Testing methodology 24](#_Toc6395243)

[3.4.2 QoS 0 Testing 25](#_Toc6395244)

[3.4.3 QoS 1 Testing 25](#_Toc6395245)

[3.4.4 QoS 2 Testing 25](#_Toc6395246)

[4 Evaluation and Discussion 26](#_Toc6395247)

[4.1 QoS 0 26](#_Toc6395248)

[4.2 QoS 1 27](#_Toc6395249)

[4.3 QoS 2 27](#_Toc6395250)

[5 Conclusions 29](#_Toc6395251)

[5.1 Overview 29](#_Toc6395252)

[5.2 Overall Conclusions 29](#_Toc6395253)

[5.2.1 MQTT 29](#_Toc6395254)

[5.2.2 The TEmPoS Server 31](#_Toc6395255)

[5.2.3 The TEmPoSmanager Application 31](#_Toc6395256)

[5.2.4 The TEmPoSmobile Application 31](#_Toc6395257)

[5.3 Further Work 31](#_Toc6395258)

[5.3.1 MQTT Security 32](#_Toc6395259)

[5.3.2 MQTT Data Transfer 32](#_Toc6395260)

[5.4 Final Thoughts 32](#_Toc6395261)

[6 Bibliography 33](#_Toc6395262)

[7 Appendices 36](#_Toc6395263)

[7.1 Agile Methodology 36](#_Toc6395264)

[7.2 Non-Functional Requirements 43](#_Toc6395265)

[7.3 Context Diagram 43](#_Toc6395266)

[7.4 Class Diagram 44](#_Toc6395267)

[7.4.1 Key Abstraction Form 45](#_Toc6395268)

[7.5 Implementation 46](#_Toc6395269)

[7.5.1 TEmPoS Point of Sale Application – (Server) 46](#_Toc6395270)

[7.5.2 TEmPoSmanager Management Application – (Desktop Application) 47](#_Toc6395271)

[7.5.3 TEmPoSmobile Point of Sale – (Client mobile application) 48](#_Toc6395272)

[7.5.4 MQTT Broker 50](#_Toc6395273)

[7.6 Testing 52](#_Toc6395274)

[7.6.1 QoS 00 Test 001 52](#_Toc6395275)

[7.6.2 QoS 00 Test 002 53](#_Toc6395276)

[7.6.3 QoS 00 Test 003 54](#_Toc6395277)

[7.6.4 QoS 01 Test 001 55](#_Toc6395278)

[7.6.5 QoS 01 Test 002 56](#_Toc6395279)

[7.6.6 QoS 01 Test 003 57](#_Toc6395280)

[7.6.7 QoS 02 Test 001 58](#_Toc6395281)

[7.6.8 QoS 02 Test 002 59](#_Toc6395282)

[7.6.9 QoS 02 Test 003 60](#_Toc6395283)

[7.6.10 QoS Testing Code Snippets 61](#_Toc6395284)

Table of Figures:

[Figure 1: Typical PoS Architecture (infodartglobal.com, 2019) 8](file:///C:\Users\LiamN\Dropbox%20(Personal)\Honours%20Project\Documentation\Final%20Report\Final%20Report.odt#_Toc6322469)

[Figure 2: Communication between Applications through Middleware (Oracle, 2010) 9](file:///C:\Users\LiamN\Dropbox%20(Personal)\Honours%20Project\Documentation\Final%20Report\Final%20Report.odt#_Toc6322470)

[Figure 3: Example of a Message queue (Amazon, 2019) 10](#_Toc6322471)

[Figure 4: MQTT as a means of connecting devices to AWS (Michael Garcia, 2016) 11](file:///C:\Users\LiamN\Dropbox%20(Personal)\Honours%20Project\Documentation\Final%20Report\Final%20Report.odt#_Toc6322472)

[Figure 5: Agile Development Cycle (Nahian-Al-Hossain, 2016) 13](file:///C:\Users\LiamN\Dropbox%20(Personal)\Honours%20Project\Documentation\Final%20Report\Final%20Report.odt#_Toc6322473)

[Figure 6: Use Case Diagram for Salesperson 14](#_Toc6322474)

[Figure 7: Use Case Diagram for Store Manager 15](#_Toc6322475)

[Figure 8: Context Diagram 16](#_Toc6322476)

[Figure 9: Class Diagram Snippet 17](#_Toc6322477)

[Figure 10: Github Commits for MQTT in React Native 20](#_Toc6322478)

[Figure 11: Architectural Design 21](#_Toc6322479)

[Figure 12: MQTT Publish QoS Code Snippet 22](#_Toc6322480)

[Figure 13: Wireshark Log for QoS 0 23](#_Toc6322481)

[Figure 14: Wireshark Log for QoS 1 24](#_Toc6322482)

[Figure 15: Wireshark Log for QoS 2 25](#_Toc6322483)

[Figure 16: QoS 2 Four-Way Handshake (HiveMQ Team, 2015) 25](#_Toc6322484)

[Figure 17: QoS Downgrading 27](#_Toc6322485)

Additional Resources:

|  |  |
| --- | --- |
| TEmPoS Server |  |
| TEmPoSmanager |  |
| TEmPoSmobile |  |
| Testing Documentation |  |
| Full Code Listings |  |

# Introduction

This introductory section will outline the background of this project by providing some context of the problem itself, the aims of the project and the methodologies employed to achieve those aims as well as some notes on the structure of this report.

## Background

Point of Sale (PoS), sometimes also referred to as Point of Purchase is, as the name suggests, the point at which a transaction takes place. Traditionally this takes the form of some sort of Till or Cash Register attached to a device - often a PC running software which provides access to other elements of the business such as a Customer Relationship Management System or Stock Management System. This process allows for electronic data capture which offers cost and productivity benefits to retailers (Connelly, 1988). This system of having a location or set of locations within a business premises for conducting transactions has for decades been the standard, usually in the form of a checkout. However, with the massive growth and expansion of the mobile and wireless technology sectors, the technology behind static Point of Sale terminals is following suit and going increasingly mobile with the mPoS market expected to grow to an astonishing 52 million units deployed by 2018 (Guerin Jacques, 2014).

Mobile-enabled technology is seen to be a catalyst for improved quality of services as well as offering cost effectiveness for businesses and better user experiences for customers (Guerin Jacques, 2014). Mobility has led to the emergence of new business models, driving innovation in the way transactions happen (Reportlinker.com, 2017). Mobile PoS is gradually being adopted as a replacement for the more traditional point of sale systems widely in use and has a growing potential for expansion into other industries with similar requirements for mobility in their transactions; including restaurants, hospitality, healthcare, retail, warehousing and distribution, entertainment, transportation, government, and consumer services and products (Anon., 2017b). As a result, research carried out in the context of one of these scenarios could be relevant and applicable to many industries.

The development and deployment of mobile devices has brought about a requirement for robust and reliable means of communication between devices. A client-server architecture, for example, will fundamentally require a reliable connection between the client device and the server. The benefit of a mobile device is that they can physically move their location, which very quickly leads to the problem of maintaining their connection with the server. If the primary goal of a business is to complete transactions, then the functionality of their PoS systems are of critical importance, and therefore it stands to reason that a mobile Point of Sale device must be able to maintain a channel of communication with a server regardless of the reliability of the network.

A potential answer to this problem is Message-Oriented Middleware. Message Oriented Middleware or MoM has increasingly been used to provide communication between devices in a number of different industries, “now that we all agree Web services must converse in a loosely coupled and asynchronous manner, it's clear that Message-Oriented Middleware has a vital role to play” (Udell, 2002). Much like mPoS, middleware has a huge range of applications in an expanding number of industries. Middleware as a term defines a software layer that acts as a link between systems, usually between some sort of database and an application. Middleware has proven itself useful in creating bridges between newer software designs and legacy hardware that they may depend on, but more recently has been implemented more so as a means of delivering the features of an application.

The aim of this project is to establish whether the MQTT protocol can be successfully deployed within a mobile Point of Sale application as a Middleware Layer and the degree to which it successfully facilitates the transfer of messages between a client-side application and a receiving server.

The key design factor to be considered is implementing the reliable sending and receiving of messages in an asynchronous fashion using MQTT. Mobile applications often make it difficult to deliver messages reliably (Choi *et al.*, 2006), hence the need for a robust middleware layer, particularly in a PoS scenario where the business relies on the data gathered at these mobile applications. A potential issue arises here in that there is a lack of standardisation for communication between such devices which can lead to interoperability problems(Anon., 2017a). This will need to be solved by appropriate selection of both hardware devices and software languages and libraries to support communication between the devices. Furthermore, it must be ensured that data captured by the PoS device only be transmitted once to avoid the duplication of transaction data which would be a failure in business logic. This will invoke the use of a feature of MQTT which allows for the specification of a Quality of Service attribute which controls how many times a message will be delivered. The testing methodology used will also need to enforce standards for what constitutes as an unreliable or occasionally-connected network environment which will be the intended market for this solution.

## Aims and Methodology

***Using MQTT to provide reliable messaging for a mobile Point of Sale application in unreliable network environments.***

The objective of this project is to create a functional prototype that successfully deploys MQTT within a mobile PoS application which will reliably allow for the sending of transaction information to a receiving server in an unreliable network environment where MQTT ensures that the information is persisted until the network will allow for it to be sent to the server.

Through the use of a Literature and Technology review, the following objectives were achieved:

* [S01] Investigate mobile Point of Sale Applications

Investigate existing examples of commonly used architectural systems and designs for mobile Point of Sale applications to ascertain their functional requirements and later help evaluate how an MQTT middleware layer might be applied. Understanding the typical environment in which these systems would be deployed to determine the usefulness of this solution.

* [S02] Examine the use of Middleware Layers for Reliable Asynchronous Messaging

Message Oriented Middleware layers are already being used to provide communication between devices and services and the ways in which other technologies like MQTT have been deployed for this purpose and more specifically the architectural significance of the Middleware layer.

* [S03] Understanding MQTT

Developing a solid understanding of what MQTT is capable of, why it was developed and for which purposes will greatly inform whether it is a good candidate for this topic, as well as influencing how it might be deployed in the prototype solution.

The Primary phase of the project led to the fulfilment of the following objectives:

* [P01] Develop a Test Plan

A key element of this project is the integrity of its testing. Given that the tests require the ability to simulate an unreliable network environment this needs to be accurately simulated in a controlled way to create meaningful tests.

* [P02] Architectural Design

An appropriate architectural design was established. This included the hardware and software components, their relation to the MQTT middleware layer and the methods by which these components needed to be linked.

* [P03] Development of Server Application

A server application was developed using Java to provide functionality typical of a Point of Sale application such as customer, product and transaction management.

* [P04] Development of Client Applications

A client-side mobile application was developed to collect data typical to that of a PoS system and then attempting to send this data to the Server application to process a transaction. In addition to the mobile application, a desktop application was developed to allow for access to the server on a platform other than the mobile application to have easier access to the results of testing for different identified use cases. This is typical of PoS systems whereby a management application is developed to allow for a broader overview of transactions across an entire store for example.

* [P05] Implement MQ Telemetry Transport Protocol

MQTT was deployed to facilitate communication between the mobile client application and the server and involved setting up a Broker, a Client within the mobile application to publish messages to the Broker and a Subscriber within the server to receive transaction messages from the Broker to process them.

* [P06] Testing

The test plan was executed, and results gathered.

* [P07] Analysis of Results

This document contains the analysis and discussion of the results gathered in the Testing phase.

## Structure of the rest of the report

The rest of this report is split into four main chapters covering the Literature and Technology Review, Execution, Evaluation and Conclusions of this project respectively. A sixth chapter contains the Bibliography and references for this document. Where appropriate chapters have been split into subsections, many of which will draw from supporting documentation which can be found in Chapter 7: Appendices.

# Literature Review and Technology Assessment

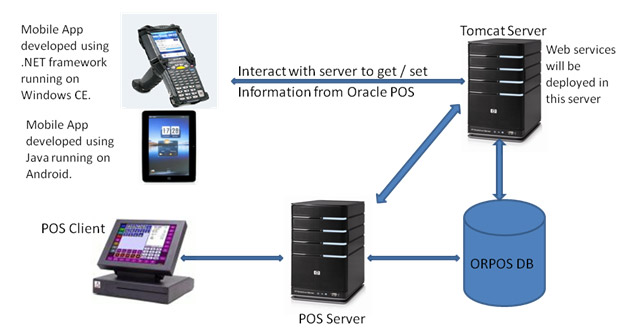
The following section is an appreciation of existing work relevant to mobile Point of Sale and its current trends for adoption, the use of Message Oriented Middleware for creating reliable messaging systems and MQTT as a specific choice of Message Oriented Middleware.

## Mobile PoS

Mobile Point of Sale applications seem to be increasing in their popularity as evidenced by major tech companies such as Apple making the switch to mPoS services as early as 2009, using a third-party application EasyPay coupled with their own iPod devices and magnetic swipe card readers. This allows customers to be served wherever they are in the store without having to wait in queues for a designated PoS area, creating what has widely been viewed as one of the easiest checkout procedures in the world (Catlow, 2014). This is evidenced by larger retailers implementing similar devices and services, resulting in smaller vendors following suit, driving demand and a need for innovation in the field (Toplin, 2016).

Figure 1: Typical PoS Architecture (infodartglobal.com, 2019)

Sales environments aren’t the only market for these technologies either; consumer-driven businesses such as the Hospitality industry are also benefitting from mobile Point of Sale by facilitating convenient service at non-traditional locations – allowing staff to serve drinks at beach resorts or on golf courses for example (Hertzfeld, 2017). The basic functionality usually allows sales persons to gather or select from some existing customer information and then link this data to one or more products.

The standard system architecture for these solutions generally follows a Client-Server model whereby the mobile device creates transaction data which will typically be a marriage of customer information such as a name and contact details, tracked in the wider application using a unique customer ID, with one or more instances of product information such as a SKU (Stock Keeping Unit) used as a unique identifier for the product and a quantity to show how many purchases the customer is making as well as calculating the overall cost associated with the sale. This data is then sent to a server which logs the transaction and carries out any associated business logic such as removing the items from stock. A typical architecture for this kind of system is shown in Figure 1: Typical PoS Architecture (infodartglobal.com, 2019).

The server often provides access to this information for multiple sub-systems based on different user requirements. For example, a sales person working on a shop floor only needs to generate transactions, however the manager of the store needs to be able to query the server for all the transactions being captured in the store to get a broader overview. This is usually done through a different interface to the one that is used to capture the data; the client creating the transaction might be a mobile application while the store manager is in an office using a desktop application to view the captured data.

The key component with relation to this project is the network connection required between the Client and Server. It is widely accepted that a PoS application is the most important part of a retail environment and so maintaining this network connection is critically important (Morris, Ken, Boston Retail Partners, 2014). In most cases, solving this problem appears to translate to system architectures that are both expensive (Weinberg, Neil, Johna, Till Johnson, 2018) and complex, which would perhaps explain why as little as 2% of UK micro-businesses (businesses with less than nine employees) claimed to use some element of mPoS (Sheehan, 2014). Furthermore, given that most deployments of mPoS rely on a broadband connection, recent Ofcom reports would suggest that small businesses in the UK will struggle to implement such a system as a disproportionate number of them do not have access to a good standard of broadband (Chapman, 2017). The increase in demand for mobile Point of Sale services seems to be at odds with the availability of reliable messaging to facilitate their use.

## Message Oriented Middleware

Message Oriented Middleware (MOM) is fast becoming an integral layer when it comes to Application and Systems design, to the extent of replacing what was traditionally referred to as data-access middleware which acts as a bridge between applications and the data they use (Aluise, 1999). One of the key features of MOM is that it allows for applications on entirely different platforms to exchange data reliably and securely (Tim Ouellette, 1998) on a loosely coupled basis. This allows for better separation of concerns in that applications can be developed independently of one another and use middleware as a means of universal communication.

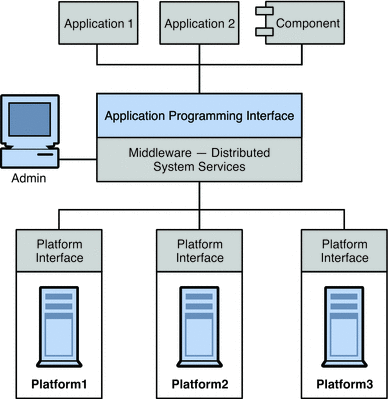
This has proved itself invaluable in a variety of industries, for example, many banks still operate legacy data storage systems (Flinders, 2014) – the lifetime of which have been drastically increased by the implementation of MoM to feed data to several more modern systems from cash points to mobile banking applications. MOM is now commonly applied as a sort of glue between clients and servers to provide secure asynchronous communication between application layers (Tucker, 1997) as demonstrated in Figure 2: Communication between Applications through Middleware (Oracle, 2010)

Figure 2: Communication between Applications through Middleware (Oracle, 2010)

To allow for asynchronous messaging between applications, Message Queue design patterns have been developed and are often implemented at the Middleware layer. Queues work in conjunction with the Publish/Subscribe pattern (Gamma, Helm, Johnson & Vlissides, 1994) where applications can send or ‘publish’ information and receive or ‘subscribe’ to information published by another application or device. Queuing occurs when there is a backlog of information to send or process and acts as a buffer to persist that information until it can be processed. The result is asynchronous messaging where messages are sent on a fire-and-forget basis and placed in a Queue independent of the application, see Figure 3: Example of a Message queue (Amazon, 2019).

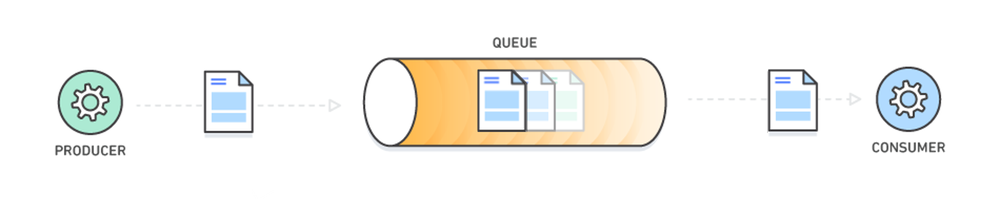


Figure 3: Example of a Message queue (Amazon, 2019)

Likely one of the most commonly used applications of Message Queues are instant messaging services. Facebook Messenger allegedly reached over 1.3 billion monthly users (Shinal, ) in 64 countries (Snelling, 2018) and uses MOM and Message Queuing to ensure delivery of messages regardless of the reliability of the network as well as to leverage the publish-subscribe pattern to allow for group chat. Given that Facebook often leads the market in mobile app installs (Brown, 2018), their use of Message Oriented Middleware would suggest that it can be effectively utilised to provide reliable messaging on mobile devices.

## MQ Telemetry Transport

Message Queue Telemetry Transport (MQTT) is a current and popular Middleware solution which utilises a Message Queue Publish/Subscribe implementation offering three Quality of Service levels which define rules for the sending of messages; At Most Once, At Least Once and Exactly Once (IBM Knowledge Center, ). These QoS levels offer developers flexibility depending on the problem scope, for example; in the scenario for this project a retail transaction must only be processed once; MQTT explicitly provides a QoS level of “Exactly Once” for such a use case.

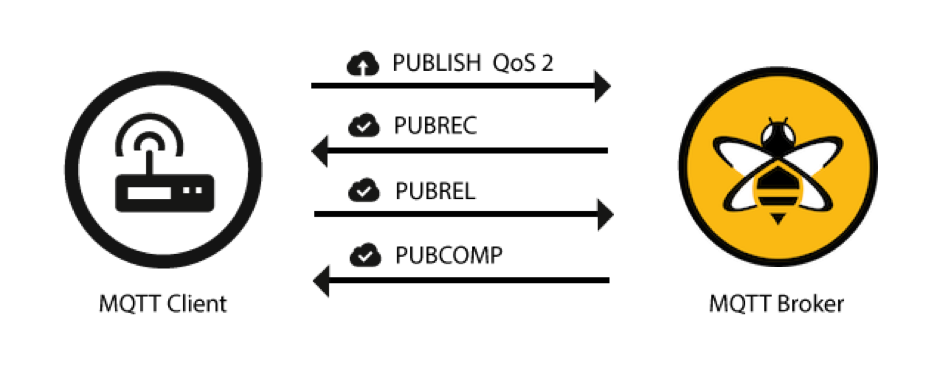


Figure 4: MQTT QoS 2 (HiveMQ Team, 2015)

MQTT has in recent years seen a huge spike in adoption due to the current trend for Internet of Things (IoT) devices which shows no signs of slowing; Market research group Gartner forecasts there will be more than twenty billion appliances, TVs and other “smart” devices connected to the internet by 2020 (Kuchler, 2017). MQTT is a lightweight and portable solution to the connectivity issues associated with the networking of many low-power devices. This makes it well suited for implementation in mobile applications where it will have less effect on battery usage and networking overheads. MQTT operates as a Publish-Subscribe messaging transport protocol optimized to connect physical world devices and events with enterprise servers and even other consumers. It is designed to overcome the complexities of connecting the rapidly expanding number of information gathering sensors such as phones, tablets and IoT devices like smart thermostats, with software processing technologies (Anon., 2013) in a data-agnostic fashion, meaning that it is well suited to working with multiple systems rather than being developed for one bespoke system. This would better allow for incorporation into existing technology stacks, perhaps where a legacy Point of Sale system already exists.

Other protocols exist for the purposes of providing reliable messaging with some inherent benefits over MQTT. For example, Constrained Application Protocol (CoAP) (Bormann, 2014) supports content negotiation similarly to HTTP which would allow clients to know what type of data they will be receiving (JSON, XML, etc), a feature which MQTT lacks, meaning clients must know up front what type of data they will be receiving and in what format (Jaffey, 2014). While this would be useful, CoAP operates over UDP and therefor does not have baked in support for SSL or TLS encryption and so would be a severe vulnerability given the financial nature of transaction data. MQTT on the other hand uses TCP/IP or Web Sockets and so is easier to secure with existing standards.

The applications of middleware like MQTT have grown exponentially, from its initial conception as a way of transferring telemetry data between satellites – hence the name, to healthcare, oil and gas, as well as more consumer facing devices such as media and entertainment, smart devices, and even has vehicular applications (Anon., 2016). MQTT saw a boom in popularity recently when it was implemented by both Amazon - who chose MQTT to facilitate scaling to “billions of responsive long-lived connections between things and your cloud applications” (Jeff Barr, 2015), and Facebook - who used it as a platform from which to build their hugely popular Messenger service, stating that it allowed them to design the platform for mobile devices without compromising battery life (Zhang, 2011).

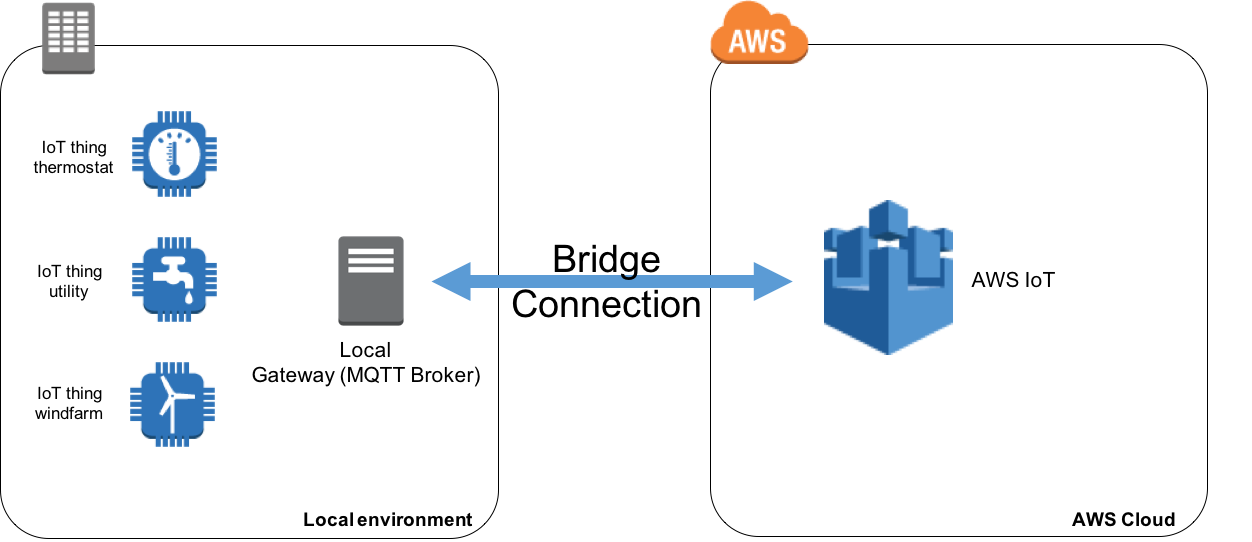
This brought the technology into the spotlight and proved that despite originating in 1999 it still exists as a powerful tool today. Given that Amazon use the protocol to facilitate messaging at enormous scale in their cloud environments as seem in Figure 4: MQTT as a means of connecting devices to AWS (Michael Garcia, 2016), it would stand to reason that it would easily be able to handle the comparatively lighter data load associated with smaller brick and mortar retail businesses. MQTT is also abundantly deployed in aid of Internet of Things devices to facilitate their ability to communicate. IoT devices rely on lightweight, low-latency middleware to establish the networking of physical devices. Typical examples including smart lights for the home which can be switched on and off from a mobile application, or coffee machines linked to a similar app that allows them to be synced with the user’s calendar to save them time in the morning.

Figure 4: MQTT as a means of connecting devices to AWS (Michael Garcia, 2016)

The low overheads associated with MQTT in a mobile application would allow more bandwidth to be allocated to downloading the large amounts of data associated with modern multimedia app experiences. A modern mobile PoS application might include everything from text-based product details, high-resolution product images, or even demonstration videos or reviews. MQTT’s lightweight network load would better allow for this data to be downloaded while transactions are reliably handled by a fast and lightweight solution.

Many of the potential applications of MQTT in the wider field are still in their infancy and appear to be considered as highly lucrative. Cisco reports published in 2016 suggested that IoT devices created $19 trillion of value for companies and industries (Wig, 2016), two years later that figure is now estimated to rise to $60 trillion by 2020 (Albert, 2018). With such a growing industry it is surely worth testing the applications of this emerging technology in other new and expanding fields with mobile Point of Sale being a compelling contender.

# Execution

The following section will outline the methods, development processes and technologies that were used to develop the prototype solution and the justification for their selection. In summary this will cover the requirements analysis and design process and testing of the following elements:

* A server to carry out the typical functionality of a retail Point of Sale system as discovered in the Literature Review process 2.1 Mobile PoS
* A desktop application designed to fulfil the needs of a store manager in a Point of Sale context.
* A mobile Point of Sale application using MQTT as a transport protocol and the testing of this solution as a viable way to reliably send transaction messages from the client mobile PoS application to the server application.
* The MQTT Broker required to facilitate the sending and receiving of messages between the mobile application and the server.

## Analysis

This section describes the requirements analysis process undertaken.

### The scenario

To put this project into real-world context, the scenario of a brick and mortar store selling musical instruments was selected as a prospective customer for a mobile PoS system. Musical instruments are often bespoke in nature and therefor require a high level of customer service as part of a transaction, often in the form of browsing through a store to try various models with the assistance of a salesperson. This makes a good candidate for an environment that would require a mobile PoS solution due to the roaming nature of the transaction throughout a physical location as discussed in 2.1 Mobile PoS. Considering the musical theme, the prototype software has been dubbed TEmPoS – Telemetry Enabled mobile Point of Sale.

### Lifecycle

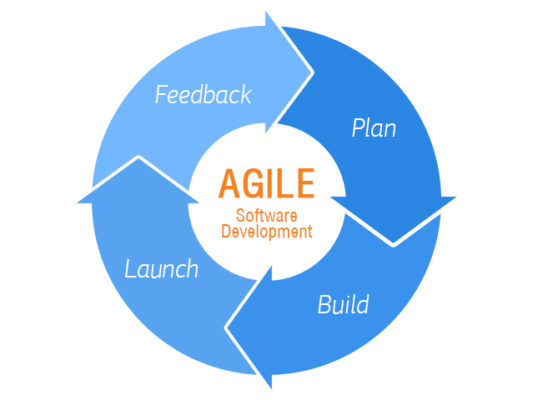
Development was carried out using an Agile cycle consisting of Sprint periods and resulting in the completion of set deliverables – see Figure 5: Agile Development Cycle (Nahian-Al-Hossain, 2016). This allowed for iterative design to continually improve the quality of the prototype applications. Agile Development allows for rapid adaptability to change (Singh & Sharma, 2014) throughout the research process and allows for development to operate on a reactive basis to overcome challenges. To complement the Agile methodology, Trello (Atlassian, 2019) was used to document and plan Sprints – See Appendices: 7.1 Agile Methodology.

Figure 5: Agile Development Cycle (Nahian-Al-Hossain, 2016)

### Requirements

The basic functionality of a PoS terminal as examined in 2.1 Mobile PoS allows for a salesperson to couple customer information with one or more sets of product information such as the type of product, a unique identifier for that product and the quantity to purchase. This data then needs to be sent to a server to be processed and made available to other clients such as an application used by a store manager.

To establish the functional requirements of the prototype in more detail, Use Case diagrams and accompanying User Stories were written, see Appendices: 7.2 Use Cases. The following significant requirements for each respective actor were elicited from this process as they represent key features of PoS applications established in 2.1 Mobile PoS:

**Significant Requirements for Salespersons:**

*Authenticate*: To implement basic security, Salespersons must verify their identity by authenticating.

*Select Customer*: Select a customer from the existing database of customers.

*Select Product*: Select one or more products and assign a quantity to each representing how many are required for the transaction.

*Process Transaction*: At this point a transaction has been created by the marriage of customer and product data, this will then be processed by the server.

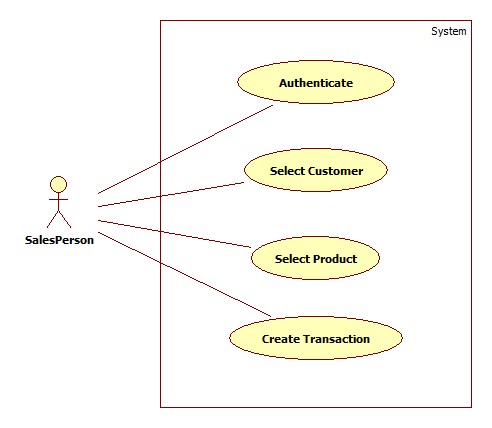


Figure 6: Use Case Diagram for Salesperson

The functionality required by Salespersons will be fulfilled by the mobile application as per the requirements of this project to have Transactions created on a mobile device and then sent to a receiving server.

**Significant Requirements for Store Manager:**

*Authenticate*: To implement basic security, Store Managers must verify their identity by authenticating.

*Manage Products*: Creation, editing and deletion of Products for the store including stock management.

*View Customers*: View Customer details, this could be to help establish purchasing trends, authorise discounts or returns, etc.

*View Transactions*: The Store Manager should be able to view all transactions processed within the store.

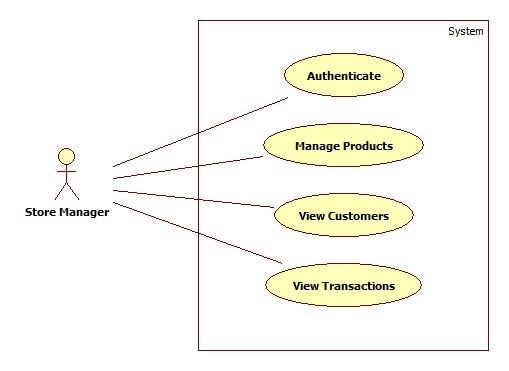


Figure 7: Use Case Diagram for Store Manager

The requirements of the Store Manager actor will be implemented as a Desktop Application as there is no requirement for the Manager to work from a mobile device as they are not creating transactions, simply viewing the transaction data.

Non-Functional requirements were also collated which span both identified actors and indeed the system, see Appendices: 7.3 Non-Functional Requirements.

To satisfy the problem statement of this project, the following underlying requirements are also in place for the system to meet both business logic needs of a PoS application as discussed in 2.1 Mobile PoS but also to implement the solution in a way that reflects the project goals.

* Communication of transaction data between the client-side mobile application and the server must be carried out using the MQTT protocol.
* Transactions created by the client-side mobile application must always reach the server, regardless of the reliability of the network connection at the time.
* Transactions must only be processed once and only once to avoid duplicate transactions.

## Design

This section details the design developed for the prototype solution:

### Context Diagram

The following diagram is intended to illustrate the business logic relationship between the high-level architectural components and the business needs they are intended to fulfil.

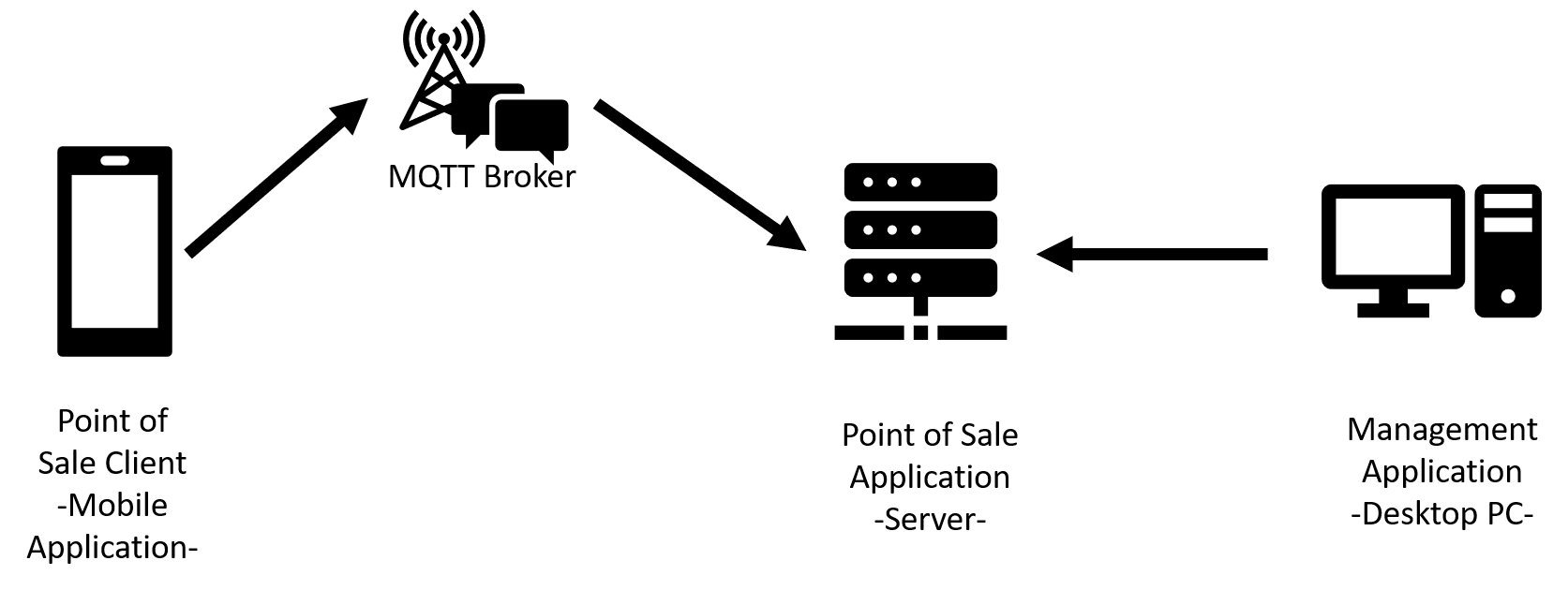


Figure 8: Context Diagram

The Context Diagram highlights that some form of management application to expose the functionality of the Point of Sale Server application is required; a Store Manager for example would want to be able to see all the transactions generated by mobile client devices at the end of the work day. See Appendices: 7.4 Context Diagram for a description of the Context Diagram’s significant entities.

### Class Diagram

A class diagram was developed to map the relationship between the objects in the system based on a Key Abstraction process; see Appendices 7.5.1 Key Abstraction Form. Due to the agile nature of development, these key abstractions and the class diagram itself evolved over several iterations of the development cycle. Figure 9: Class Diagram Snippet illustrates the basic relationship between a Customer and Product object which creates a Transaction. This is fundamentally the core of the system as far as the object-oriented design with most of the other object classes being supplementary to help create these three. See Appendices 7.5 Class Diagram for a more detailed discussion of the Class Diagram.

The main points to take from the class diagram is that to avoid complications due to multiplicity, each Transaction is a marriage of a single Customer instance to a single Product instance with a quantity field to state the number of products to be purchased. This is consistent with the general form of a Transaction as discovered in 2.1 Mobile PoS. The Transaction object itself duplicates some of the information from the Customer and Product objects, this is for ease of use when processing the transaction at the server and prevents views designed to examine Transaction data from having to fetch entire Customer and Product records just to show Transactions in a more readable form.

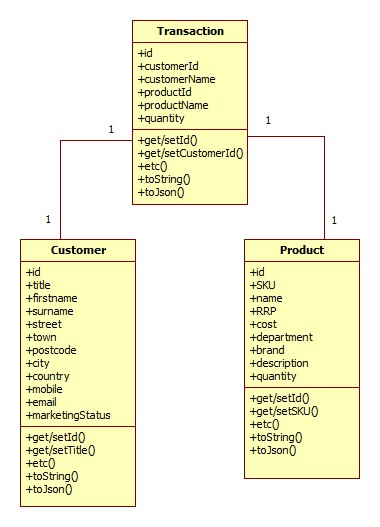


Figure 9: Class Diagram Snippet

## Implementation

This section describes how the prototype solution was implemented, its components and which technologies were utilised for those components and why.

### Components

Summary of developed components and their workspace names:

* Point of Sale mobile Application (Client) – “TemPoSmobile”
* Point of Sale Application (Server) – “TEmPoS”
* Management Application (Desktop Application) – “TEmPoSmanager”
* MQTT Broker – “mosquitto”

\*See Figure 8: Context Diagram

### Technologies and Implementation

Summary of technologies used and the implementation for each component.

**TEmPoS Point of Sale Application – (Server)**

The Server application is designed to work as an API which exposes endpoints to carry out actions on the data stored in the system. Due to existing experience with the technology and its popularity and interoperability across many platforms, particularly in enterprise level applications (Benson, 2017), Java was used as the programming language of choice to develop the server application using the Java Servlet framework to create endpoints for requests by the mobile and management. Java provides well established and documented framework support for Message Oriented Middleware and Publish-Subscribe MQTT implementations which proved easy to integrate through the Eclipse PAHO.mqttv3 library.

The server can be ran locally using the integrated Apache Server framework. Data persistence is handled by the H2 SQL library. MongoDB was also considered as a non-relational system would have afforded more flexibility throughout the established Agile development lifecycle, H2 SQL was selected largely due to existing developer experience with the technology. For more on the structure of the TEmPoS Point of Sale Server Application see Appendices 7.6.1 TEmPoS Point of Sale Application – (Server).

The server design remained consistent throughout development, although it was incrementally expanded upon to provide more functionality over time in the form of more endpoints, but the design was consistent for the most part. The first iteration allowed for the creation, editing and deleting of Customer data, further iterations then added similar functionality for Products and Transactions. The final iterations of the Server saw the implementation of an MQTT Subscriber which would listen for messages received from the Broker and would then attempt to process these messages as a Transaction.

**TEmPoSmanager Management Application – (Desktop Application)**

The Management Application creates a user interface that can communicate with the API provided by the Server and so allows for easier creating, editing and visualisation of the data being handled by the Server. This is intended to fulfil the requirements of the Store Manager actor as established in 3.1.3 Requirements. Given that in practice, this application would allow access to sensitive customer and financial information and likely would be run exclusively on authorised machines within the business, it made sense to consider desktop application framework solutions that provide good data analysis, business centric user interfaces and security tools.

The JavaFX application framework ticks all these boxes and shares Java’s cross platform advantage (Vos, 2018) and so was a logical choice. A Python based Django web-app was also considered here for its ease of setup but was ruled out as it would introduce another programming language to the system which seemed an unnecessary complication. For more on the structure of the TEmPoSmanager Management Application see Appendices 7.6.2 TEmPoSmanager Management Application – (Desktop Application).

The evolution of the Management Application coincided with the development of the Server – as more API endpoints were made available by the server, the Management Application was expanded to provide access to the new functionality. The architectural design remained consistent throughout development; adding new functionality was simply a case of adding more pages for the user to interact with.

**TEmPoSmobile Point of Sale – (Client mobile application)**

The selection of a language for the mobile application took a fair bit of consideration. Most mobile application development frameworks take CSS styled HTML pages with JavaScript functionality and create a wrapper around them to allow them to run on mobile devices (Dimitry, 2017). While this method is widely used and would capitalize on existing developer knowledge and strengths – it would also require duplication of the app to serve more than one platform such as both Android and iOS.

React-Native is a relatively new framework developed by Facebook which uses elements native to mobile devices rather than deploying web assets within a virtual environment. While still very actively being developed and less mature than other approaches, React-Native has been steadily growing in popularity due to the inherent benefit of being able to simultaneously develop for both iOS and Android (Sekulic, 2016).

The use of this language required a steep learning curve, but the resulting product was a significantly more modern solution taking advantage of current trends and multi-platform compatibility. This point was proved during the development process when the iOS device being used for deployment testing died and an Android device had to replace it which worked seamlessly with very little effort or time lost in switching environments.

Another modern feature of React Native that this project benefited from is the use of Web Sockets which React Native implements as an upgrade of TCP/IP for HTTP requests to provide back and forth communication between a client and server, rather than the traditional stateless protocols. Web Sockets are rapidly becoming a popular standard for web based communication so as well as operating on a more modern paradigm, using Web Sockets also allowed for easier debugging with Wireshark (Gerald Combs, 1998) as discussed later in 3.4.1 Testing methodology.

As for the implementation of MQTT, React-Native has several libraries available, however, many of these are based on older versions of the platform and therefor suffer from compatibility issues, more on this in 5.2.4 The TEmPoSmobile Application. To provide the best compatibility with the chosen frameworks and greatest ease of use, an open-source React-Native wrapper around the Eclipse PAHO MQTT JavaScript library was selected as the library of choice (Ryan, 2018). This was the second choice after trialling another library which failed to provide all the necessary functionality of MQTT. For more information on the TEmPoSmobile Point of Sale client mobile application see Appendices 7.6.3 TEmPoSmobile Point of Sale – (Client mobile application).

Unlike the Server and Desktop application elements of this prototype, the Mobile application changed repeatedly throughout this project. This was partly due to inexperience with the language, tooling and framework but also since it is such a current and relatively immature language. A good example of this would be that half way through development, as well as converting to an Android environment as previously mentioned, the choice of library implementing MQTT changed from react-native-paho-mqtt to react\_native\_mqtt. This was a result of discovering that the react-native-paho-mqtt library does not fully support the automatic reconnect functionality provided by the original Eclipse Paho MQTT library. React-native-paho-mqtt had not been updated with these features as it had been superseded by other libraries. React\_native\_mqtt was found to offer more of the features required, including the automatic reconnect functionality as well as being more actively supported and developed by its authors.

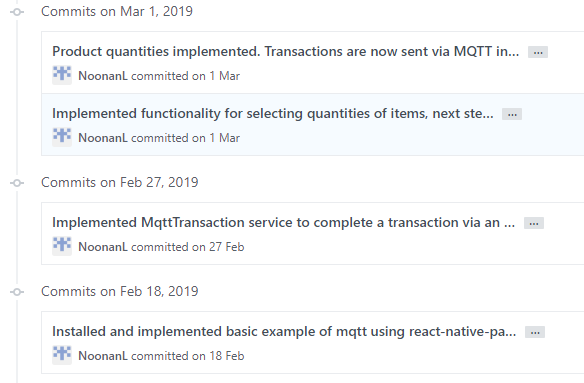


Figure 10: Github Commits for MQTT in React Native

The iterative nature of the development process allowed for functionality to be added incrementally. For example, to become more familiar with the language and development environment used for React Native, the first iteration of the mobile application simply featured a login screen that would return some text to confirm a successful login attempt. The next major hurdle was to implement some navigation between pages, so a further increment allowed a user to login and then be redirected to a home page. By that point the general framework of the application was better established and understood and so the rest of the User Interface was quickly established and most of the rest of development was focussed on elements of functionality as per the requirements established in 3.1.3 Requirements.

The first iteration that allowed the application to send a transaction to the server was implemented using HTTP requests rather than MQTT. This was done purely as proof of concept to ensure that the server would be able to correctly parse a request from the mobile client. This was then transitioned over to using MQTT which involved rapid iterations and refactoring as seen in Figure 10: Github Commits for MQTT in React Native. These iterations took the MQTT implementation from a basic draft to a more fully realised implementation which provided the required functionality

**MQTT Broker:**

MQ Telemetry Transport is available through several different frameworks to serve different platforms and technology stacks. The technology stack established for the other elements of this project have in some ways narrowed the choice of MQTT framework for the prototype. The first contender for the MQTT implementation was HiveMQ, a popular and free online hosted MQTT broker (HiveMQ, 2018). This would have fulfilled most of the requirements of MQTT for this project with the notable exception of security and transparency. MQTT offers several security features, namely the use of user and password protecting access to published topics and SSL encryption. HiveMQ requires a paid subscription for access to these features which would be out-with the scope of this project financially. Additionally, by using an online service, the use of MQTT would be abstracted and therefor obscure the exact knowledge of the design, implementation and performance and so could skew the results of testing.

The use of React-Native for the mobile application had a knock-on effect on the choice of MQTT broker. React-Native implements Web Sockets for network communication as an evolution of the traditional TCP/IP stack as they allow for bi-directional communication – something that TCP/IP was never really intended to provide but has been jury-rigged to do so (Dion Misic, 2018). This creates the requirement of Web Socket support for the MQTT broker. Therefore, based on the constraints of the technology stack, Eclipse Mosquitto (Eclipse Mosquitto, 2018) has been selected as the choice of MQTT broker.

By running Mosquitto on a Linux system, with some configuration, Web Socket support can be achieved to serve the needs of the Mobile Application while the Java based Server can continue to use more traditional TCP/IP access. This decision to deploy the MQTT broker on the Linux platform was later changed as Web Socket support became available on Windows during development, more on this in MQTT Broker MQTT Broker. The decision to run a custom MQTT broker with Mosquitto turned out to be largely beneficial for the project as it allowed for better debugging during development, greater granularity in testing the prototype solution and the ability to capture the network traffic at the testing stage that would not have been as easily achieved if a third party broker had been used.

MQTT allows for three Quality of Service levels as discussed in MQ Telemetry Transport MQ Telemetry Transport. All three of these QoS levels were tested as part of the prototype to determine which, if any would provide the most reliable performance. As a result, the MQTT implementation needed to be clear and readable and easy enough to choose between the different QoS levels. The library used made this straightforward as the QoS level is defined as an integer between 0 and 2 representing each of the QoS levels and selected as part of a client publish or subscribe action. See Appendices 7.6.4 MQTT Broker for more detail.

### System Architecture

As a result of the design decisions made and justified in section 3.3.1 Components and 3.3.2 Technologies and ImplementationComponents, the architectural design of the prototype can be visualised with Figure 11: Architectural Design on the following page.

### Developer Tools

This project covers several technologies across the development stack and so required the use of a suite of tools and IDEs. Most of these choices were developer preference. IntelliJ IDE (IntelliJ, 2018) running in a Windows environment was used for the development of the Java based Server and Management Application.

The Mosquitto MQTT broker was eventually deployed on a Windows operating after early attempts to run it on a Raspberry Pi (Raspberry Pi Foundation, 2018) before it became clear that a Windows deployment could be supported and would help simplify the deployment. Mosquitto Broker configuration is done using a config file and the broker itself can be interacted with using shell commands.

React-Native development lent itself to development on iOS despite being cross-platform once deployed. This is due to the extensive use of libraries and the inability to test the iOS version on a non-native platform. Most of this work was done using Sublime Text editor (SublimeHQ, 2018) and the official iOS IDE Xcode (Apple, 2018) as well as a physical iPhone 6 device for testing purposes until the device died during development and was replaced by a Google Pixel 3. As a result, the React-Native development environment switched to Windows and Android with the editor of choice being Visual Studio Code (Microsoft, 2019) These technologies were backed up by some additional software such as Microsoft Office as well as 3rd party applications such as Trello (Trello & Atlassian, 2018) for project management – particularly for the purposes of coordinating Agile Sprints.

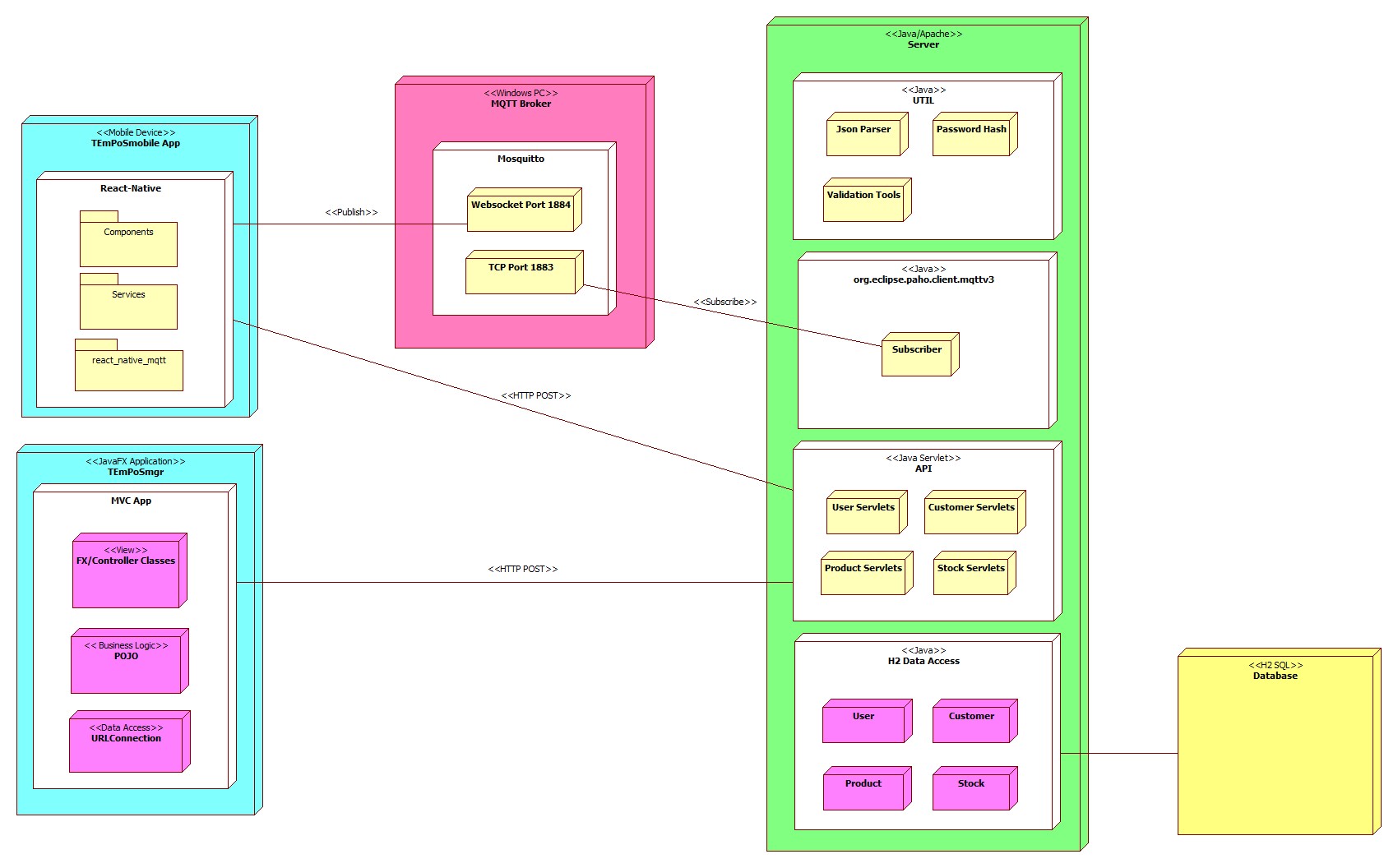


Figure 11: Architectural Design

## Testing

This section describes the testing methodology used and an overview of the results. For full listings of test data see Appendices 7.7 Testing.

### Testing methodology

To determine the effectiveness of MQTT in providing reliable messaging in an unreliable network environment for the purposes of sending transactions between the mobile client and the application server, two main aspects of the prototype needed to be controlled. Firstly, the unreliable network environment had to be simulated. Secondly, each Quality of Service level of MQTT would have to be tested to fully establish whether the protocol could provide the required functionality. To simulate the unreliable network connection, tests would be carried out with the internet connection to the client mobile devices being disconnected during the test. This took the form of three tests, each of which were applied to all three Quality of Service levels of MQTT; see sections 3.4.2 QoS 0 Testing, 3.4.3 QoS 1 Testing and 3.4.4 QoS 2 Testing. For a more detailed description of each QoS level of MQTT, see Appendices 7.6.4 MQTT Broker.

The three tests carried out at each QoS level are as follows:

Test 001: Control. The purpose of this test to ensure that with an established network connection the client can successfully publish a message to the MQTT broker.

Test 002: Short loss of connection. Before attempting to publish a message to the broker, the client is disconnected from the network. The attempt to publish is then made, however the client device is not reconnected to the network for a brief period - approximately ten seconds or less. The purpose of this test is to determine if MQTT can successfully publish a message to the broker even when there is no internet connection at the time of sending.

Test 003: Lengthy loss of connection. Before attempting to publish a message to the broker, the client is disconnected from the WIFI. The attempt to publish is then made, however the client device is not reconnected to WIFI for a period – this time, at least one minute or more. The purpose of this test is to determine if MQTT can persist a message over a longer period and publish it when a connection is later re-established.

Two main sources were used for gathering test data. Firstly, the console logs of the applications themselves were used to print out various statements to highlight what was happening behind the scenes, and secondly Wireshark (Gerald Combs, 1998) was used to capture the network traffic itself to determine if the packets were indeed reaching their destination.

Using Wireshark was particularly effective when coupled with the use of React Native for the mobile application. React Native specifically implements Web Sockets as an upgrade from TCP/IP HTTP requests as established in 3.3.2 Technologies and Implementation and further discussed in Appendices 7.6.3 TEmPoSmobile Point of Sale – (Client mobile application). In summary, the local network used for testing had no Web Socket traffic on it other than that created by the testing process, making it much easier to filter for than the thousands of TCP/IP connections. Examining the Wireshark log shows the direction of the traffic as it gives detailed information on the source IP address and destination IP address allowing for identification of both the client mobile application and the MQTT broker (Pradeep Singh, 2016). See Figure 13: Wireshark Log for QoS 0 for an example.

Disconnecting the client device from the internet was achieved by turning off the WIFI connection at specific stages during the test as seen in Appendices 7.7 Testing. The device would still have access to mobile data in the form of the 3G connection supplied by the network carrier, however since the test servers were running on a local network, the client device would not be able to use the 3G connection to connect to the locally hosted MQTT broker or Server thus maintaining the integrity of this testing methodology.

The QoS level of the MQTT messages being published was controlled by a method within the mobile application code as seen in Appendices 7.7.10 QoS Testing Code Snippets.

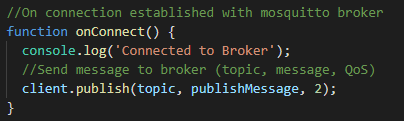


Figure 12: MQTT Publish QoS Code Snippet

### QoS 0 Testing

MQTT QoS 0 is a fire and forget operation in which the message from the client is sent down the wire to the broker once, with no acknowledgement of it being delivered or subsequently relayed to any listening subscribers (OASIS, 2015). As a result, if the message does not reach the broker, no attempt is made to redeliver it and the message is lost. Through all three tests every message sent by the client mobile device reached the MQTT broker and was subsequently relayed to any subscribed clients including the TEmPoS PoS Server application which then processed the transaction.

### QoS 1 Testing

MQTT QoS 1 ensures “at least once” delivery of a message (OASIS, 2015). This is achieved by having the MQTT broker acknowledge receipt of the message, however the client will attempt delivery continuously until this happens. This can result in the message being sent to the broker and therefor any subscribed clients more than once. Through all three QoS 1 tests, every message sent by the client mobile device reached the MQTT broker and was subsequently relayed to any subscribed clients including the TEmPoS PoS Server application which then processed the transaction.

### QoS 2 Testing

MQTT QoS 2 ensures delivery of messages exactly once. This is done via a four-way handshake operation between the client and the broker acknowledging receipt of the message (OASIS, 2015). Through all three QoS 2 tests, every message sent by the client mobile device reached the MQTT broker exactly once and was subsequently relayed exactly once to any subscribed clients including the TEmPoS PoS Server application which then processed the transaction.

# Evaluation and Discussion

As seen in 3.4Testing and Appendices 7.7 Testing, the prototype solution developed for this project passed all of the tests undertaken. In summary, the solution developed is a mobile application which can communicate Point of Sale style transaction data to a receiving server via an MQTT broker such that the messages sent will always arrive at their destination regardless of any loss of connection that might occur during the process. Based on the findings of 3.1.3 Requirements and from earlier Literature Review 2.1 Mobile PoS the two main challenges for the prototype is the ability to ensure the delivery of the message as well as ensuring that the transaction would only be processed once. The following is an evaluation of each QoS level of MQTT and the extent to which they can overcome these challenges.

## QoS 0

Despite passing all tests, QoS 0 would be the least effective QoS level with which to deploy this solution. The primary reason for this service level passing its tests is due to the way MQTT was implemented on the client side. In short, the code responsible for publishing a message to the broker is only called once the client has made a connection with a broker. The default implementation of the MQTT library used for the mobile application will continue to try and achieve a connection to the broker until a time limit is reached (see Appendices 7.6.3 TEmPoSmobile Point of Sale – (Client mobile application)). As a result, since in each test case the connection is eventually established, the message is subsequently always published successfully.

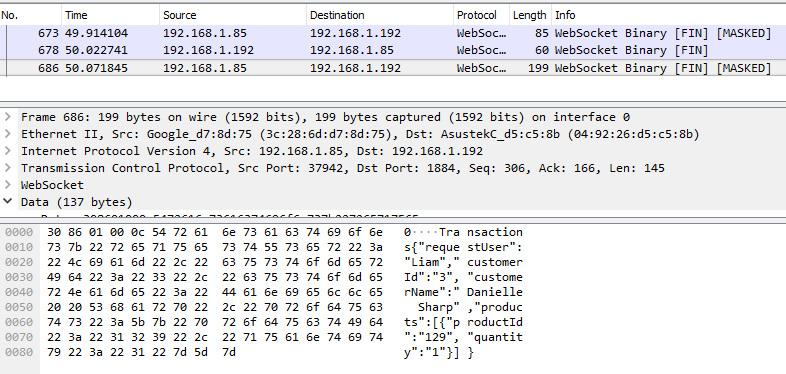


Figure 13: Wireshark Log for QoS 0

This can be seen in Figure 13: Wireshark Log for QoS 0 – the first two items No. 673 and 678 are the packets which create the connection between the client and the broker, No. 686 is the publish message (the JSON for which can be seen in the lowermost window). Notice that there are no follow up responses from the broker (192.168.1.192) to the client (192.168.1.85) to acknowledge the message. As a result, the broker will not reattempt delivery but also has no confirmation that it was ever delivered meaning that Transactions generated by the mobile PoS application could be lost.

For example, if a connection with the broker had already been established and then a message was published when a network connection was present for the client application but not the broker, then the client would have sent the message, but it would not be received by the broker and thus a transaction lost. This would be a failure to reliably send the transaction as established as a requirement in Mobile PoS Mobile PoS and so makes QoS 0 unsuitable for use in the PoS scenario of this project.

## QoS 1

Quality of Service 1 also passed all tests, but again is not a bulletproof solution to reliable messaging with MQTT in the scope of a Point of Sale application. As defined in 2.1 Mobile PoS, it is unacceptable for transactions to be duplicated within a PoS environment. QoS 1 ensures delivery of transaction messages as the client will continue sending the message at increasing intervals of time until it receives an acknowledgement packet from the server (OASIS, 2015) confirming that the message has been published as seen in Figure 14: Wireshark Log for QoS 1 where the final packet listed (No. 325) is the acknowledgement packet sent from the broker (192.169.1.192) to the client (192.168.1.85) which will prompt the client to stop reattempting delivery of the message and drop it from memory.

This has the potential to result in several of the client’s repeated attempts to publish reaching the broker and then being rebroadcast to subscribed clients before the client is notified that the message has been received. This results in duplicated Transactions and therefor a failure in business logic as established in 2.1 Mobile PoS through Literature Review and defined in 3.1.3 Requirements, making QoS 1 unsuitable for ensuring reliable messaging for a Point of Sale scenario.

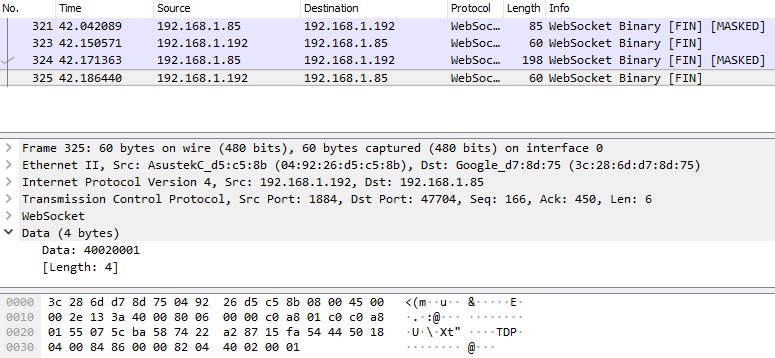


Figure 14: Wireshark Log for QoS 1

## QoS 2

Quality of Service 2 is the best suited of MQTT’s QoS options in the context of a mobile Point of Sale application. QoS 2 ensures delivery once and only once which not only provides the desired reliable messaging as set out in 1.2 Aims and Methodology which QoS 0 cannot reliably provide (see QoS 0 QoS 0) but also eliminates the problems of duplicated Transactions introduced by QoS 1 (see 4.2 QoS 1). As seen in Figure 15: Wireshark Log for QoS 2, the client connection is created and acknowledged (No. 341 and 344 respectively) and then a four-way handshake takes place whereby the client sends the publish message and then receives a PUBREC message from the broker to confirm that the published message was received. If a PUBREC message is not received by the client, the client will resend the original publish message but with an additional flag to mark it as a duplicate to ensure that the broker will not relay it to any subscribed clients more than once.

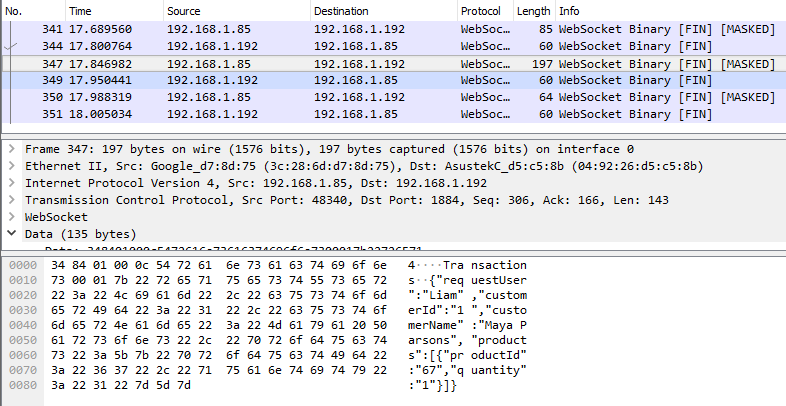


Figure 15: Wireshark Log for QoS 2

Once the PUBREC message has been received by the client it will then send another message (PUBREL) to the broker to confirm that the client has seen that the message was received which the broker will reply to with a PUBCOMP message to tell the client that the publishing process has been completed.

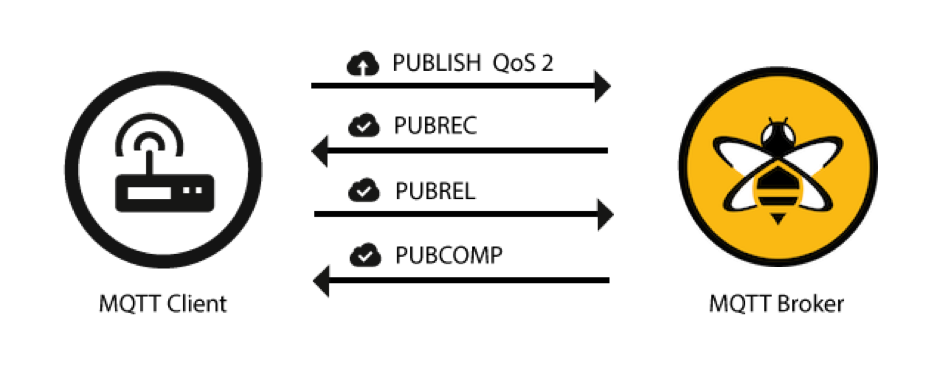


Figure 16: QoS 2 Four-Way Handshake (HiveMQ Team, 2015)

This four-way handshake illustrated by Figure 16: QoS 2 Four-Way Handshake (HiveMQ Team, 2015) not only guarantees that the message always reaches the broker, but that it only does so exactly once which satisfies the business requirements established through Literature Review in 2.1 Mobile PoS and specified as a requirement in 3.1.3 Requirements.

As shown in Figure 15: Wireshark Log for QoS 2, this creates greater network overhead due to the additional packets required for the various confirmations but given the inherently low overhead nature of MQTT itself, this is fairly negligible; At any one time a bricks and mortar retail store could only generate as many transactions as it has staff. As a result, it is likely that the throughput for a system of this kind would be low, even factoring for multiple branches and so any network overhead introduced by QoS 2 can be ignored.

This makes QoS 2 the most reliable implementation as it guarantees message delivery regardless of the reliability of the network while also maintaining the business logic of a Point of Sale system by ensuring Transactions are not duplicated.

# Conclusions

This final chapter provides the conclusions drawn from the execution of this project, starting with an overview of the work completed, conclusions relating to the developed solution’s effectiveness and finally some notes on potential further avenues of research.

## Overview

To discover if MQTT can be used to provide reliable messaging in unreliable network environments for a Mobile Point of Sale application this project completed the following objectives:

* Using a literature and technology review; evaluated the requirements of Point of Sale environments and the extent to which mobile PoS is a growing market. Examined existing technologies such as Message Oriented Middleware used to provide reliable messaging and more specifically the capabilities and existing applications of MQTT.
* Established the functional and non-functional requirements typical of a Point of Sale system.
* Designed a client-server mobile Point of Sale application including the design for the mobile Client using React-Native, the design of a receiving Server using Java Servlets as well as a JavaFX desktop application for interacting with the Server out-with the mobile application.
* Implemented MQTT to provide messaging between the client Mobile Application and the Server Application.
* Tested the reliability with which MQTT could provide messaging between the Client and Server applications at different Quality of Service levels as established through the Literature and Technology Review; section 2.3 MQ Telemetry Transport. Each Quality of Service level was tested across three scenarios in which the network environment was intentionally disconnected and then reconnected to validate whether the messages would still be received at the Server.
* Evaluated the results of the testing process to determine that MQTT could indeed provide reliable messaging in unreliable network environments in the scenario of a Mobile Point of Sale application.

## Overall Conclusions

A summary of the overall findings of this report.

### MQTT

Although MQTT can be used to provide reliable messaging in unreliable network environments for a mobile Point of Sale system it must be carefully configured to do so. Quality of Service levels 0 and 1 were found to be unsuitable as they either do not guarantee delivery of the Transaction to the Server or they pose a risk of duplicating Transactions which would lead to a failure in business logic, resulting in customers being charged twice or stock being incorrectly decremented for example.

Quality of Service level 2 was found to fully provide reliable messaging between the mobile application and the server as it ensures delivery of the Transaction once and only once. This works even if the network connection is interrupted momentarily or even for longer periods of time. The MQTT publisher embedded in the mobile application will attempt to deliver the message to the Broker and will attempt to do so until it receives an acknowledgement from the Broker that the message was received. The message has a unique identifier which the Broker uses to ensure that it only receives the message once.

There is however, a caveat to QoS 2 that must be addressed if MQTT is to be successfully deployed to provide reliable messaging for this project. MQTT QoS levels are established on a client-broker basis. This means that the mobile application agrees on a QoS level with the broker, but so does every other client such as the Server application which in this case must receive the Transaction message. As a result, when using QoS 2 for communication between the mobile application and the Broker, all the associated components i.e. the Server must also use QoS 2. Using a lower QoS level would result in “QoS downgrading” – a situation where the Transaction would be communicated from the mobile application to the Broker once and only once, but the broker will then relay the message to the Server in line with the QoS level that the subscription has specified. For example, if the Server was subscribed to the broker with a QoS of 0 then despite the fact that the mobile application has used QoS 2 to send the message to the Broker once and only once, the Broker may then attempt to relay the message to the Server but fail to do so as there is no guarantee or proof of delivery as seen in 3.4.2 QoS 0 Testing and illustrated by Figure 17: QoS Downgrading.

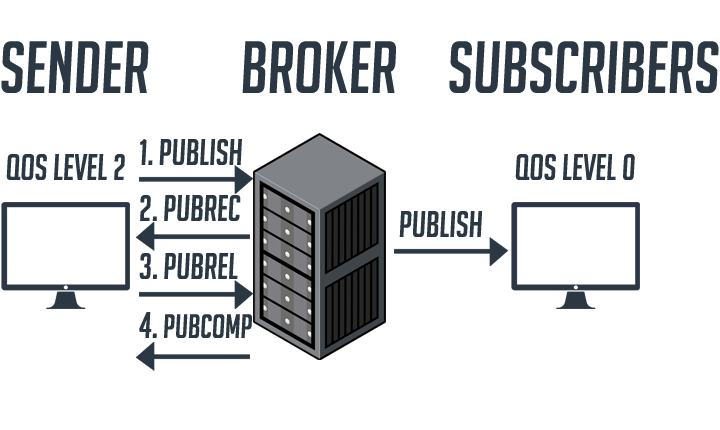


Figure 17: QoS Downgrading

This would represent a failing of the business logic established in 3.1.3 Requirements and so would be a failure of MQTT for providing reliable messaging in a PoS scenario.

A potential limitation of MQTT in the retail Point of Sale sector is its security. Modern PoS systems have stringent security requirements, from encryption to protect credit card numbers to providing privacy for User data. GDPR (European Union, 2018) recently established at the time of writing is more strict than previous data protection laws and is changing the way vendors design software to handle user’s data (Burns, 2018). To fully establish whether MQTT could be deployed in a real-world application with appropriate security features, a further investigation would need to be carried out. Section 2.3 MQ Telemetry Transport of this report established that MQTT packets could be encrypted using SSL, however it is unknown if that would be enough to fully secure User data in a PoS environment. Using the default settings, the raw JSON data sent as MQTT messages for the purposes of this report can be seen previously in Figure 15: Wireshark Log for QoS 2. This is likely not acceptable from a security standpoint.

MQTT was also not tested with different payload sizes as part of this project. This report assumed it is sufficient for Transactions to be represented by a JSON string but as Metadata and associated information becomes such a huge part of data structures it is unclear if MQTT can handle increasingly large messages, this could be another potential drawback.

Using MQTT itself was straightforward and through several early iterations it was found that MQTT could work on almost any operating system which makes it ideal for working with any number of technology stacks. An early idea for this project involved having the Broker deployed on a Raspberry Pi small form factor computer running a Linux based operating system. This was later decided against to simplify the technology stack but represents another example of MQTT’s versatility across the broad PoS market established in 2.1 Mobile PoS.

### The TEmPoS Server

Retrospectively, many development aspects of this project could have been better and more easily implemented. Largely this is due to the availability of better frameworks and libraries which would have greatly reduced the workload. For example, the Server element essentially involved coding an API from scratch including the handling of the data objects and interacting with the storage tables. This could have been almost entirely handled by a framework such as Django or Swagger which automates the process of building and maintaining a RESTful APIs and would have saved countless hours of coding.

### The TEmPoSmanager Application

Another factor of this project which took up more time than perhaps necessary was the Management Application. While not strictly necessary to determine the results of this project, it proved useful in allowing for the creation of scenario-relevant data to lend more credibility to the test environment, i.e., more realistic data representing customers and products. It also allowed for the viewing of the Transaction data in a more meaningful way, similar to how it would be viewed in a real-world deployment scenario by an appropriate member of staff such as a Store Manager as determined in 2.1 Mobile PoS and 3.1.3 Requirements. A mobile Point of Sale application such as the one developed for this project would need to be able to integrate with other existing technologies and software solutions, so the addition of the management application helped to add a broader perspective to the project.

### The TEmPoSmobile Application

Using React Native is considered a great success of this project as it provided the benefits suggested by the Literature and Technology review and in section 3.3.2 Technologies and Implementation - including the notable ability to simultaneously develop for both Android and iOS devices. In some places development of the mobile application was held back by limited integration with comparatively older technologies. For example, setting up MQTT to work with React Native required testing with multiple frameworks as React Native is currently under development to the point that there simply isn’t a huge amount of support for MQTT as so few people have attempted to use it with the framework. This suggests that there is some usefulness to this project in that it has attempted something with technologies in a way that previously has not been tried, at least not to a well-documented extent.

## Further Work

The points listed in 5.2.1 MQTT state that MQTT can be used to provide reliable messaging for a mobile Point of Sale application in unreliable network environments but that it perhaps should not do so based solely on the merits of this report. With that in mind, there are many ways in which more could be done to establish its viability and there are other scenarios in which MQTT could be used which have become apparent from the execution of this project as outlined in the following sections.

### MQTT Security

To fully establish if MQTT is suitable for a PoS environment it is important to establish if it can be used to securely send messages to an acceptable standard with respect to things like encryption of credit card details and the privacy of customer data as per GDPR (European Union, 2018).

### MQTT Data Transfer

As previously mentioned in 5.2.1 MQTT, MQTT could potentially be used to send larger messages but the limits of this would need to be explored. A possible issue with the mobile application developed for this project lies in the fact that it still requires a network connection for all the functionality other than the sending of Transactions to the server, i.e. a connection is still is required to log in, fetch the list of available customers and products. This somewhat undermines the usefulness of MQTT if the user needs a connection to use the app in the first place. Further investigation could help establish if it is possible to use MQTT for the entirety of the communication between the client and the server, including securely logging in and fetching the customer and product data the application relies on.

## Final Thoughts

This project has proved that MQTT can be used to provide reliable messaging for a mobile Point of Sale application in unreliable network environments, but that it perhaps needs some further investigation from different viewpoints such as security to fully validate it as a solution to move forward with. There are however many other potential applications of MQTT that this project has brought to light.

MQTT could be utilised as a tool to prompt updates in systems that need to operate in near-real time. For example, one potential problem with mobile PoS is that client devices must be able to keep track of stocking information, otherwise customers could easily be sold products that the store no longer has stock of because another client has already sold them without updating the stock status. MQTT could be used to publish a message to other clients whenever a Transaction is completed to prompt them to update their stock levels.

In a similar fashion, MQTT could be used to update other applications. When viewing the list of Transactions, the TEmPoSmanager application developed for this project requires the user to a click a “Refresh” button to reload the list of Transactions on the page. This could be set up with an MQTT subscriber listening for new Transactions which would then automatically refresh the list if a new Transaction appears.

One of the largest problems facing the retail industry in current markets, particularly on the high street, is the competition of online stores such as Amazon. Amazon leverage machine learning to change their prices sometimes dozens of times per day to take advantage of consumer trends using information such as how many people have viewed that product in the last hour and so forth (Mehta, Agashe & Detroja, 2017). Brick and mortar stores may try to keep up with these tactics by price matching online vendors. MQTT could potentially be deployed to low power, electronic price cards which would subscribe to an MQTT broker which would update the prices of products with their online price-matched RRP, thus competing with online vendors.

To conclude, MQTT is a powerful, lightweight and useful messaging protocol with numerous potential applications in the retail Point of Sale sector, including as a means of providing reliable messaging for a mobile PoS device in unreliable network environments, but it is not a silver bullet, and its lightweight nature means it should instead be supported by additional technologies.

# Bibliography

. Internet of Things Market - Global Industry Analysis, Size, Share, Growth, Trends and Forecast 2016 - 20242017a*. PR Newswire.* , n/a.

. Mobile Point-of-Sale Market - Global Industry Analysis, Size, Share, Growth, Trends, and Forecast 2017 - 20252017b*. PR Newswire.*

. OASIS MQTT Internet of Things Standard Now Approved by ISO/IEC JTC12016*. Targeted News Service.* , n/a.

. OASIS Members to Advance MQTT Standard for M2M/ IoT Reliable Messaging2013*. Business Wire.* , n/a.

ALBERT, C. 2018, April 06,. Next big things in IoT predictions for 2020*. IT Pro Portal*.

ALUISE, S.J., 1999. Struck in the middle again?: Message-oriented middleware enhancements increasingly drive business integration*. Enterprise Systems Journal; Dallas.* [viewed Oct 29, 2018] **14**(3), pp. 48+. Available from: <https://search.proquest.com/docview/218645780/C039AA6F745A44DCPQ/9>.

AMAZON., 2019. *What is a Message Queue?* Available from: <https://aws.amazon.com/message-queue/>.

Apple., 2018. *Xcode.*

Atlassian., 2019. *Trello.*

BENSON, D. 2017, Feb 1,. Overview - Developers: The Top 10 List of Popular Programming Languages*. Open Source for You; New Dehli*.

BORMANN, C., 2014. *Constrained Application Protocol.* <https://coap.technology/>: .

BROWN, E. ,Facebook leads app install market share, but Google is rising fast, 2018. Oct 19. Available from: <https://www.zdnet.com/article/facebook-leads-app-install-market-share-but-google-is-rising-fast/>.

BURNS, H. 2018, Feb 27,. How GDPR Will Change the Way You Develop*. Smashing Magazine*.

CATLOW,A., 2014. *What Point Of Sale System Does Apple Use?* 25 Oct, [viewed Oct 29, 2018]. Available from: <https://www.expertmarket.com/pos/apple-pos-system>.

CHAPMAN, B. 2017. A shocking number of UK businesses don't have a reliable internet connection*. The Independent.* , -03-24T11:24:33+00:00.

CHOI, S., KIM, H., BYUN, E., HWANG, C. & BAIK, M., 2006. Reliable Asynchronous Message Delivery for Mobile Agents*. IEEE Internet Computing.* **10**(6), pp. 16-25. Available from: <https://search.proquest.com/docview/197335670?accountid=15977> Available from: //dx.doi.org/10.1109/MIC.2006.132.

CONNELLY, J.C., 1988. Marketing POS Services to Retailers*. Bank Administration; Rolling Meadows.* [viewed Oct 9, 2018] **64**(8), pp. 30. Available from: <https://search.proquest.com/docview/204158723/27BCEF50C1A94A37PQ/39>.

DIMITRY,K., 2017. *Guide to the Top 25 Mobile App Development Frameworks.* December 26, [viewed Nov 3, 2018]. Available from: <https://magora-systems.com/frameworks-for-mobile-app-development/>.

DION MISIC., 2018. *A Beginner's Guide to WebSockets.* DION MISIC*, et al.* eds., PyCon Australia: youtube.com.

ECLIPSE MOSQUITTO., 2018. *Eclipse Mosquitto.* -01-08T15:39:57Z, [viewed Nov 3, 2018]. Available from: <https://mosquitto.org/>.

General Data Protection Regulation2018, .

FLINDERS, K. 2014, January. Big banks' legacy IT systems could kill them*. ComputerWeekly.Com*.

GAMMA, E., HELM, R., JOHNSON, R. & VLISSIDES, J., 1994. *Design Patterns, Elements of Reusable Object-Oriented Software.* Addison-Wesley.

Gerald Combs., 1998. *Wireshark.*

GUERIN JACQUES., 2014. *Four key benefits of mobile POS in emerging markets&nbsp;* Sep 23, Available from: <https://mobile-solutions.ingenico.com/blog/four-key-benefits-of-mobile-pos-in-emerging-markets/>.

HERTZFELD,E., 2017. *Why mobile point-of-sale systems are gaining ground | Hotel Management.* May 19, 10:37am, [viewed Oct 29, 2018]. Available from: /tech/why-mobile-point-sale-systems-are-gaining-ground.

HIVEMQ., 2018. *HiveMQ - The Enterprise MQTT Broker.* [viewed Nov 3, 2018]. Available from: <https://www.hivemq.com/>.

HIVEMQ TEAM. ,MQTT Essentials: Quality of Service, 2015. Feb 16,. Available from: <https://www.hivemq.com/blog/mqtt-essentials-part-6-mqtt-quality-of-service-levels/>.

IBM KNOWLEDGE CENTER. *IBM Knowledge Center - Qualities of service provided by an MQTT client.* [viewed Oct 29, 2018]. Available from: <https://www.ibm.com/support/knowledgecenter/en/SSFKSJ_9.0.0/com.ibm.mq.dev.doc/q029090_.html>.

INFODARTGLOBAL.COM., 2019. *Mobile Point of Sale System and Solution - Mobile POS.* Available from: <https://infodartglobal.com/products/mobilepos>.

INTELLIJ, J., 2018. *IntelliJ IDE.*

JAFFEY,T., 2014. *MQTT and CoAP, IoT Protocols.* Available from: <https://www.eclipse.org/community/eclipse_newsletter/2014/february/article2.php>.

JEFF BARR., 2015. *AWS IoT – Cloud Services for Connected Devices.* -10-08T10:06:07-07:00, [viewed Apr 11, 2018]. Available from: <https://aws.amazon.com/blogs/aws/aws-iot-cloud-services-for-connected-devices/>.

KUCHLER, H., 2017. The internet of things: Home is where the hackers are*. FT.Com.* , pp. n/a. Available from: <https://search.proquest.com/docview/1885619087?accountid=15977>.

MEHTA, N., AGASHE, A. & DETROJA, P., 2017. *Swipe to Unlock: The Primer on Technology and Business Strategy.* CreateSpace Independent Publishing Platform (20 Sept. 2017).

MICHAEL GARCIA. ,How to Bridge Mosquitto MQTT Broker to AWS IoT, 2016. Aug 18,. Available from: <https://aws.amazon.com/blogs/iot/how-to-bridge-mosquitto-mqtt-broker-to-aws-iot/>.

Microsoft., 2019. *Visual Studio Code.*

MORRIS, KEN, BOSTON RETAIL PARTNERS, 2014. Virtual POS is Retail Ready - Are you Ready?. Available from: <https://brpconsulting.com/wp-content/uploads/2014/09/BRP-White-Paper-Virtualized-POS-2014.pdf>.

NAHIAN-AL-HOSSAIN. ,NECESSARY FACTORS TO MAKE AGILE SOFTWARE DEVELOPMENT A SUCCESS, 2016. June 9. Available from: <https://www.nascenia.com/necessary-factors-to-make-agile-software-development-a-success/>.

OASIS., 2015. *MQTT Version 3.1.1.* 18 Nov, Available from: <http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html>.

ORACLE., 2010. *Message Oriented Middleware (MOM).* Available from: <https://docs.oracle.com/cd/E18930_01/html/821-2443/aeraq.html>.

PRADEEP SINGH. ,Capturing and Analysing MQTT Packets, 2016. 21 Apr. Available from: <https://iotbytes.wordpress.com/capturing-and-analysing-mqtt-packets/>.

RASPBERRY PI FOUNDATION., 2018. *Raspberry Pi.* Available from: <https://www.raspberrypi.org/>.

REPORTLINKER.COM. 2017. Mobile Point-of-Sale Market - Global Industry Analysis, Size, Share, Growth, Trends, and Forecast 2017 - 2025: Global Mobile Point-of-Sale (mPOS) Market: Report Description*. PR Newswire; New York.* , Oct 16,.

RYAN, C., 2018. *React\_Native\_Mqtt.* <https://github.com/Introvertuous/react_native_mqtt#readme>: , Dec 3,.

SEKULIC, R. ,A brief history of React Native, 2016. -10-03T13:49:03.056Z [viewed Nov 3, 2018]. Available from: <https://medium.com/react-native-development/a-brief-history-of-react-native-aae11f4ca39>.

SHEEHAN, D. ,Only 2% of UK micro-businesses using Mobile Point-of-Sale, 2014. -07-14T13:18:08+00:00 [viewed Oct 29, 2018]. Available from: <https://www.hospitalityandcateringnews.com/2014/07/2-uk-micro-businesses-using-mobile-point-sale/>.

SHINAL, J. . Facebook Messenger is now about as popular as WhatsApp, 2017-09-14T14:36:03-0400.

SINGH, M. & SHARMA, A., 2014. Use and Significance of Agile Software Development*. International Journal of Computer Applications; New York.* [viewed Nov 2, 2018] **94**(15). Available from: <https://search.proquest.com/docview/1525794675/C07B17C7E28F45A0PQ/3> Available from: //dx.doi.org/10.5120/16419-6058.

SNELLING, D. 2018. WhatsApp vs Facebook Messenger - New map reveals the truth about their popularity*. Express.co.uk.* , -02-18T07:53:00+00:00.

SublimeHQ., 2018. *Sublime Text.* Sydney: .

TIM OUELLETTE. 1998. Quick study: Message-oriented middleware--packaged middleware makes inroads*. Computerworld.* , Apr 13, 29.

TOPLIN,J., 2016. *Shifting trends and priorities are set to transform the growing mPOS ecosystem.* Oct 3, [viewed Oct 29, 2018]. Available from: <http://uk.businessinsider.com/mobile-point-of-sale-mpos-trends-business-opportunities-ecosystem-2016-10>.

Trello and Atlassian., 2018. *Trello.*

TUCKER, M.J., 1997. Bridge your legacy systems to the Web*. Datamation.* **43**(3), pp. 114. Available from: <https://search.proquest.com/docview/1791684848>.

UDELL, J., 2002. Reliable Web services*. InfoWorld.* **24**(40), pp. 36. Available from: <https://search.proquest.com/docview/194353539?accountid=15977>.

WEINBERG, NEIL, JOHNA, TILL JOHNSON., 2018. *MPLS explained.* -03-16T14:43-05:00, [viewed Oct 29, 2018]. Available from: <https://www.networkworld.com/article/2297171/sd-wan/network-security-mpls-explained.html>.

WIG, T., 2016. How to secure Internet of Things?*. Express Computer.* , pp. n/a. Available from: <https://search.proquest.com/docview/1789029724?accountid=15977>.

ZHANG,L., 2011. *Building Facebook Messenger.* Aug 12, Available from: <https://www.facebook.com/notes/facebook-engineering/building-facebook-messenger/10150259350998920>.

# Appendices

The following documentation may be useful in better understanding the contents of this report and includes more detailed technical information that would otherwise interrupt the discussion of the main report.

Table of Figures:

[Figure 1: Trello Board During Development 34](#_Toc6346788)

[Figure 2: System Context Diagram 41](#_Toc6346789)

[Figure 3: Transaction Class from Server Application 42](#_Toc6346790)

[Figure 4: Server Application File Structure 44](file:///C:\Users\LiamN\Dropbox%20(Personal)\Honours%20Project\Documentation\Final%20Report\Final%20Report.odt#_Toc6346791)

[Figure 5: Management Application File Structure 45](file:///C:\Users\LiamN\Dropbox%20(Personal)\Honours%20Project\Documentation\Final%20Report\Final%20Report.odt#_Toc6346792)

[Figure 6: Management Application Screenshot 45](#_Toc6346793)

[Figure 7: Mobile Application File Structure 46](file:///C:\Users\LiamN\Dropbox%20(Personal)\Honours%20Project\Documentation\Final%20Report\Final%20Report.odt#_Toc6346794)

[Figure 8: Products page from the developed mobile application 47](#_Toc6346795)

[Figure 9: Mosquitto Broker Configuration 49](#_Toc6346796)

[Figure 10: MQTT Broker QoS 2 Capture 49](#_Toc6346797)

## Agile Methodology

A description of the development methodology used to create the prototype solution and a brief description of Trello which was used as a planning tool.

***-Sprints-***

Development will take place in fortnightly sprints. The sprint cycle will be of the following pattern.

**Trello:**

[www.trello.com](http://www.trello.com) (Atlassian, 2019)

Trello was found to be an incredibly useful tool for planning Agile development cycles. To aid in development, a Trello board was set up with lists that would act as the Product Backlog and individual lists for each Sprint period as well as lists for testing and reviewing items in the Product Backlog.

**Product Backlog:**

The product backlog will act as the staging area for units of work that should be completed, with emphasis on delivering Minimum Viable Product before adding supplementary functionality or features. For the most part, this will take the form of User Stories but in some cases, may include specific changes or features that need to be added that might not necessarily accompany a User Story.

**Sprint Planning:**

Sprint planning will take place iteratively throughout the sprint cycle based on reviews of work carried out. New items may be added to the Product Backlog at any point for consideration by the rest of the team. Each item in the Backlog will be given one of the following status labels:

* Large – Break me down

*-As suggested, Large units of work must be broken down into smaller, better understood units before they can be considered for the next Scrum cycle.*

* Medium

*-Units of work that may be complex to implement and could span the entire Scrum cycle.*

* Small

*-Small units of work that can probably be implemented by a single developer within a few hours.*

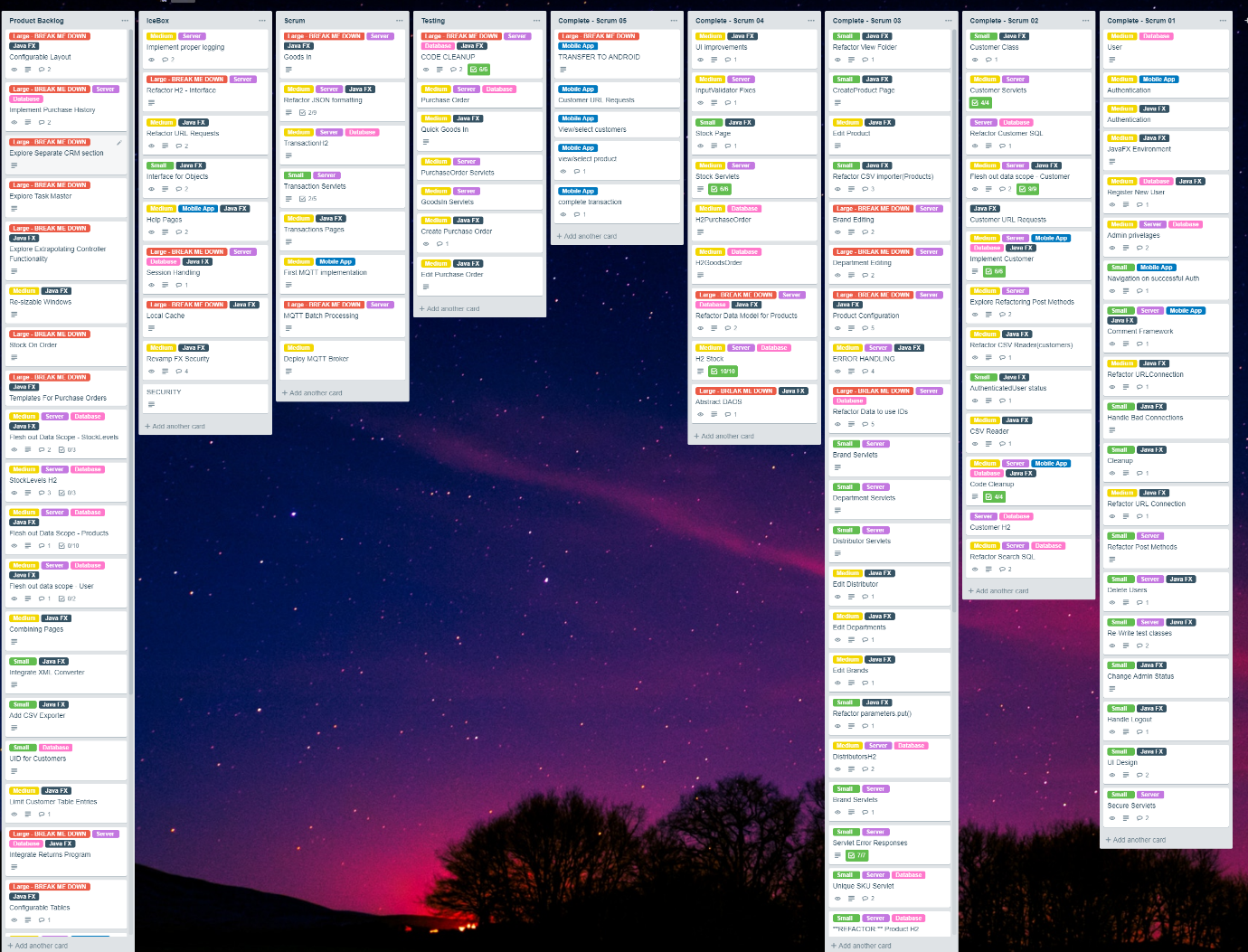


Figure 1: Trello Board During Development

**Sprint Review:**

Traditionally this happens once at the end of each Sprint. Although to an extent this will be the case, for the most part individual items from the Product Backlog will be placed in a Review board via Trello and reviewed throughout the Sprint process. The following questions will be asked at the end of each Sprint and the Trello board adjusted as required, such as refining the backlog, moving cards into the next pending Sprint, etc.

* What was achieved in the last Sprint?
* What will be completed in the next Sprint?
* Is there any impediment in the way of the Sprint goal?

*The process of Reviewing, Retrospective and Refinement should be an ongoing process throughout the course of each Sprint.*

## Use Cases

An example of a Use Case Scenario and Forms which helps define the functionality required of the mobile Point of Sale application.

**Activity:** *Create a transaction*

**Tasks**

1. Authenticate.

* Select the Username field to input their username credential
* Select the password field to input their password credential
* Press the Login button to authenticate

1. Select Customer

* Select “Customers” button from the home screen, this will take them to the list of available customers
* Salesperson can scroll through the list of available customers
* Salesperson can select a customer by pressing the row in the list

1. Select Product

* Salesperson must have already selected a customer, this will take them to the list of available products
* Salesperson can scroll through the list of available products
* Select a product by pressing the green + icon to increment the quantity of that product in their basket
* Decrease the quantity of a product in their basket by pressing the orange – icon. Decreasing the quantity to zero will remove the product from their basket

1. Create Transaction

* Select the green Checkout button to proceed to the checkout.
* Select the “Complete Transaction” button to create the transaction

**Assumptions:**

* The Salesperson already has valid login credentials.
* The customer already exists within the customer database.
* The products are available in the list of products.
* Product availability is not dependant on the quantity selected.
* Access to the server for the purposes of authentication and fetching customer and product data is always available.

**Use Case Form**

|  |  |
| --- | --- |
| **Use Case Name** | Create a Transaction |
| **Description** | A salesperson completes a transaction with an existing customer and needs to add it to the system. |
| **Actors** | Salesperson |
| **Pre-conditions** | Salesperson has access to a mobile device running the TEmPoSmobile Point of Sale application. |
| **Trigger** | A transaction has been agreed upon with a customer. |
| **Main Flow of**  **Events** | 1. Salesperson logs into PoS Application on their mobile device [A1] 2. Salesperson selects an existing customer from the Customer List 3. Salesperson selects the required products from the Product List by incrementing or decrementing their quantity as necessary 4. Salesperson proceeds to View Cart page 5. Salesperson selects Complete Transaction, application notifies user of successful transaction and returns to Home page [A2] |
| **Alternative flow**  **Of Events** | A1: Invalid login details, authentication failed, application returns to login screen.  A2: No connection available to send transaction to server – Salesperson should be notified but the application should still ensure delivery of the transaction to the server as soon as a connection becomes available. |
| **Post-condition** | Transaction is received by the server application and successfully processed. |
| **Issues** | N/A |

**Use Case Form**

|  |  |
| --- | --- |
| **Use Case Name** | View Transactions |
| **Description** | A Store Manager wants to view all of the Transactions processed in the store. |
| **Actors** | Store Manager |
| **Pre-conditions** | Store Manager has access to the TEmPoSmanager Desktop Application |
| **Trigger** | N/A |
| **Main Flow of**  **Events** | 1. Store Manager logs into Management Application on their machine [A1] 2. Store Manager selects the Transactions button from the Home Screen 3. The application fetches from the server and lists all the Transactions in the Transaction table. [A2][A3] 4. Store Manager can select individual Transactions to view more information about them 5. Store Manager can select the “Refresh” button to re-load the data in the Transaction table to check for new Transactions. |
| **Alternative flow**  **Of Events** | A1: Invalid login details, authentication failed, application returns to login screen.  A2: No connection available – Transaction list is empty.  A3: No Transactions exist – Transaction list is empty. |
| **Post-condition** | N/A |
| **Issues** | N/A |

**Use Case Form**

|  |  |
| --- | --- |
| **Use Case Name** | Manage Products |
| **Description** | A Store Manager wants to Add, Update or Remove products from the system. |
| **Actors** | Store Manager |
| **Pre-conditions** | Store Manager has access to the TEmPoSmanager Desktop Application |
| **Trigger** | N/A |
| **Main Flow of**  **Events** | 1. Store Manager logs into Management Application on their machine [A1] 2. Store Manager selects the Products button from the Home Screen 3. The application fetches from the server and lists all of the existing Products in the Products table. [A2][A3] 4. Store Manager can select individual Products to view more information about them in the right-hand pane. 5. Store Manager can create a new product by selecting the “New” Button and following the prompts. 6. Store Manager can delete an existing product by selecting it in the List and selecting the “Delete” button and confirming the action. [A4] 7. Store Manager can edit an existing product by selecting a product in the list and selecting the “Edit” button and following the prompts.[A5] |
| **Alternative flow**  **Of Events** | A1: Invalid login details, authentication failed, application returns to login screen.  A2: No connection available – Product list is empty.  A3: No Products exist – Product list is empty.  A4: No product selected, user prompted to select a product  A5: No product selected, user prompted to select a product |
| **Post-condition** | N/A |
| **Issues** | N/A |

**Use Case Form**

|  |  |
| --- | --- |
| **Use Case Name** | View Customers |
| **Description** | A Store Manager wants to Add, Update or Remove customers from the system. |
| **Actors** | Store Manager |
| **Pre-conditions** | Store Manager has access to the TEmPoSmanager Desktop Application |
| **Trigger** | N/A |
| **Main Flow of**  **Events** | 1. Store Manager logs into Management Application on their machine [A1] 2. Store Manager selects the Customers button from the Home Screen 3. The application fetches from the server and lists all the existing customers in the customers table. [A2][A3] 4. Store Manager can select individual customers to view more information about them in the right-hand pane. 5. Store Manager can create a new customer by selecting the “New” Button and following the prompts. 6. Store Manager can delete an existing customer by selecting it in the List and selecting the “Delete” button and confirming the action. [A4] 7. Store Manager can edit an existing product by selecting a customer in the list and selecting the “Edit” button and following the prompts. [A5] |
| **Alternative flow**  **Of Events** | A1: Invalid login details, authentication failed, application returns to login screen.  A2: No connection available – customers list is empty.  A3: No customers exist – customers list is empty.  A4: No customers selected, user prompted to select a customer  A5: No customers selected, user prompted to select a customer |
| **Post-condition** | N/A |
| **Issues** | N/A |

## Non-Functional Requirements

It is worth noting that non-functional requirements often overlap and cover the same concerns across many different types of project; the following have been noted as being of specific importance to this scenario but realistically this list could be significantly longer and include items such as serviceability, regulatory concerns and so on. These have been omitted as they are not the primary concern of this project. The following Non Functional Requirements have been adhered to as best as possible throughout development:

* The System should have appropriate performance to allow for an acceptable user experience. Since this is just a prototype this is not of utmost necessity but should be considered where possible.
* Scalability should be considered; Point of Sale applications tend to follow a client-server pattern and so inherently must cater for multiple clients interacting with the server at once.
* The prototype should where possible show examples of security considerations as the intended market for this solution would involve the transfer of financial information.
* The system should be easy to use, the mobile application should be as user friendly as possible.
* Communication of transaction data between the client-side mobile application and the server must be carried out using the MQTT protocol.
* The use of MQTT should be abstracted from the user, ie it should not require any specific user interaction for it to work.
* Transactions created by the client-side mobile application must always reach the server, regardless of the reliability of the network connection at the time.

## Context Diagram

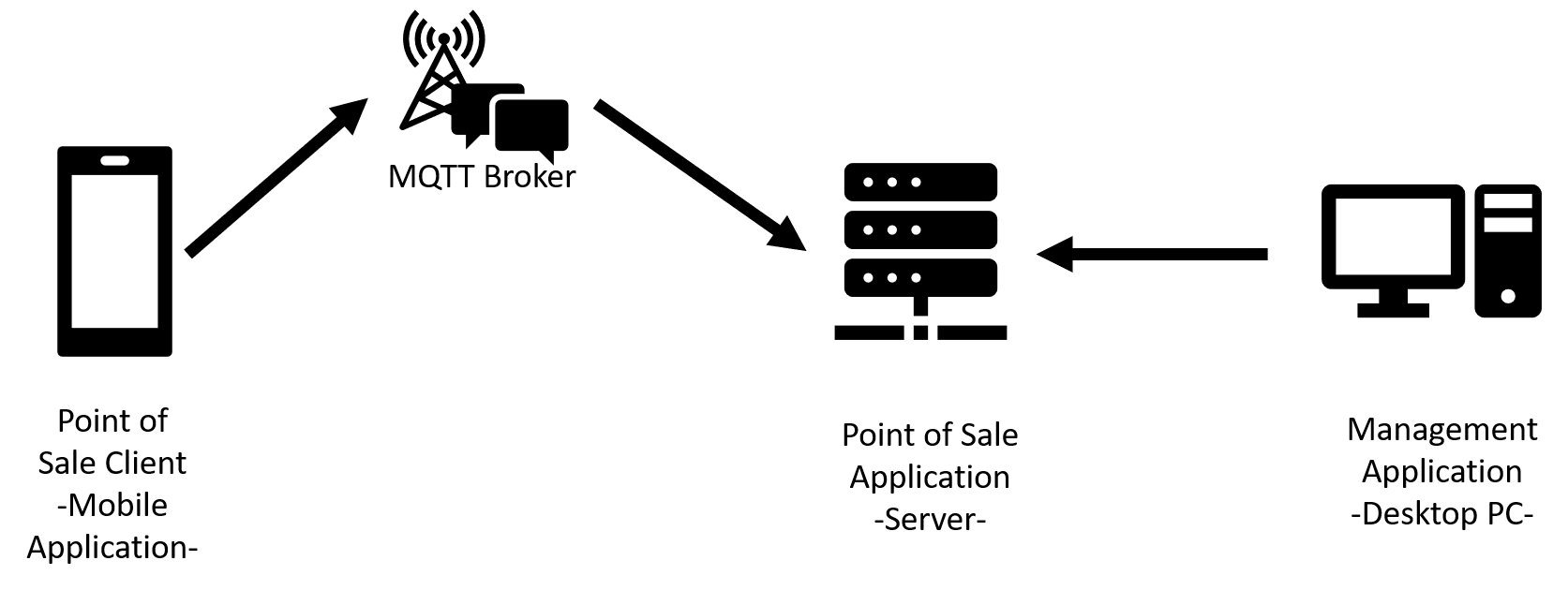


Figure 2: System Context Diagram

Significant Entities:

A brief description of entities that appear in the diagram and their primary relationships.

1. Point of Sale Client – Represents the Point of Sale client application running on a mobile device.
2. MQTT Broker – The MQTT Broker which will facilitate the sending of messages between the client and server applications.
3. Point of Sale Application – A server which handles basic business logic and database persistence for a retail business such as Product, Stock and Customer management.
4. Management Application – Allows for access to the functionality made available by the Server Application, such as adding new Products to the system, viewing Transaction history, etc.

## Class Diagram

Due to the iterative nature of this project and the limited time and resources for documentation, the Class Diagram for the project was all but abandoned at a fairly early stage as documenting it at the pace it was developed and across the full stack of applications being developed was deemed as a less valuable use of time.

Generally, the complexity of the database interactions and object interactions were kept to as low a complexity as possible and so most of the classes essentially work as standalone objects, where a reference to other objects are required, an object will have an additional field which will store a unique ID to the referenced object and this object can be queried separately rather than through the use of more complex SQL queries.

An example of this is shown in Figure 3: Transaction Class from Server Application whereby a Transaction keeps a reference to both the CustomerId and ProductId which can then be queried separately if needed. The inclusion of the customerName and productName creates some duplication of variables but avoids the need for additional queries to the customer object associated with the customer ID or the product object associated with the product ID to get basic information that pertains to the Transaction.

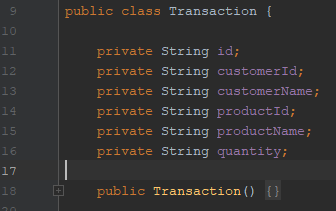


Figure 3: Transaction Class from Server Application

This design principle was followed for most of if not all the Classes throughout the project. Full views of the classes used in the finished prototype are available through code listings.

### Key Abstraction Form

|  |  |  |  |
| --- | --- | --- | --- |
| **Candidate** | **Comment** | **Potential** | **Approved Class** |
| Sale | Type of Invoice | Medium | Rejected |
| Purchase | Another word for Transaction | Rejected | Rejected |
| Transaction | Potentially the same as Invoice? | Medium | Transaction |
| Merchant |  | Medium | Rejected |
| Invoice |  | Medium | Rejected |
| Customer |  | Likely | Customer |
| Payment | Non-specific | Rejected | Rejected |
| Receipt | Another word of Invoice | Rejected | Rejected |
| Goods | Non-specific, see Product | Rejected | Rejected |
| Service | Out-with scope see Product | Rejected | Rejected |
| Barcode | Attribute of Product | Attribute | Rejected |
| Return | Type of Invoice | Attribute | Rejected |
| Cost |  | Attribute | Rejected |
| Price (RRP?) |  | Attribute | Rejected |
| Order | Another word for Transaction | Rejected | Rejected |
| Product |  | Likely | Product |
|  |  |  |  |
| Key |  |  |  |
| Likely | Pretty well confirmed |  |  |
| Medium | Potentially useful, depending on scope of system | |  |
| Rejected | Not likely to make it into the system |  |  |
| Attribute | Attribute of a class object |  |  |

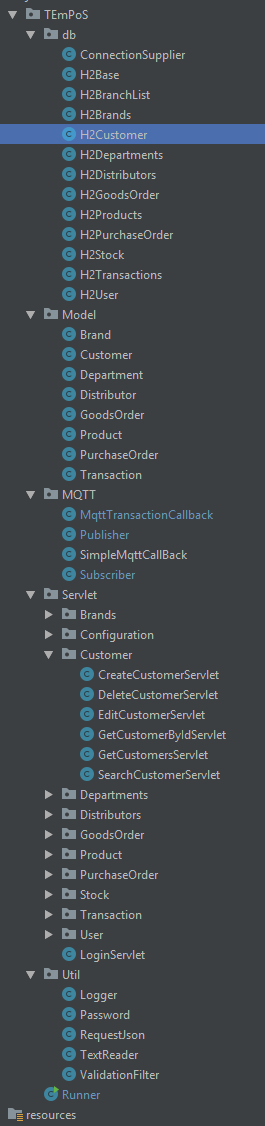
## Implementation

This section of the Appendices gives more detail on the design of the software components of this project.

### TEmPoS Point of Sale Application – (Server)

The server for this scenario was developed based on the functional requirements of a basic retail software system. Typically, this covers three main areas;

* Recording sales
* Product management
* Customer Relationship management

To create this functionality and make it available to users, the server provides endpoints in the same fashion as an API to achieve various operations such as creating a new sale, creating new customers, managing the stock level of products, etc. Java Servlets were used to create the endpoints which would call methods from appropriate database handler classes to update the required rows in an H2 SQL database.

Clients can make POST requests with JSON formatted body to the endpoints which are then checked for their validity for security reasons. This includes;

* Checking that the request IP address is present on a whitelist of approved addresses to ensure the request is from a recognised device.
* Checking that the username of the request client is a valid user to ensure that they are logged in to a valid user account.
* Checking that the request includes a valid secret key – a pre-determined String which only authorised users would know to ensure that the request origin knows the shared secret and therefor is an approved user.
* Checking that the request contains all required parameters to carry out its function, for example if the request is to the Edit Customer Servlet, the request must contain a valid customerId to specify which customer is to be edited.

Once the request is validated, any necessary objects are instantiated, such as a new instance of a Customer which can then be passed to an appropriate Data Access Object to perform the required database operation. With a few exceptions or additions, Servlet endpoints exist to carry out each of the CRUD operations per each type of object in the system.

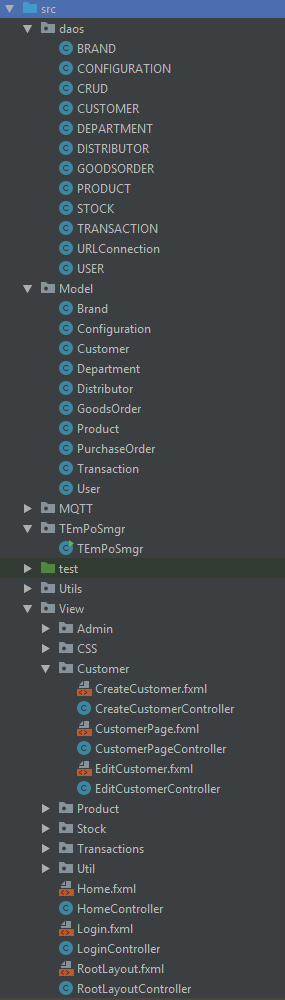
An MQTT client is started when the server starts, connects to the broker and subscribes to the Transactions topic. It will then wait for any messages to arrive at which point it will process them, similar to how the Servlets work but with MQTT sitting in the middle.

Figure 4: Server Application File Structure

***Evaluation:***

For the most part this design worked fine as far as functionality is concerned. Usability and maintainability however were difficult to manage as each object, data handler and servlet was designed manually rather than using an existing framework which could have made this much easier to implement and maintain. One example of a common issue was adding new fields to an object Class. This required manually updating the database handlers, the object, the servlets and then any knock-on effect in client applications. This regularly led to frustrations in development and time spent fixing bugs that were introduced due to poor maintainability. Using an existing framework such as the LoopBack REST API (StrongLoop, 2017) would have made building and maintaining the server side API much easier. This likely would have been at the expense of granular control over how the application behaves and more importantly in this instance, the ability to implement an MQTT client to subscribe to the broker and wait for Transaction messages to arrive for processing.

### TEmPoSmanager Management Application – (Desktop Application)

The manager application exists largely to make debugging and testing easier by creating a user interface with which to interact with the server application. Without this application, the only way to view the data in the SQL database or to view or update that data would be by checking the console log in the server application and making HTML POST requests with an application such as Postman (Postman Inc, 2019).

The application is a JavaFX Desktop app following the Model View Controller design pattern (Gamma, Helm, Johnson & Vlissides, 1994). The user interface is designed using FXML with the help of the bundled SceneBuilder application. These pages then are assigned a controller which provides the business logic of each page and where necessary makes calls to data access methods to then send requests to the Server application to carry out operations such as editing a product for example.

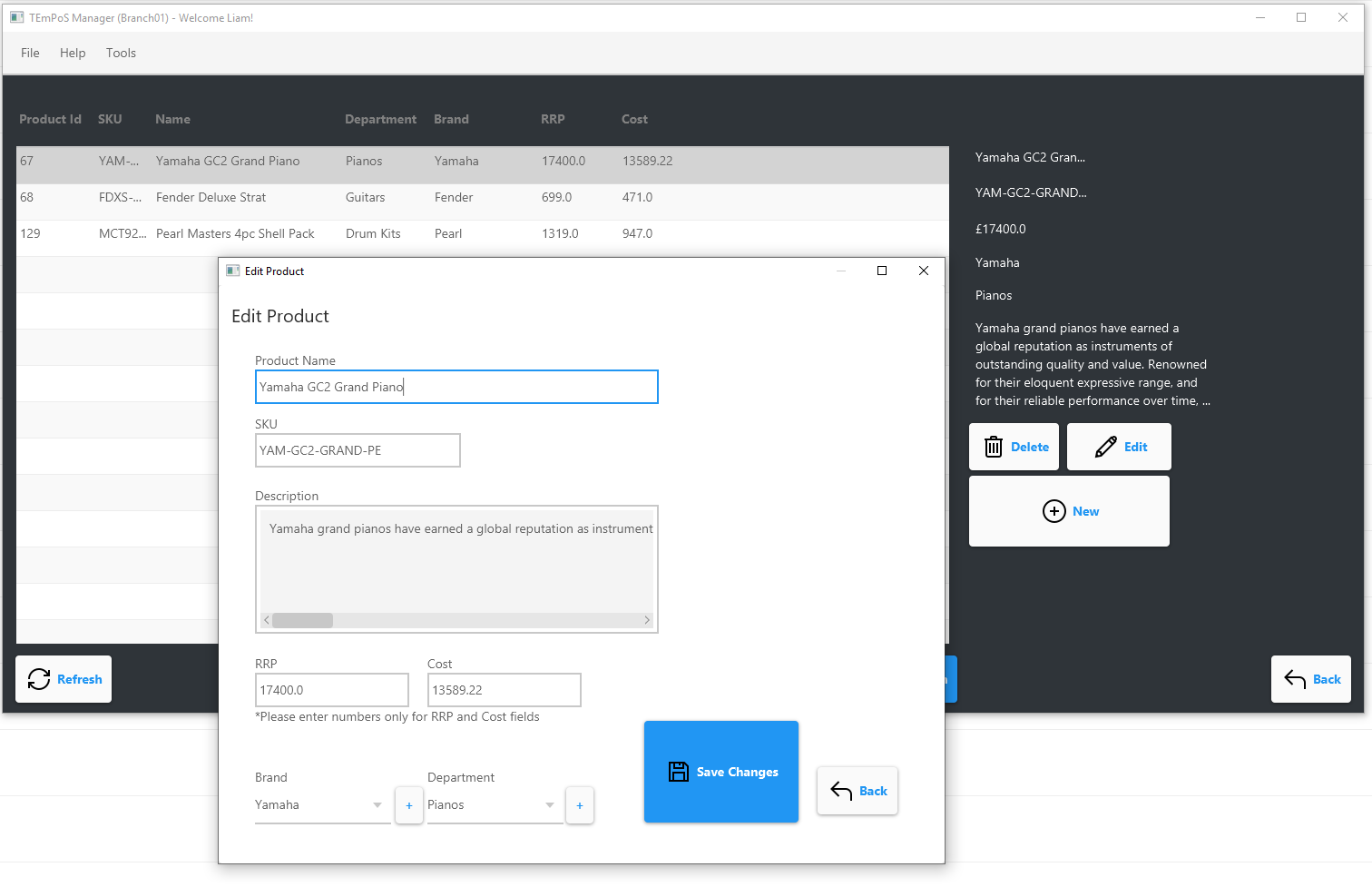


Figure 6: Management Application Screenshot

Figure 5: Management Application File Structure

***Evaluation:***

While not completely necessary for this project, the management application added realism to the scenario by providing an interface to the server application consistent with the requirements of the PoS scenario established in The scenario The scenario. This made for a much more fluid experience when it came to working with sample Transaction information and working out what kind of data would be required in communicating between the client and the server before building the mobile application which would leverage MQTT. The use of the management application also brought to light several other areas within the Point of Sale scenario in which MQTT would be very useful as discussed in Final Thoughts Final Thoughts.

### TEmPoSmobile Point of Sale – (Client mobile application)

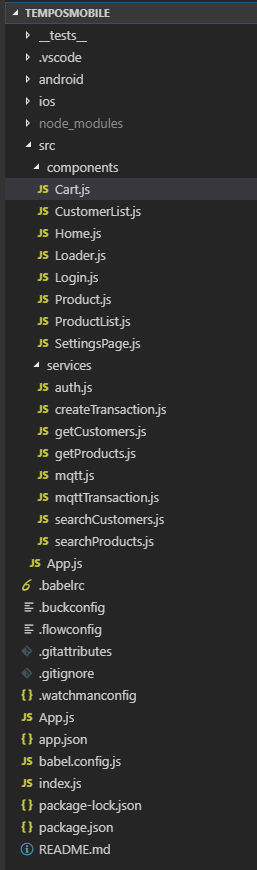
The mobile application was an interesting challenge as part of this project. React Native is a fairly new technology actively being added to and adopted by companies striving to find new ways to write code once and have it serve as many platforms as possible, such as Facebook, AirBnB and many others. Built on React Javascript, React Native uses the components native to mobile devices to construct its User Interface rather than using HTML and CSS and having this converted to native components or placed inside a wrapper to deploy it on a mobile device.

Figure 7: Mobile Application File Structure

React Native is based around designing modular, reusable and individually testable functional components and creating views from them. For example, a Login page may be a component but within that page there could be dozens of smaller components each of those being made up of even smaller yet components, all of which can have their own functionality. This results in very granular, maintainable and scalable code whereby you can develop and test a button with some set of functionality and UI attributes and then reuse it throughout the code base.

In the case of this project the application is quite simple, so the code developed is split into components and services. Components are essentially anything that is used as part of the UI and Services are background tasks that require code only and do not produce UI elements. This is illustrated in the file structure in Figure 7: Mobile Application File Structure.

React Native uses NodeJS to set up the developer environment with all the required files and file structure. Additional libraries can be added using NPM through the console to install or update packages from either official React Native sources or community libraries such as the MQTT libraries used for this project.

During the Build process, this source code is then translated as needed into appropriate binaries for both Android and iOS and the resulting package can be run on either platform within seconds.

***Evaluation:***

React Native was great to work with despite the steep initial learning curve. It provides many cutting-edge features allowing for rapid development such as auto hot reloading where code changes would instantly be deployed to the test device any time the developer hits the save button. The tooling and libraries are designed with excellent supporting documentation and the coding style itself is intended to be very human readable.

Because it is built using modern standards rather than with emphasis on backwards compatibility, it did however have a number of teething problems. The frameworks exclusive use of Web Sockets provided a challenge in finding an MQTT broker that could support this functionality which then had a knock-on effect and resulted in early iterations of this project expecting to deploy MQTT in a Linux environment as the MQTT documentation and support of Web Sockets on windows was limited at the time.

In addition, React Native seems to play to the strengths of iOS environments and was easier to initially set up on the iOS platform. To its credit however, when the decision was made to switch to Android to be able to test the app on a physical device (in this case a Google Pixel 3) React Native absolutely delivered on its cross-platform write-once run-everywhere principle and the code used for the iOS development worked exactly as intended on the Android device. This came at the expense of a longer than expected transition period as it proved more difficult to set up the other tools React Native relies on such as NPM and NodeJS on the Windows operating system, while on iOS it was simply a case of executing a few console commands.

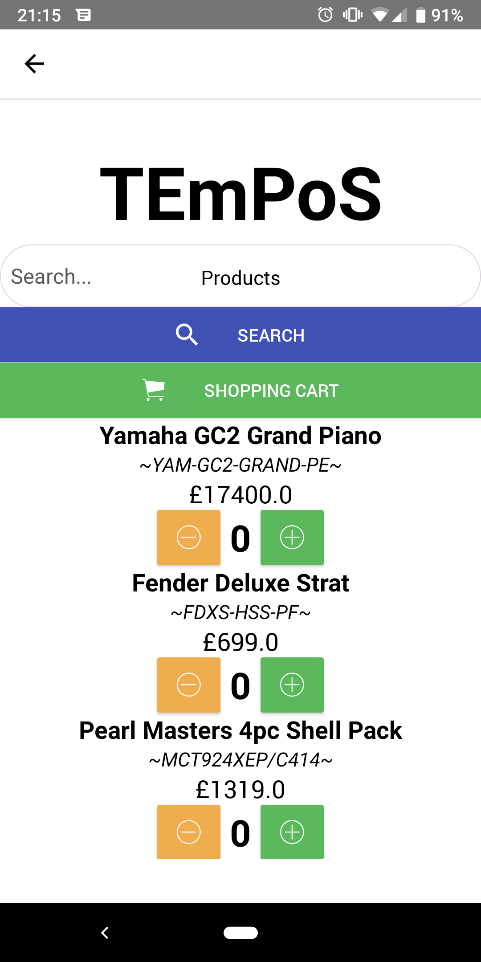


Figure 8: Products page from the developed mobile application

Despite the challenges, React Native was an excellent candidate for this application, providing a modern and forward-thinking solution for the mobile application.

### MQTT Broker

Given that most of this report is targeted at the features and capabilities of MQTT this addition to the Appendices is largely just to give some further detail on the different QoS levels and how they work and to cover some interesting points found throughout development.

QoS 0:

This level is a fire and forget attempt to deliver a message. There is no guarantee of delivery and no acknowledgement of the receipt of the message. Furthermore, the message is not stored in memory for attempted retransmission. This does little more than the underlying TCP protocol as far as transmitting the message to the broker, other than defining the topic it should be attached to should it reach the broker.

QoS 1:

This QoS level ensures at least once delivery of messages by storing the message at the client side and resending it until the client receives a publish acknowledgement or PUBACK packet from the broker. This can result in messages being sent or delivered repeatedly.

The client uses an identifier for each packet to match a publish packet to a corresponding acknowledgement packet at which point it knows to cease any attempt to redeliver the message as it has now been sent to the broker successfully, and so can be removed from memory on the client side.

QoS 2:

As established in this report, QoS 2 is the most reliable of the MQTT QoS levels with respect to the Point of Sale scenario of this project. QoS 2 ensures delivery of a message from a client to the broker or vice versa exactly once. It inherently requires more network overhead to provide this assurance.

Once a message has been published from the client to the broker, the broker replies to the client with a PUBREC or publish-received packet. Until this PUBREC packet arrives at the client, the client will try to reattempt delivery with an additional flag attached to the packet marking it as a duplicate. Once the PUBREC has been received at the client, the client will cease attempting redelivery and can discard the original message to be published. The PUBREC packet however, is saved and a PUBREL or publish-released packet is sent to the broker. This is to let the broker know that the client is freeing up any identifiers previously used by this publish transaction. Once this PUBREL message arrives with the broker, a final PUBCOMP or publish-complete message is relayed back to the client so that it can drop the PUBREC packet from memory as well.

***Evaluation:***

MQTT was easy to set up and required minimal configuration from the outset. There are many options available for configuration, but the defaults provided everything required of this project except for opening an additional port for Web Socket traffic which was required by the React-Native framework used for the mobile application.

Due to a lack of documentation about the required configuration to run Web Sockets in a Windows deployment of the Mosquitto broker selected for the project, the initial plan was to run the Broker on a Raspberry Pi in a Linux based environment. Because MQTT is so well adopted for IoT devices which are well established to communicate with devices like the Raspberry Pi, Web Socket support is better supported and documented for Linux based systems. However, during development it was eventually noted that better documentation had become available which would allow for certain installed versions of MQTT to handle Web Socket traffic with just a single addition to the config file to add an additional port for the traffic as shown in Figure 9: Mosquitto Broker Configuration.

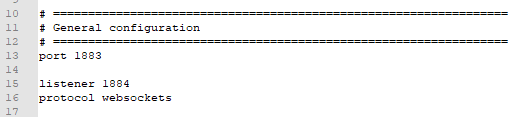


Figure 9: Mosquitto Broker Configuration

Overall the Mosquitto Broker selected worked fairly seamlessly throughout development and required very little in the way of maintenance. Some additional reading was required in order to make sure when running the broker it would use a custom configuration as shown above and to ensure that the console window would then operate in -v verbose mode and show log information to allow proper debugging and testing as show in Figure 10: MQTT Broker QoS 2 Capture.

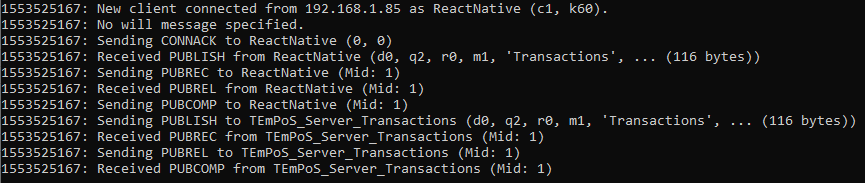


Figure 10: MQTT Broker QoS 2 Capture

The Mosquitto Broker version used for this project is 1.5.4 which is available online from the official Eclipse website.

## Testing

### QoS 00 Test 001



### QoS 00 Test 002



### QoS 00 Test 003



### QoS 01 Test 001



### QoS 01 Test 002



### QoS 01 Test 003



### QoS 02 Test 001



### QoS 02 Test 002



### QoS 02 Test 003



### QoS Testing Code Snippets

The following images show the onConnect() function from the React-Native mobile client application. More specifically, within this function lies the publish() method call which is responsible for sending the message to the Mosquitto MQTT Broker. This is of the following form:

The first argument is the topic to publish to in this case passed as a variable representing a String value. The topic used throughout testing was “Transactions”.

The second argument is the message you wish to send to the specified topic, again in this case represented by the publishMessage variable which was simply a json String representing the transaction data of the form that the server would be able to parse.

The third argument passed to the publish method is an Integer value representing the QoS to use for that message; 0, 1 or 2. This is the variable that changes throughout testing:

Figure 16: onConnect() QoS 0

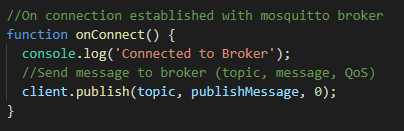


Figure 17: onConnect() QoS 1

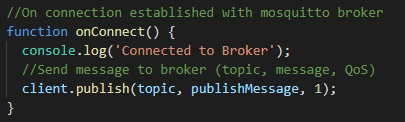
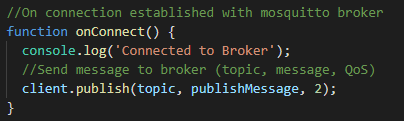
****

Figure 18: onConnect() QoS 2

****