ABSTRACT

KAYAL, PARIDHIKA. A Comparison of IoT Application Layer Protocols Through A Smart Parking Implementation. (Under the direction of Dr. Harry Perros).

Several IoT protocols have been introduced inorder to provide efficient communication for resource-constrained applications. However there are few guidelines to help an application developer choose an appropriate communication protocol inorder to achieve desired goals related to performance, reliability, security, scalability, energy efficiency, and expressiveness. To address this issue, we studied and evaluated the following communication protocols: CoAP, MQTT, XMPP, WebSocket, SMQ and CoSIP. For this, we implemented a smart parking application using open source softwares for CoAP, MQTT, XMPP and websockets and analyzed their performance by varying the demand for this application. We further analyzed various application requirements that may help a user to choose the best protocol for his/her application.

© Copyright 2017 Paridhika Kayal

All Rights Reserved

A Comparison of IoT Application Layer Protocols Through A Smart Parking Implementation

by
Paridhika Kayal

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the degree of
Master of Science

Computer Networking

Raleigh, North Carolina
2017

APPROVED BY:

_______________________________	_______________________________
Dr. Harilaos Perros Committee Chair	Dr. Khaled Harfoush

____
Dr. Muhammad Shahzad

DEDICATION

To my parents.

BIOGRAPHY

Paridhika Kayal was born on the 25th of May, 1989 in Ujjain, India. She was awarded the Bachelor's degree in Computer Science Engineering in 2012 by International Institute of Information Technology Hyderabad, India. She has worked as Software Developer at Ivy Comptech Pvt. Ltd., India for a period of 2 years and as Associate Application Developer at Service Now Inc., India for a period of 1.5 years. She is currently a graduate student in the Department of Computer Science pursuing MS in Computer Networking and Telecommunication at North Carolina State University, Raleigh. She has worked for Amazon.com Inc, Seattle, WA as part of her Master's degree for a period of 3 months.

ACKNOWLEDGEMENTS

First and foremost, I thank my parents, Arun and Pratibha Kayal for their unconditional love and support throughout the years. Words cannot begin to describe the ways in which they have encouraged and inspired me. I would never have been able to accomplish this work had it not been for their unwavering faith in my abilities.

I thank my advisor, Dr. Harry Perros for guiding me in this work. This thesis would not have been possible without his help. I also thank him for teaching me Queueing theory - CSC579 which was one of the most enjoyable course of my Master's degree. I thank my committee members, Dr. Khaled Harfoush and Dr. Muhammad Shahzad for their valuable suggestions and taking time to evaluate my work.

I thank my friend, Anisha Dey and brother, Vatsal Kayal for always being there for me and for egging me on during stressful times. Lastly, but most importantly, I thank my boy friend Abhitesh Singh for being my pillar of strength. He has been my greatest source of encouragement throughout this degree. He has had more sleepless nights than me in helping me with this work and for that, I shall always be grateful.

TABLE OF CONTENTS

LIST OF TABLES vii

LIST OF FIGURES viii

Chapter 1 INTRODUCTION 1
- 1.1 Contribution 2
- 1.2 Organization of the Thesis 2

Chapter 2 Related Work 4

Chapter 3 Protocols Overview 7
- 3.1 CoAP:- Constrained Application Protocol 7
  - 3.1.1 Request/Response Model 7
  - 3.1.2 Advantages 8
  - 3.1.3 Weaknesses 9
- 3.2 MQTT:- Message Queuing Telemetry Transport 9
  - 3.2.1 Publisher/Subscriber Model 10
  - 3.2.2 Advantages 10
  - 3.2.3 Weaknesses 11
- 3.3 XMPP:- eXtensible Messaging and Presence Protocol 11
  - 3.3.1 Client/Server Model 12
  - 3.3.2 Advantages 12
  - 3.3.3 Disadvantages 12
- 3.4 WebSockets 13
  - 3.4.1 Architecture Overview 13
  - 3.4.2 MQTT over Websockets 13
  - 3.4.3 Advantages and Disadvantages 14
- 3.5 SMQ:- Simple Message Queue 14
  - 3.5.1 Architectural overview 14
  - 3.5.2 Advantages 15
- 3.6 CoSIP:- Constrained Session Initiated Protocol 16
  - 3.6.1 Architecture Overview 16
  - 3.6.2 Advantages 17

Chapter 4 Smart Parking System 19
- 4.1 Background 19
- 4.2 Architecture and Design 19
- 4.3 Assumptions 20
- 4.4 Implementation Details 20
- 4.5 Performance Evaluation 22
  - 4.5.1 CoAP 22
  - 4.5.2 MQTT 26
  - 4.5.3 XMPP 29

4.5.4 Websockets . . . 34
4.5.5 Protocol Comparison . . . 38

Chapter 5 Choosing the appropriate protocol . . . 46
5.1 Architectural Requirements . . . 47
5.1.1 Request/Response vs Publisher/Subscriber . . . 47
5.1.2 Device Solution Vs Browser Solution . . . 47
5.2 Discovery and Addressing . . . 47
5.3 Scalability . . . 47
5.3.1 Types of traffic . . . 48
5.3.2 Estimated number of devices . . . 48
5.4 Latency and Throughput Requirements . . . 49
5.5 Application level Qos . . . 49
5.6 Identification and Authentication . . . 50
5.7 Security . . . 50

Chapter 6 Summary and Future Work . . . 52

BIBLIOGRAPHY . . . 54

LIST OF TABLES

Table 4.1 CoAP Response times 23
Table 4.2 MQTT Response times 27
Table 4.3 XMPP Response times 32
Table 4.4 MQTT over Websockets Response times 36

Table 5.1 IoT Communication Protocols Comparison 51

LIST OF FIGURES

Figure 3.1 CoAP Message Format 8
Figure 3.2 MQTT Message Format 10
Figure 3.3 CoSIP Message Format 17

Figure 4.1 Mean Response Time vs libcoap Server Utilization 25
Figure 4.2 Mean Response Time vs Mosquitto broker Utilization 29
Figure 4.3 Mean Response Time vs Openfire Server Utilization 34
Figure 4.4 Mean Response Time vs HiveMQ Server Utilization 38
Figure 4.5 Mean Response Time for the 4 protocols vs Server Utilization 39
Figure 4.6 Mean Response Time vs Server Utilization a closer view 40
Figure 4.7 Mean Response Time for CoAP and MQTT vs Server Utilization 41
Figure 4.8 Mean Response Time for XMPP and MQTTWS vs Server Utilization 42
Figure 4.9 Mean Response Time vs Server Utilization for DELETE Clients 43
Figure 4.10 Mean Response Time vs Server Utilization for PUT Clients 44
Figure 4.11 Mean Response Time vs Server Utilization for GET Clients 45

CHAPTER

1 INTRODUCTION

An IoT application typically involves a large number of sensors and gateways deployed and interconnected. The sensor nodes measure the physical environment and send the data to a gateway node. The gateway aggregates the data from various sensor nodes and then sends it to a server/broker. Meanwhile, clients that are interested to receive sensor data connect to the server to obtain the data. The integration of sensor devices into the internet requires IP compatible protocol stacks which is bandwidth-efficient, energy-efficient and capable of working with limited hardware resources. The lack of optimized application protocols for sensors can cause performance degradation in terms of bandwidth usage and battery lifetime for wireless sensors.

The Internet of Things is a big place, with room for many application protocols suitable for sensors. The fundamental goals of all protocols differ, the architectures differ, and the capabilities differ. All of these protocols are critical to the (rapid) evolution of the IoT. It is important to understand the class of use that each of these protocols addresses and choose the one for an application carefully, especially when key system requirements such as performance, quality-of-service, interoperability, fault tolerance and security are taken into account.

The MQTT protocol is used for collecting device data and communicating it to servers, XMPP is used for connecting devices to people. It can support distributed message exchanges between processes on a single node (Intra Device). However XMPP was not designed for high performance message exchanges within the same mode and is more appropriate when used to communicate between nodes or with internet based applications. CoAP is a specialized web transfer protocol

for use in constrained nodes and networks. It can be used for data collection in systems that do not require very high performance, real-time data sharing or real-time device control. In many cases data is collected for subsequent “offline” processing. SMQ provides a browser solution for applications using HTTP. This can be very inefficient is terms of request processing time and resources consumed. The WebSocket (WS) standard provides bi-directional Web communication and connection management. Using Websockets is a good IoT solution if the devices can afford the Websocket payloads. Some of the above protocols have already gained traction, while some others like CoSIP are fast picking up in the IoT landscape. All these protocols are positioned as real-time publish-subscribe IoT protocols, with support for millions of devices. Depending on how you define “real time” (seconds, milliseconds or microseconds) and “things” (WSN node, multimedia device, personal wearable device, medical scanner, engine control, etc.), the protocol selection for an application is critical.

1.1 Contribution

There are three accepted techniques for evaluating and analyzing networks: analytical methods, computer simulation and practical implementation. In this thesis, we used the last method since it is more realistic. Additionally, the proportion of algorithms that are analyzed through practical evaluation is comparatively low, possibly due to the deployment cost, time required, broad diversity and application dependence of WSNs. As a result, event based simulation is currently the most widely adopted method of analyzing WSNs, allowing the rapid evaluation, optimization and adjustment of proposed algorithms and protocols. In this thesis we used a smart parking application as a testbed in which we have implemented the open-source IoT protocols designed for low powered devices. We particularly implemented the testbed using CoAP, MQTT, XMPP and WebSockets and compared their mean response time for different server utilizations. Based on the obtained performance results, we also tried to identify the protocol that will serve best, based on the specific application features and requirements.

1.2 Organization of the Thesis

In chapter 2, we review the literature on IoT communication protocols and their performance. Various types of IoT applications and their requirements are also described.

In chapter 3, the features, advantages and weaknesses of IoT application layer protocols, the CoAP, MQTT, XMPP, WebSockets, SMQ and CoSIP are described.

In chapter 4, the architecture of a smart parking testbed is first described. Then, numerical results are given for the evaluation and comparison of the CoAP, MQTT, XMPP, WebSockets protocols.

In chapter 5, different application requirements are discussed. These requirements are analyzed to help an application developer decide which protocol will suit best for a given application.

Finally, in Chapter 6, we summarized the conclusion and discussed future research directions.

CHAPTER

2 RELATED WORK

Several surveys have provided description and comparative analysis of IoT application layer protocols without providing real data measurements, see [Jaf14], [Kar15], [Bab16], [YS16]. Authors in [YS16] focused on the evaluation of IoT application layer Protocol: XMPP, MQTT, CoAP, RESTFUL, DSS, AMQP and WebSocket protocol in term of architecture, communication model, security, and achieving QoS. They also addressed the weaknesses and strengths for each protocol. The paper provides comprehensive comparison between the existing protocols, so it can help developers and researchers know how to select the suitable protocol for the existing environment and applications. The article in [Kar15] provides a description of the key protocols that are being used today in IoT, particularly CoAP, MQTT, HTML 5s, XMPP and Websocket. The authors argued about suitability of these protocols for the IoT by considering reliability, security, and energy consumption aspects without any statistical comparisons between the protocols. The authors in [Bab16] analyzed latencies induced by employing different communication protocols, such as, HTTP long polling, HTTP streaming, and both TCP socket and WebSocket, as well as different message encodings like XML, JSON, and platform-supported binary formats. They also compared the latencies of sensor data propagation, and the message throughput rate achieved by IoT application layer protocols MQTT, AMQP, XMPP, and DDS. The authors finally concluded that MQTT is the most appropriate messaging protocol for a wide set of IoT Web applications.

Several works experimentally tested the most popular IoT application layer protocols, typically comparing two selected protocols. In [Tha14] the authors compared MQTT and CoAP by creating a

middleware component in order to perform testing. They found that MQTT has a smaller latency for smaller packet loss than CoAP, and in contrast, higher latency than CoAP for higher packet loss. Two lightweight application protocols, CoAP and MQTT, have been assessed in terms of energy consumption, bandwidth utilization and reliability in [BB13]. According to the result, CoAP is the most efficient in terms of energy consumption and bandwidth usage while MQTT provides high reliability. [DC13] provides a qualitative and quantitative comparison between MQTT and CoAP when used as smartphone applications. The performance analysis showed that CoAP achieves better results both in terms of bandwidth usage and round trip time. This, combined with the CoAP caching feature, makes CoAP a more appropriate choice for the development of efficient applications having the goal of reducing both network utilization and device resource usage. In terms of reliability, MQTT performs better as a consequence of its more sophisticated reliability and congestion control mechanisms. However, a significant difference is only observed when data have to be exchanged very frequently.

In [LS12] the performance of three different XMPP communication techniques in a LAN: (1) HTTP Polling, (2) BOSH, and (3) XMPP sub-protocol for WebSocket, were evaluated. The results suggest that in terms of round trip rates, XMPP sub-protocol for WebSocket performed substantially better than the HTTP Polling and BOSH techniques. In paper [Mij16] CoAP, WebSocket and MQTT were evaluated for their performance in terms of protocol efficiency, strictly related to the overhead, and average Round Trip Time (RTT) with no consideration of increased load. Results show that CoAP achieves the highest protocol efficiency and the lowest average RTT, closely followed by WebSocket. The performance of MQTT protocol strongly depend on the QoS profile. Changing the environment, from a LAN network to a realistic IoT scenario, does not significantly impact the protocol efficiency, but has a considerable influence on the average RTT, which increases by a factor of 2 or 3, depending on the protocol.

In [Mun16] the performance, energy consumption, and resource usage characteristics of IoT protocols including CoAP, MQTT, MQTT-SN, WebSocket, and TCP for varying packet size. The experiments show that MQTT and CoAP protocols are largely affected by the packet size. In particular, CoAP and MQTT start to fragment a packet from 1 kB and 1.5 kB respectively. In particular, in order to recover errors due to packet losses or delays, CoAP employs a straightforward retransmission mechanism using exponential backoff time on top of UDP while MQTT relies on the TCP's error recovery mechanism. The results also showed that CoAP's retransmission mechanism mainly affected the memory usage rather than the CPU usage. In terms of energy efficiency the paper concludes that CoAP, MQTT, and MQTT-SN are considerably vulnerable to network conditions due to the underlying transport protocols (TCP or UDP), frequent packet fragmentation (CoAP) and straightforward retransmission mechanisms (CoAP, MQTT, and MQTT-SN).

All the above surveys identified CoAP, MQTT, XMPP, AMQP, websockets and REST services as the most representative protocols for internet of things. With respect to the literature, few measures are

available on protocol efficiency, strictly related to overhead, RTT, frequent packet fragmentation, QoS and retransmission. To the best of our knowledge there is no performance measurements available on varying load, particularly comparing CoAP, MQTT, XMPP and WebSockets. In this work we take a fully experimental approach by considering a real IoT scenario. We implemented a smart parking application using above mentioned application layer protocols. Being implemented on the same platform, the comparison between protocols is fair and realistic. The considered performance metrics is the average response time, the 99th percentile, and its confidence interval for different server utilizations.

CHAPTER

3

PROTOCOLS OVERVIEW

This chapter discusses the most common application layer protocols that are used in IoT applications. The architecture, features, advantages and weaknesses of various communication protocols are described.

3.1 CoAP:- Constrained Application Protocol

CoAP is the Constrained Application Protocol (CoAP) [She14] was developed by the CoRE (Constrained Resource Environments) IETF group. CoAP was specifically developed to allow resource-constrained devices to communicate over the Internet using UDP instead of TCP. Developers can interact with any CoAP-enabled device the same way they would with a device using a traditional REST-based API. There are more than 30 CoAP open source implementations written in various languages including C, C++, Java, Python, JavaScript, etc.

Although CoAP could be used for refashioning simple HTTP interfaces into a more compact protocol, more importantly it also offers features for M2M such as built-in discovery, multicast support, and asynchronous message exchanges.

3.1.1 Request/Response Model

CoAP is, primarily, a one-to-one protocol for transferring state information between client and server. Clients make requests to servers and servers send back responses. Clients may send GET, PUT, POST

and DELETE resource requests to the server. CoAP defines four types of messages: Confirmable, Non-confirmable, Acknowledgement, Reset. Reliability is provided by marking a message as Confirmable (CON). A Confirmable message is retransmitted using a default timeout and exponential back-off between retransmissions, until the recipient sends an Acknowledgement message. A message that does not require reliable transmission can be sent as a Non-confirmable message (NON). These are not acknowledged, but still have a Message ID for duplicate detection. When a recipient is not able to process a Non-confirmable message, it may reply with a Reset message (RST).

CoAP messages are encoded in a simple binary format. The message format starts with a fixed-size 4-byte header. This is followed by a variable-length Token value, which can be between 0 and 8 bytes long. Following the Token value comes a sequence of zeroes or more CoAP Options in Type-Length-Value (TLV) format, optionally followed by a payload that takes up the rest of the datagram.

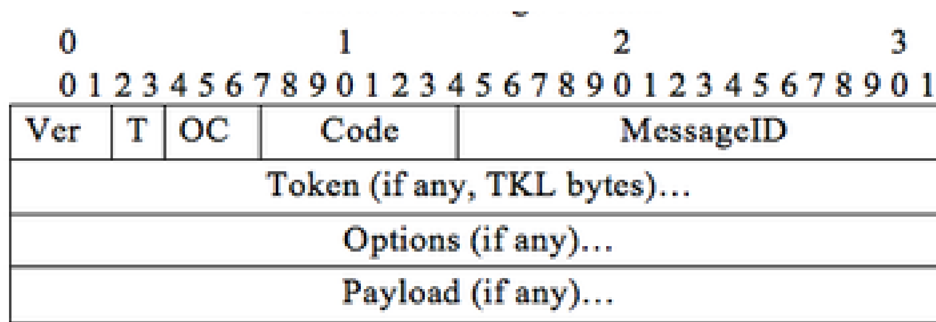

Figure 3.1 CoAP Message Format

3.1.2 Advantages

- Like HTTP, CoAP is a document transfer protocol designed for the needs of constrained devices. CoAP packets are much smaller than HTTP TCP flows. CoAP packets use bit fields to maximize memory efficiency, and they make extensive usage of bitfields mappings from strings to integers to keep the data packets small enough to transport and interpret on-device. CoAP packets are much smaller than HTTP TCP flows. Packets are simple to generate and can be parsed in place without consuming extra RAM in constrained devices.

- Aside from the extremely small packet size, another major advantage of CoAP is its usage of UDP; using datagrams allows for CoAP to be run on top of packet-based technologies like

SMS.

- Application level QoS: Requests and response messages may be marked as “confirmable” or “non-confirmable”. Confirmable messages must be acknowledged by the receiver with an ack packet. Non confirmable messages are “fire and forget” and do not require an acknowledgement

3.1.3 Weaknesses

- Because CoAP is built on top of UDP not TCP, SSL/TLS are not available to provide security. DTLS, Datagram Transport Layer Security provides the same assurances as TLS but for transfers of data over UDP. The default level of encryption is equivalent to a 3,072-bit RSA key. Typically, DTLS capable CoAP devices will support RSA and AES or ECC and AES.

- Another downside of CoAP is that it is a one-to-one protocol. Though extensions that make group broadcasts possible are available, broadcast capabilities are not inherent to the protocol. Arguably, an even more important disadvantage is the lack of a publish-subscribe message queue. There is an "observer" CoAP extension, currently in the IETF standardization process, that aims at defining a CoAP based subscribe/notify service implemented over the basic CoAP REST model.

- CoAP does not define any of the cache-suppressing Cache-Control options that HTTP/1.1 provides to better protect sensitive data.Hence the threat to confidentiality and integrity of request/response data is amplified. Unlike the "coap" scheme, responses to "coaps" identified requests are never "public" and thus MUST NOT be reused for shared caching, unless the cache is able to make equivalent access control decisions to the ones that led to the cached entry.

- Since CoAP runs on UDP which does not involve handshake, a rogue endpoint that is free to read and write messages carried by the constrained network, may easily attack a single endpoint, a group of endpoints, as well as the whole network. Hence CoAP is prone to spoofing and amplification attacks.

3.2 MQTT:- Message Queuing Telemetry Transport

MQTT is ideal for the ioT world of connected devices. MQTT v3.1.1 [BG14] is an OASIS Standard. It is a many-to-many communication protocol for passing messages between multiple clients through a central broker. MQTT has a lightweight packet structure designed to conserve both memory usage and power. Its minimal design makes it perfect for built-in systems, mobile phones and other memory and bandwidth sensitive applications.

3.2.1 Publisher/Subscriber Model

MQTT is a client/server publish/subscribe messaging protocol designed for lightweight M2M communications. It was originally developed by IBM and is now an open standard. MQTT has a client/server architecture, where every sensor is a client and connects to a server, known as a broker. The protocol runs over TCP/IP, or over other network protocols that provide ordered, lossless, bidirectional connections. MQTT is message oriented. Every message is published to an address, known as a topic. Clients may subscribe to multiple topics. Every client subscribed to a topic receives every message published to the topic. MQTT exchange control messages. An MQTT Control Packet consists of up to three parts: Fixed header, Variable header and payload. Figure 3.2 shows the MQTT message format.

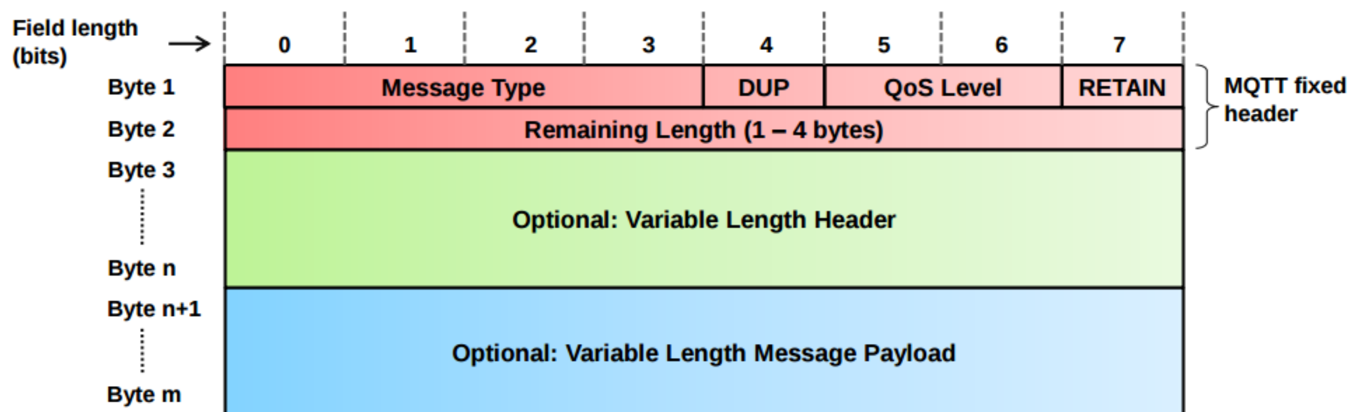

Figure 3.2 MQTT Message Format

3.2.2 Advantages

- Application Level QoS: MQTT supports three quality of service levels, "Fire and forget", "delivered at least once" and "delivered exactly once".
- Last Will And Testament: MQTT clients can register a custom "last will and testament" message to be sent by the broker if they disconnect. These messages can be used to signal to subscribers when a device disconnects.
- Using a long-lived outgoing TCP connection to the MQTT broker, sending messages of limited bandwidth back and forth is simple and straightforward.

- Security: MQTT brokers may require username and password authentication from clients to connect. To ensure privacy, the TCP connection may be encrypted with SSL/TLS.

3.2.3 Weaknesses

Even though MQTT is designed to be lightweight, it has few drawbacks for very constrained devices.

- Every MQTT client must support TCP and will typically hold a connection open to the broker at all times. For some environments where packet loss is high or computing resources are scarce, this is a problem.
- Having an always-on connection limits the amount of time the devices can be put to sleep. If the device mostly sleeps, then another MQTT protocol can be used: MQTT-SN, which works with UDP instead of TCP.
- MQTT topic names are often long strings which make them impractical for 802.15.4.
- Another disadvantage of MQTT is the lack of encryption in the base protocol. MQTT was designed to be a lightweight protocol, and incorporating encryption would add a significant amount of overhead to the connection.

All of these shortcomings are addressed by the MQTT-SN protocol [SCT08], which defines a UDP mapping of MQTT and adds broker support for indexing topic names.

3.3 XMPP:- eXtensible Messaging and Presence Protocol

The Extensible Messaging and Presence Protocol (XMPP) is an application profile of the Extensible Markup Language [XML] that enables the near-real-time exchange of structured yet extensible data between any two or more network entities. XMPP [SA11] is an open protocol standardized by the IETF and supported and extended by the XMPP Standards Foundation. The eXtensible Messaging and Presence Protocol (XMPP) is a TCP communications protocol based on XML that enables near-real-time exchange of structured data between two or more connected entities. XMPP powers a wide range of applications including instant messaging, multi-party chat, voice and video calls, collaboration, lightweight middle ware, content syndication, network management, collaboration tools, file sharing, gaming, remote systems monitoring, web services, cloud computing etc. Out-of-the-box features of XMPP include presence information and contact list maintenance. While both features were originally designed for instant messaging, they have obvious applications for IoT. Due in part to its open nature and XML foundation, XMPP has been extended for use in publish-subscribe systems, again, perfect for IoT applications.

3.3.1 Client/Server Model

XMPP is typically implemented using a distributed client/server architecture, wherein a client needs to connect to a server in order to gain access to the network and thus be allowed to exchange small pieces of structured data (called "XML stanzas") with other entities (which can be associated with other servers). The process involves following steps:

1. Determine the IP address and port at which to connect and open a TCP connection.
2. Open an XML stream over TCP negotiate TLS for channel encryption.
3. Authenticate using a Simple Authentication and Security Layer (SASL) mechanism.
4. Bind a resource to the stream and exchange an any number of XML stanzas.
5. Close the XML stream and then the TCP connection.

Within XMPP, one server can optionally connect to another server to enable inter-domain or inter-server communication. For this to happen, the two servers need to negotiate a connection between themselves and then exchange XML stanzas.

3.3.2 Advantages

- The primary advantage is XMPP's decentralized nature. XMPP works similar to email, operating across a distributed network of transfer agents rather than relying on a single, central server or broker (as CoAP and MQTT do). As with email, it's easy for anyone to run their own XMPP server, allowing device manufacturers and API operators to create and manage their own network of devices. And because anyone can run their own server, if security is required, that server could be isolated on a company intranet behind secure authentication protocols using built-in TLS encryption.
- Simple Authentication and Security Layer (SASL) provides a generalized method for adding authentication support, XMPP uses an XML namespace profile of SASL that conforms to the profiling requirements.

3.3.3 Disadvantages

- One of the biggest flaws is the lack of end-to-end encryption. While there are many use cases in which encryption may not yet be necessary, most IoT devices will ultimately need it.
- Another downside is the lack of QoS. Making sure that messages are delivered is even more important in the IoT world than it was in the instant messaging world.
- The message format used by XMPP is XML, which makes it heavier for power constrain devices.

3.4 WebSockets

WebSocket [FM11], a new tool component for web developers, allows developers to create web apps without the hassles of HTTP. Websockets operate over TCP as an upgrade to a standard HTTP connection allowing for full-duplex, low-latency communication between a server and a client. The WebSocket protocol was standardized by the IETF as RFC 6455 in 2011, and the WebSocket API in Web IDL is being standardized by the W3C. The WebSocket protocol is currently supported in most major browsers including Microsoft Edge, Google Chrome, Internet Explorer, Firefox, Safari and Opera.

3.4.1 Architecture Overview

The WebSocket protocol facilitates the real-time data transfer from and to the server by allowing interaction between a browser and a web server. This is made possible by providing a standardized way for the server to send content to the browser without being solicited by the client, and allowing for messages to be passed back and forth while keeping the connection open. In this way, a bi-directional ongoing conversation can take place between a browser and the server. The communications are done over TCP port number 80, which is of benefit for those environments which block non-web Internet connections using a firewall. The WebSocket protocol specification defines ws and wss as two new uniform resource identifier (URI) schemes that are used for unencrypted and encrypted connections, respectively. Just two WebSocket frame bytes can replace hundreds of HTTP header bytes.

3.4.2 MQTT over Websockets

Websockets [Weba] have very little overhead in terms of bandwidth and latency compared to classic HTTP requests which are needed when using (long) polling. This philosophy of having as little overhead as possible fits very well to MQTT. With MQTT over websockets every browser can be a MQTT device. Due to the publish/subscribe pattern of MQTT, a real time push to a browser occurs when an event, literally anywhere in the world occurs, as long as the browser's user is subscribed to the correct topic.

When using the term MQTT over WebSockets, it means that a MQTT message is encapsulated within a websocket packet. Messages over websockets are sent in frames. These frames have only 2 byte overhead. The whole MQTT message (with all its headers and payload) is now sent with the websocket frame. WebSockets are suitable as transport for MQTT because the communication is bi-directional, ordered and lossless (which is essentially because WebSockets also leverage TCP).

3.4.3 Advantages and Disadvantages

- If we are creating an app that needs constant real-time updates (chat, interactive streaming media, live multiplayer games, etc) then web sockets is the best solution.

  If you are creating an app that only needs to update periodically or it is based on user events, then web sockets is a less economical choice. Web sockets keeps the connection open on the server for the duration of the time the user is interacting with the page. This increases the demand on the server, and means that we will always have to scale out.

- Websockets provide encrypted end-to-end communication and advanced authentication and authorization features. It also provides flexibility to enable specific security features for your concrete use.

  SSL/TLS comes with an additional overhead in terms of bandwidth and CPU. It can make a huge difference in scenarios where small bandwidth usage is key.

3.5 SMQ:- Simple Message Queue

SMQ [Smqb] is an easy to use IoT (M2M) publish/subscribe protocol designed and optimized for embedded systems providing instantaneous edge node connectivity. The solution allows resource constrained real-time control, analysis, and updates.

3.5.1 Architectural overview

The following are some of the features of SMQ, for further details, see [Smqa].

- SMQ is an easy to implement, maintain and use message queues architecture solution that follows the publish subscribe broadcast design pattern.
- Designed for resource constrained edge nodes like sensors and actuators.
- Clients initiate the IoT communication with the SMQ Broker by use of a standard web URL, which is either HTTP for non secure or HTTPS for secure communication. The device to broker communication is established in a similar fashion to how WebSockets are initiated. Once a connected session occurs, the SimpleMQ Client may then publish and subscribe to topic names.
- A topic name can be any string value, but it is typically structured in a hierarchical fashion, which is equivalent to that of a file system hierarchy found in a UNIX like environment. Topic names are then translated into numbers by the server, thus enabling fast processing of

messages by using lookup tables and values stored on the server. The client subscribes to a topic and the server responds with a topic ID that identifies the topic channel.

- Each client is also assigned a unique ID by the broker known as the ephemeral topic ID. The ephemeral topic ID is used in a unique way such as "publisher's address" by subscriber, thus enabling each subscriber the ability to send messages directly to a device by using the ephemeral topic ID. The ephemeral topic ID also enables an application to simulate a remote procedure (RPC) call.

3.5.1.1 Device Solution

SimpleMQ Client is provided in C Source Code and is built for a direct device interaction with TCP/IP. The communication client is designed and utilized to preserve resources, which are typically vital for small constrained micro controllers. Devices may authenticate themselves by using standard credentials such as username/password, or key values by example. The SimpleMQ client while operating in a (non-secure) mode still allows for an easy use of hash based authentication, which is useful to establish a secure encrypted password connection to prevent eavesdropping.

3.5.1.2 Browser Solution

The IoT Message Queue protocol for a browser is channeled over a WebSocket connection. All major browsers support WebSockets as a persistent real-time communication protocol, which in turn enables JavaScript code running in a browser to interact with a WebSocket enabled server. The browser solution consists of the SimpleMQ-JS JavaScript library that enables JavaScript applications to subscribe to and publish to topics.

3.5.2 Advantages

- The SMQ broker provides easy segmentation of products/customers by enabling an unlimited number of broker instances running in the server. Each Message Queue entry (URL) in the server is associated with one broker instance, thus the number of instances that can be installed in the server at any given time are only limited to any inherent memory limitations.
- Enables end-to-end IoT security and military grade encryption by providing SOCKS and HTTPS proxy support.
- Provides direct addressing without Firewall/NAT Hassle, hence enables private IP to private IP communication.

- SMQ Broker enables IoT SSL termination.
  If we have an external browser using secure Web Sockets sending a message to non-secure devices utilizing the SimpleMQ Client, then the SMQ Broker decrypts the message received from the browser and then forwards the unencrypted result to the non-secure devices. This communication also works in reciprocal pattern where a non secure client is able to send messages to a secure client.

3.6 CoSIP:- Constrained Session Initiated Protocol

CoSIP [Cos] is a constrained version of the Session Initiation Protocol (SIP), whose intent is to allow constrained devices to instantiate communication sessions in a lightweight and standard fashion and can be adopted in M2M application scenarios. Session instantiation can include a negotiation phase of some parameters which will be used for all subsequent communication. The proposed CoSIP is a binary protocol which maps to SIP, similarly to what CoAP does to HTTP. CoSIP can be adopted in several application scenarios, such as service discovery and publish/subscribe applications. The CoSIP protocol is intended to minimize the amount of network traffic, and therefore energy consumption, targeted for IoT scenarios.

3.6.1 Architecture Overview

In both constrained and non-constrained environments there are many applications that may require or simply may obtain advantages by negotiating end-to-end data sessions. In this case the communication model consists of a first phase in which one endpoint requests the establishment of a data communication and, optionally, both endpoints negotiate some communication parameters of the subsequent data sessions, see [Cir13]. A CoSIP message can be virtually seen as a standard SIP message, consisting of one request line or one status line (depending if the message is a request or a response), followed by a sequence of SIP header fields, followed by a message body, if present.

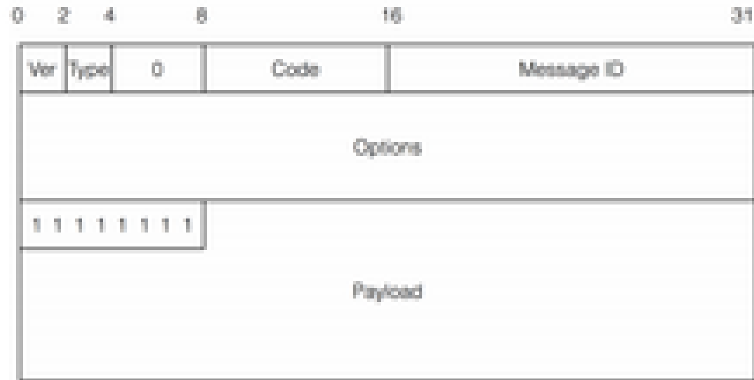

Figure 3.3 CoSIP Message Format

3.6.2 Advantages

- The exchange of parameters like transfer protocols, data formats, endpoint IP addresses and ports, encryption algorithms and keying materials, and other application specific parameters may be useful for both client-server or peer-to-peer applications, regardless the data sessions evolve or not according to a request/response model. The main advantage is that all such parameters, including possible resource addressing, may be exchanged in advance, while no such control information is required during data transfer. The longer the data sessions, the more the advantage is evident with respect to a per-message control information. Also in the case of data sessions that may vary formats or other parameters during time, such adaptation may be supported by performing a session renegotiation.

- CoAP Service Discovery: CoSIP allows smart objects to register the services they provide to populate a CoSIP Registrar Server, which serves as a Resource Directory.

- Session Establishment: A session is established when two endpoints need to exchange data. CoSIP allows the establishment of session in a standard way without binding the session establishment method to a specific session protocol. For instance, CoSIP can be used to negotiate and instantiate a RTP session between constrained nodes. Once a session has been established, the data exchange between the endpoints occurs (logically) in a peer-to-peer fashion.

- Subscribe/Notify Applications: IoT scenarios typically involve smart objects which might be battery-powered devices. It is crucial to adopt energy-efficient paradigms, e.g. OS tasks,

application processing, and communication. In order to minimize the power consumed, duty-cycled smart objects are adopted. Sleepy nodes, especially those operating in LLNs, are not guaranteed to be reached, therefore it is more appropriate for smart objects to use a subscribe/notify, also denoted as publish/subscribe (Pub/Sub), approach to send notifications regarding the state of their resources, rather than receive and serve incoming requests. Such a behavior can be achieved by leveraging on the inherent capabilities of SIP, and therefore of CoSIP.

CHAPTER

4

SMART PARKING SYSTEM

4.1 Background

Parking problems are common place in most major cities. Searching for a vacant parking space in a congested area or a large parking lot and preventing auto theft are major concerns in our daily lives. Recently, researchers turned to applying technologies for efficient parking management to effectively satisfy the needs and requirements of existing parking hassles. Authors in [Chi07], [Zha12], [Lu09], [Sri09], [WH11] propose various smart parking schemes for minimizing the time consumed to find a vacant parking lot, real-time information rendering, and smart reservation mechanisms. In this thesis, we use a smart parking testbed to evaluate and compare the communication protocols CoAP, MQTT, XMPP and WebSocket given in the previous chapter. The model makes it easier for incoming clients to reach the specified position in parking to park their cars and helps to avoid the overhead of moving around in the parking looking for empty slots. This chapter is organized as follows. First the architecture, underlying assumptions and implementation details of the testbed are given, followed by the results of the response time of these protocols for different server utilization.

4.2 Architecture and Design

We considered a car parking with a sensor attached to every parking position. These sensors sense whether a parking spot is occupied or not. The architecture consists of a server/client model. Every

sensor has a client that sends status information (occupied/empty) to the central server. These sensors can send either a PUT or a DELETE message to update their status at the server. A PUT message is sent when the spot gets occupied by a car. The DELETE message is sent when a car leaves and the spot becomes free.

Also, there is a sensor is attached to every car which can interact with the server of the car parking. As a car arrives at the parking lot, this sensor sends a GET request to reserve an empty space in the parking. If there is an available spot, the server reserves it and send the exact location of the reserved slot back to the car sensor. Cars can also come and park at an empty position randomly at different times without requesting the server to reserve a spot. For all such cars arriving to the parking without reserving a spot through the server, the sensor at parking spot sends a PUT request to the server to send the information of its occupied status. The server stores the information of all the occupied and empty positions in the parking, so that it can serve requests from new clients and provide them the first available empty spot.

4.3 Assumptions

- The cars arrival rate is assumed to follow a Poisson distribution. Let λ_1 and λ_2 be the arrival rates of reservations and the arrival rate of cars entering the parking lot without reservation respectively.
- The time a car stays at a parking spot follows an exponential distribution with a mean $1/\xi$.
- The length of communication protocol server queue is assumed to be large enough to accommodate all messages pending execution. That is, messages do not get blocked upon arrival at the server.
- The communication between clients and server is assume to be fully reliable with zero packet loss.
- Each sensor is assumed to have a unique mac address in order to identify itself.

4.4 Implementation Details

In order to implement the testbed, we considered a parking space of 100 cars as a 2D array of 10x10 with a sensor attached at every location. The server stores the status of every slot as either OCCUPIED or EMPTY. The simulation runs three client threads namely PUT, DELETE, and GET for each of the three types of request described above. The GET thread generates client request for reservation according to a Poisson distribution with rate λ_1. The PUT thread generates messages due to cars occupying a parking slot without going through the reservation process, according to a Poisson

distribution with rate λ_2. The DELETE thread generates messages due to cars departing from the parking lot after staying for an exponentially distributed time with $1/\xi$. The server synchronizes the parking map according to the requests it receives. The three client threads and the server are all run on the same machine, which is an Ubuntu 14.04 64 bit system with Intel Core i5-5200U CPU @ 2.20GHz * 4. As a result the propagation delay is zero.

The service time for a single GET, PUT and DELETE message coming to an idle server was first calculated. Let $1/\mu_1$, $1/\mu_2$ and $1/\mu_3$ be the time required by the idle server to process single GET PUT and DELETE requests respectively. These service times were measured by issuing a single request to an idle server. The testbed thus can be seen as consisting of 2 queueing systems. One is the parking lot consisting of 100 servers with arrival rate $\lambda_1 + \lambda_2$ and the service time of every server equal to the mean waiting time of a car in parking lot ie. $1/\xi$. The above model is a M/M/100 queueing model and can be evaluated using the Erlang B loss model. Let η be the rate of departure of cars from the parking lot, then η can be calculated as $\lambda_1 + \lambda_2$ (1- P_b), where P_b is the blocking probability that a car will be blocked due to the parking lot being full. The other queueing model depicts the server as implemented with the given protocol where GET, PUT & DELETE messages queue up to get processed. Inorder to have a stable system we have $\lambda_1/\mu_1 + \lambda_2/\mu_2 + \eta/\mu_3 < 1$.

We fixed a value of $1/\xi$ for each protocol, then used the Erlang-B formula with 100 servers to calculate all possible combinations of $\lambda_1 + \lambda_2$ so that$P_b = 0$. In this case $\eta = \lambda_1 + \lambda_2$. Subsequently, we chose a pair of (λ_1, λ_2) that corresponds to 20%, 40%, 60% and 80% utilization of the server. We could have used a combination of (λ_1, λ_2), since we can calculate η, and thus the total arrival rate to the server. Then, vary (λ_1, λ_2) so that to obtain a given server utilization. However, this will not make any difference since we derive the experiment using the server utilization.

The experiments are run as follows. The PUT and GET threads issue messages with an exponential inter-arrival time of $1/\lambda_1$ and $1/\lambda_2$ respectively. These inter-arrival times are generated on demand from an exponential distribution with the appropriate mean ($1/\lambda_1$, $1/\lambda_2$).

Once a car occupies a position, its parking time is generated from an exponential distribution with a mean $1/\xi$. The departure times is placed into an event list which is kept sorted in an ascending manner. The next departure to occur is the one at the top of the event list. We note that the simulation clock is the real clock. For a given protocol, each experiment corresponds to a given utilization. The chosen values of (λ_1, λ_2) were such that they were as close as possible, ie $\lambda_1/\lambda_2 \sim 1$. This can be varied, but we do not expect difference in results for a given utilization.

We ran each experiment for 1000 GETs and 1000 PUTs, which creates 2000 DELETE messages. We measured the amount of time it takes to process each message including waiting time in the protocol server queue. (Recall that the propagation delay is zero.) These measurements were done at thread level by starting the clock as soon as a request is sent to the server, ie., the time when a process tread starts, and noting the time when it arrives back after being processed at the server. The difference in time is stored as the response time for that message. Consequently, we obtained

three separate samples of response times, 1000 GET samples, 1000 PUT samples and 2000 DELETE samples. For each sample we calculated the mean, standard deviation ant the 99th of the response time. The three samples were also combined to calculate similar overall statistics. The confidence interval reported in the tables below is for the mean response time. The statistical results and graphs for the simulation using each protocol, are presented in the next section. We note that the unit of time is seconds.

4.5 Performance Evaluation

The protocols evaluated are CoAP, MQTT, XMPP and Websockets. SMQ requiring different IP addresses for every client as it is based on HTTP requests and CoSIP being novice to the IoT world, were not evaluated.

4.5.1 CoAP

4.5.1.1 Software Used

- **libcoap:** C-Implementation of CoAP [Coaa]. libcoap is designed to run on embedded devices as well as high-end computer systems with POSIX OS. The library provides the core functionality for development of resource-efficient CoAP servers and clients, including resource observation and block-wise transfer.

4.5.1.2 Implementation Notes

Using this library we implemented the smart parking server to handle GET (reservation for cars), PUT (random walk-in without reservation) and DELETE (Departure from parking spot) commands accordingly. We implemented a test-server application that uses various server-side features of libcoap to handle different client commands. CoAP libcoap library provides a coap-client which is a wget-like tool to generate simple requests for retrieval and modification of resources on a remote server. We used coap-client to send the 4 commands (get, put, post and delete) to interact with my server. Below is an example of the commands used to send a coap-uri request for delete message using the libcoap coap-client.

```
./coap-client -m delete coap://[::1]/location -e "Message to be sent"
coap-URI = "coap:" "//" host [ ":" port ] path-abempty [ "?" query ]
```

The installation guide and parking implementation code with obtained results can be downloaded from github [Coab].

Table 4.1 CoAP Response times

20%	Service Time	Arrival Rate	Mean Response Time	99th Percentile	Standard Diviation	Confidence Interval
Delete	0.058	2.3	0.1338427	0.3553459	0.066598	0.002919
Put	0.058	1.1	0.1190230691	0.340153	0.054588	0.0033834
Get	0.00225	1.2	0.12446	0.520019	0.106897	0.0066
Overall		4.6	0.127793	0.403762	0.07655	0.00237
40%						
Delete	0.058	4.53998	0.143355	0.54884	0.08846	0.003877
Put	0.058	2.26999	0.123653	0.39098	0.0593	0.003675775
Get	0.00225	2.26999	0.07517	0.424652	0.0809569	0.00501776
Overall		9.07996	0.12138	0.48873	0.084895	0.00263
60%						
Delete	0.058	6.8	3.2897958	9.38536	2.496886	0.10943
Put	0.058	3.415	3.37656	9.488968	2.5992	0.1611
Get	0.00225	3.385	0.193	5.209	0.833	0.0516344
Overall		13.6	2.5373	9.3151	2.61	0.0808
80%						
Delete	0.058	9.29998	4.7728889	9.533411	1.96797	0.08625
Put	0.058	4.29999	4.63973751	9.228873	1.940275	0.12026
Get	0.00225	4.29999	0.1663	5.312	0.7445	0.0461454
Overall		18.59996	3.5879566	9.056353	2.630914	0.081533

4.5.1.3 Results

In Table 4.1 we summarize the results for the CoAP protocol. The service time in Table 4.1 is the time taken by an idle server to process a single DELETE, PUT and GET request. The service time for a DELETE request is found to be the same as that of PUT request. But the service time of GET request is significantly less compared to the other two. The three commands belong to three different classes and the libcoap server library handles them differently. Because of this, the commands have different priorities and different service times.

From the results in Table 4.1 we can see that as the server utilization is increased, the mean response time of DELETE and PUT also goes up as expected. But in case of the GET, the response time decreases as the utilization increases from 20% to 40%. This can be explained by the fact that CoAP endpoints cache responses in order to reduce the response time and network bandwidth consumption on future, equivalent requests. The goal of caching in CoAP is to reuse a prior response message to satisfy a current request. In some cases, a stored response can be reused without the need for a network request, reducing latency and network round-trips; a "freshness" mechanism is used for this purpose. Even when a new request is required, it is often possible to reuse the payload of a prior response to satisfy the request, thereby reducing network bandwidth usage; a "validation" mechanism is used for this purpose [She14].

When a response is "fresh" in the cache (ie. the requests comes fast enough before the CoAP server clears its buffer), it can be used to satisfy subsequent requests without contacting the origin server, thereby improving efficiency. As mentioned above, CoAP server treats the three processes differently and returns different response codes for each of them. The response code returned for a PUT request is either "2.01 Created" or "2.04 Changed" and both of them are a non-casheable response codes. Similarly for a DELETE process the response code returned is "2.02 Deleted" which is again a non-casheable response code. While the response code for GET process is "2.05 Content" which is a casheable response code. Hence the response time for each individual GET process includes the time to process the request and return the response code plus the time to write the response in cache. So when the arrival rate for GET messages increases, the server skips some of the cache writes for new similar codes. Thus the mean response time for GET messages reduces as the arrival rate increases, while PUT and DELETE process remain unaffected as their is no concept of cashing.

The response time again increases as the server utilization increases to 60%, this can be explained by the fact that initially the CoAP server provides a buffer of size k bytes to handle client requests and as the number of clients connecting to the server increases, it increases the buffer size which involves buffer reallocation and thus it gets added up to the response time. Further as the utilization increases to 80% we observe a similar decrease in the mean response time because of making use of cached responses with a larger buffer this time. The 99th percentiles are quite high in relation to the

mean, and they suggest a long tail of the response time distribution.

Figure 4.1 shows the plot of the mean response times and confidence interval for different server utilizations of the libcoap server.

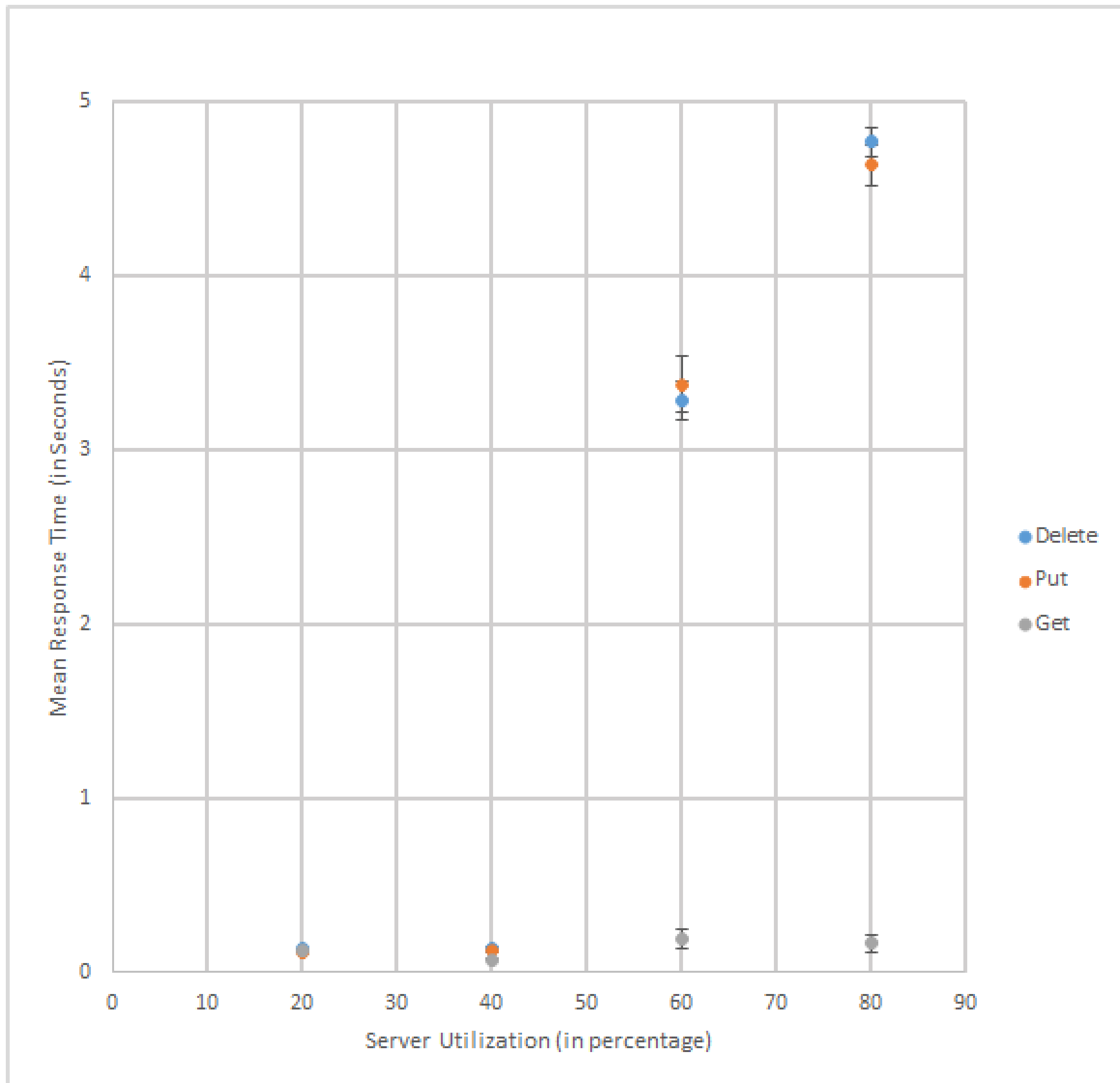

Figure 4.1 Mean Response Time vs libcoap Server Utilization

4.5.2 MQTT

4.5.2.1 Software Used

- **Mosquitto Broker:** Eclipse Mosquitto is an open source (EPL/EDL licensed) message broker that implements the MQTT protocol versions 3.1 and 3.1.1. Mosquitto is an iot.eclipse.org project [Mqta].

4.5.2.2 Implementation Notes

The mosquitto message broker provides a MQTT server which can handle publish and subscribe messages sent from the clients. It provides two types of clients “mosquitto_pub“ and “mosquitto_-sub“ to send publish and subscribe commands to the server respectively to any topic. Below is an example.

./client/mosquitto_pub -t “topic“ -m “Message to be sent”

./client/mosquitto_sub -C 1 -t “topic“

The functionality of the DELETE and PUT messages is achieved by publishing the request to the server using the mosquitto_pub command. While the functionality for GET message is achieved by client subscribing for the empty location information as mqtt topic and the server publishing back the empty location in the parking lot to the client. MQTT normally keeps the subscriber connection on until the client decides to disconnect but the optional parameter “-C 1 “ allows subscriber to disconnect automatically after receiving one publish. So as soon as the client receives one empty location from the server, it gets disconnected. The installation guide and parking implementation code with obtained results can be downloaded from github [Mqtb].

Table 4.2 MQTT Response times

20%	Service Time	Arrival Rate	Mean Response Time	99th Percentile	Standard Diviation	Confidence Interval
Delete	0.0015	50	0.1533	9.2662	1.118	0.049
Put	0.0015	25	0.1890387	10.6155	1.317957	0.081688
Get	0.0035	25	0.3679275	11.22556	1.901649	0.117865
Overall		100	0.215888	10.809685	1.404	0.0435
40%						
Delete	0.0015	100	0.21534	8.72495	1.035599	0.0543
Put	0.0015	50	0.209558	10.024133	1.2832538	0.079537
Get	0.0035	50	0.36109	10.1665	1.6582	0.10277627
Overall		200	0.2176668	9.519756	1.2817	0.03972
60%						
Delete	0.0015	150	0.19142	9.256092	1.296766	0.056833
Put	0.0015	75	0.1623244	7.4146459	1.07472	0.0666
Get	0.0035	75	0.487593	11.25258	2.11868	0.13132
Overall		300	0.2582	10.514	1.506465	0.04669
80%						
Delete	0.0015	200	0.191129	9.170857	1.193	0.05228557
Put	0.0015	100	0.189748	9.2344	1.130123	0.07
Get	0.0035	100	0.5064	10.6838	2.66	0.16491
Overall		400	0.2696	9.73647	1.6791	0.052036

4.5.2.3 Results

The results are summarized in Table 4.2. We can see that the service time required for the DELETE and PUT messages are equal while it is higher for GET messages. According to our implementation for PUT and DELETE commands, both of them sends a mosquitto_pub message to the mosquitto server and hence the server does the same amount of work to serve both types of clients. However to execute a GET message, from our implementation it is clear that it involves 2 commands, first the GET client subscribes to the server using mosquitto_sub and then the server publishes the empty slot by sending mosquitto_pub message and disconnecting the client. The 99th percentiles are quite high in relation to the mean, and they suggest a long tail of the response time distribution.

From Table 4.2 if we can clearly see that the mean response time of every individual process increases (or lies within the confidence interval of each other) as the server utilization is increased by increasing the arrival rate at the server.

Figure 4.2 shows a plot for the mean response time and confidence intervals for different server utilizations of the mosquitto broker.

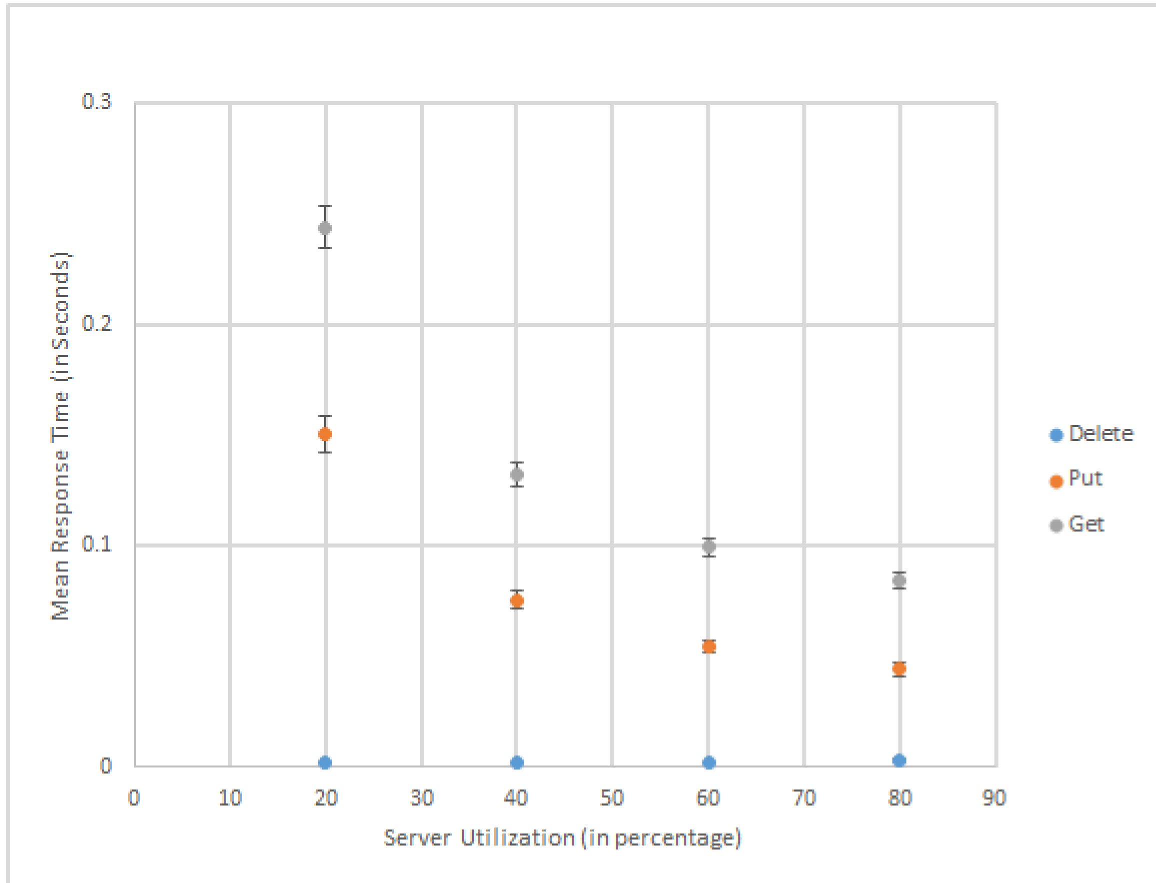

Figure 4.2 Mean Response Time vs Mosquitto broker Utilization

4.5.3 XMPP

4.5.3.1 Software Used

- **Openfire Server:** Openfire is a real time collaboration (RTC) server licensed under the Open Source Apache License. It uses the only widely adopted open protocol for instant messaging, XMPP (also called Jabber) [Xmpb].

- **Smack Client:** Smack is an Open Source XMPP (Jabber) client library for instant messaging and presence. A pure Java library, it can be embedded into an application to create anything from a full XMPP client to a simple XMPP integration, such as, sending notification messages and presence-enabling devices [Xmpb].

4.5.3.2 Implementation Notes

Inorder to implement the three types of client requests DELETE, PUT, and GET, we used the most basic feature of Presence and Instant Messaging provided by XMPP [SA04]. The client is implemented using the Smack client library to interact with openfire XMPP server. The logic to handle the smart parking clients is added to the server. The installation guide and parking implementation code with obtained results can be downloaded from github [Xmpa].

In XMPP, clients exchange XML messages called stanzas. There are three basic stanza types: message, presence, and iq (info/query). The message stanza is a root element and contains a message payload (in the child element body) as well as information about sender (from), receiver (to), message type and id. The smack client first establishes a TCP connection with the openfire server and then login for authentication. It later exchanges the presence and message stanzas inorder to send client request and get response. Below is an example for a request from a client to DELETE a specific location.

- Client sends its presence info to the server at admin

```
<presence to='admin@paridhika-satellite-c55-c' id='cURl3-68'/>
```

- The server sends a response back to the client saying that its presence is marked as available at the server.

```
<presence id="cURl3-68" type="available" from="admin@paridhika-satellite-
c55-c/7jrssqc70n" to="client@paridhika-satellite-c55-c/7jrssqc70n"/>
```

- A sensor node in the parking lot acting as a client sends a message to the server with its status information.

```
<message to='admin@paridhika-satellite-c55-c' id='cURl3-68' type='chat'>
    <body>delete:0,3</body>
</message>
```

- The Openfire processes the message body and sends back a success information message to the sensor.

```
<message to="client@paridhika-satellite-55-c/8pny1ogele" id="cURl3-68"
type="chat" from="admin@paridhika-satellite-c55-c/8pny1ogele">
    <body>deleted:0,3</body>
</message>
```

- The client after receiving a success message from the server, sends a disconnect message to remove its presence from the server.

    <presence id='cURl3-68' type='unavailable'/>

Table 4.3 XMPP Response times

20%	Service Time	Arrival Rate	Mean Response Time	99th Percentile	Standard Diviation	Confidence Interval
Delete	0.025	4	0.03354	0.08	0.02	8.92E-04
Put	0.025	2	0.03706	0.086	0.026995	0.001673
Get	0.025	2	0.0372	0.086	0.0266	0.00164964
Overall		8	0.035362	0.084	0.0238	7.42E-04
40%						
Delete	0.025	8	0.03136	0.092	0.022938	1.02E-03
Put	0.025	4	0.041806	0.108	0.03159953	0.001958559
Get	0.025	4	0.041404	0.106	0.0314	0.00194655
Overall		16	0.036548	0.098	0.02808	8.76E-04
60%						
Delete	0.025	11.9998	0.034326	0.088	0.022	9.77E-04
Put	0.025	5.9999	0.043038	0.104	0.02874	0.00178
Get	0.025	5.9999	0.043106	0.115	0.0301	0.001866
Overall		23.9996	0.038754	0.098	0.0263987	8.23E-04
80%						
Delete	0.025	15.9998	0.0328	0.093	0.02727	1.21E-03
Put	0.025	7.9999	0.042069	0.112	0.0326	0.00202
Get	0.025	7.9999	0.041518	0.109	0.030898	0.0019
Overall		31.9996	0.03735	0.102	0.02997	9.35E-04

4.5.3.3 Results

The results are summarized in Table 4.3. As mentioned above, for the parking implementation with the XMPP protocol, we used the chat message system and the three types of client requests use the same set of messages to interact with the server but with a different message body. The message body is used by the server to identify the client and send the required response back. This justifies the values of the response times obtained for all the three client request as shown in Table 4.3.

Also from the results we can see, when the arrival rate of one of the three processes (DELETE in our case) is higher as compared to others then its mean response time comes to be smaller as compared to others. But the mean response time for the GET and PUT processes are similar as they have the same arrival rates. This can be explained by the message delivery priority algorithm implemented by the XMPP server. The server gives priority to messages with the highest presence priority as "most available"(M) resource [SA04]. So it maintains a stack of users connected to the server with their presence status as "available" and sends the response back to the "most available" (ie. the last arrived client first). For this reason the DELETE messages have less lower response times as compared to the GET and PUT messages.

XMPP obviously produces higher overhead than the first two protocols by embedding messages in the XML stanza structure. However its multi threaded environment reduces the response time significantly. It opens a new TCP connection for each client. As the load increases over the server the mean response time increases for each individual DELETE, PUT and GET client request threads and hence the overall response time is also increased. From Table 4.3 we can see that the change is not very significant. As long as, we do not hit the maximum allowable number of connections for the server, the mean response time is affected minimally. As we further increase the arrival rate, the XMPP server refuses the new connections. The 99th percentiles are quite high in relation to the mean, and they suggest a long tail of the response time distribution.

Figure 4.3 shows a plot for the mean response time and confidence intervals for different server utilizations of the openfire server.

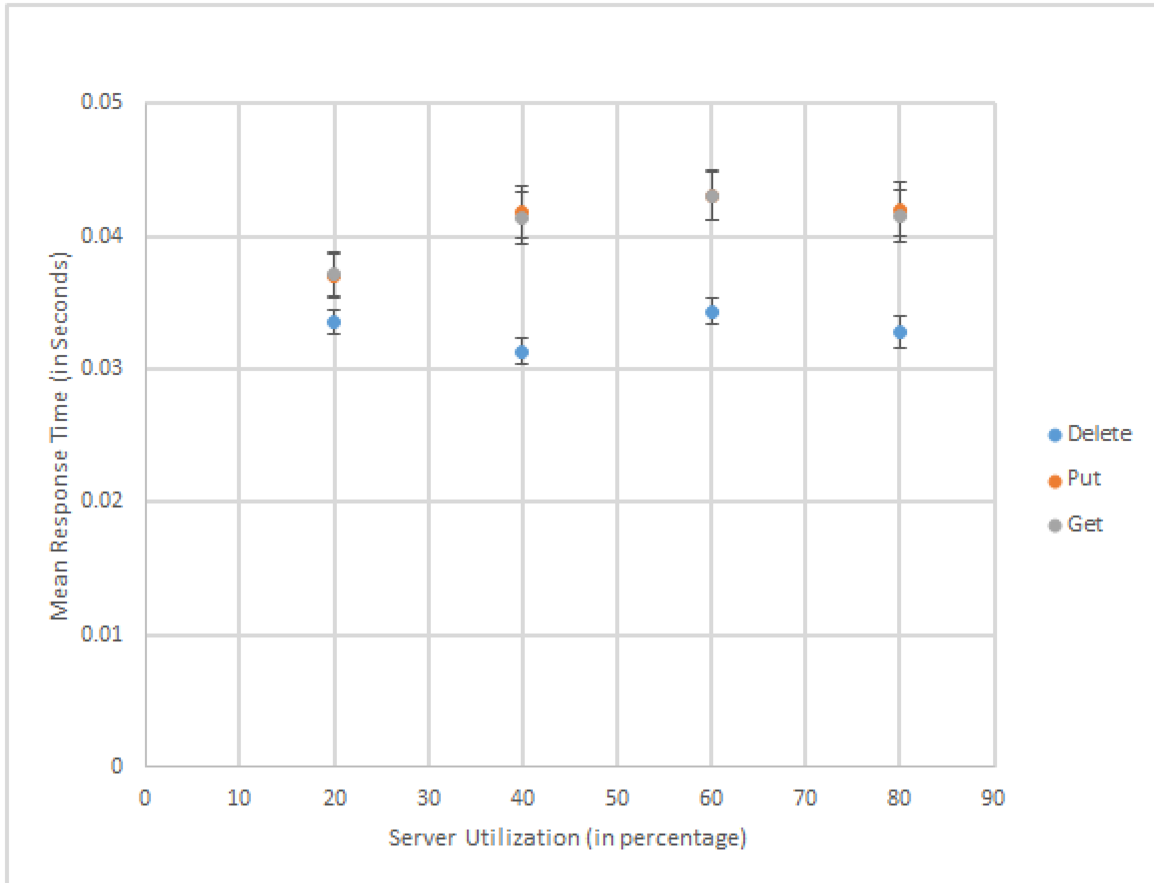

Figure 4.3 Mean Response Time vs Openfire Server Utilization

4.5.4 Websockets

4.5.4.1 Software Used

- **HiveMQ Broker [Hivc]**: HiveMQ is a simple and secure, highly scalable Enterprise MQTT Broker designed for lowest latency and highest throughput. HiveMQ is available to download for free and implements the MQTT protocol versions 3.1 and 3.1.1 with websockets support.

- **HiveMQ Plugin [Hivb]:** HiveMQ allows plugin development on top of the broker to achieve required functionality.

- **Paho python Client Library [Webb]:** The Paho Python Client provides a client class with support for both MQTT v3.1 and v3.1.1 on Python 2.7 or 3.x. Python is developed under an OSI-

approved open source license, making it freely usable and distributable, even for commercial use. It also provides some helper functions to make publishing one off messages to an MQTT server very straightforward.

4.5.4.2 Implementation Notes

The HiveMQ message broker allows clients to connect to mqtt server through websocket connection. It also provides a specifications and guide to application developers [Hiva] to build their own plugins to achieve the required behavior for their applications. We created a plugin to handle parking clients for the three request types DELETE, PUT, GET. The installation guide and parking implementation code with obtained results can be downloaded from github [Mqtc].

The Paho python client library provides API's for developers to build their own clients to connect to their hivemq server over websockets [Pah]. We created threads for running three types of clients to implement the parking model. The DELETE and PUT clients sends a publish request to the hivemq server over a websocket connection, while the GET client sends a subscribe request to the server and the server responds back with the empty location to the reservation client. All the three clients disconnects themselves from the server after receiving the required response back over the socket.

Table 4.4 MQTT over Websockets Response times

20%	Service Time	Arrival Rate	Mean Response Time	99th Percentile	Standard Diviation	Confidence Interval
Delete	0.0025	6.959998	0.0019385	0.0064299	0.0012	5.35E-05
Put	0.0025	3.479999	0.1508	0.620274	0.1304	0.008085
Get	0.05	3.479999	0.2442	0.706824	0.15262	0.0094
Overall		13.919996	0.1013	0.599645	0.14465	0.0045
40%						
Delete	0.0025	13.8998	0.0020325	0.006597	0.001384	6.14E-05
Put	0.0025	6.9399	0.0757	0.3	0.0676	0.004193
Get	0.05	6.9599	0.132296	0.37758	0.0862	0.0053
Overall		27.7996	0.0539	0.323	0.077887	0.0024253
60%						
Delete	0.0025	20.84	0.0021782	0.006458	0.0015137	6.72E-05
Put	0.0025	10.4	0.054699	0.2256	0.0476	0.00295
Get	0.05	10.44	0.0996	0.31519	0.062642	0.00386
Overall		41.68	0.04029	0.249355	0.0569	0.001773
80%						
Delete	0.0025	27.86	0.002785	0.00762	0.017464	7.85E-04
Put	0.0025	13.95	0.044395	0.180227	0.04632	0.00287
Get	0.05	13.91	0.084797	0.245836	0.060326	0.00371
Overall		55.72	0.03469	0.196246	0.053	0.00166

4.5.4.3 Results

The results are summarized in Table 4.4. To establish a WebSocket connection, the client sends a WebSocket handshake request, similar to HTTP handshake, for which the server returns a WebSocket handshake response. Once the connection is established, the client and server can send WebSocket data or text frames back and forth in full-duplex mode. The websocket protocol also has the ability to multiplex several streams simultaneously over the same connection. This explains the result in Table 4.4, ie., as we increase the arrival rate to the server the mean response time for each of the DELETE, GET and PUT request decreases and hence the overall server mean response time decreases. As the arrival rate to a websocket enabled hivemq server increases, the probability of a new client connecting before the websocket connection is stopped, increases and hence the client connects to the same bidirectional channel. As the new client communicates over the same already established socket, there is no need for header exchange between the parties, so the control data overhead is minimal which results in decreased response time with increased arrival rates.

Again, as the hivemq server works in a multi-threaded environment, it has an advantage on the response time as compared to the MQTT and CoAP servers which are based on message queues. The 99th percentiles are quite high in relation to the mean, and they suggest a long tail of the response time distribution.

Figure 4.3 shows a plot for the mean response time and confidence intervals for different server utilizations of the openfire server. Figure 4.4 shows a plot for the mean response time and confidence intervals for different server utilizations of hivemq websocket enabled mqtt server.

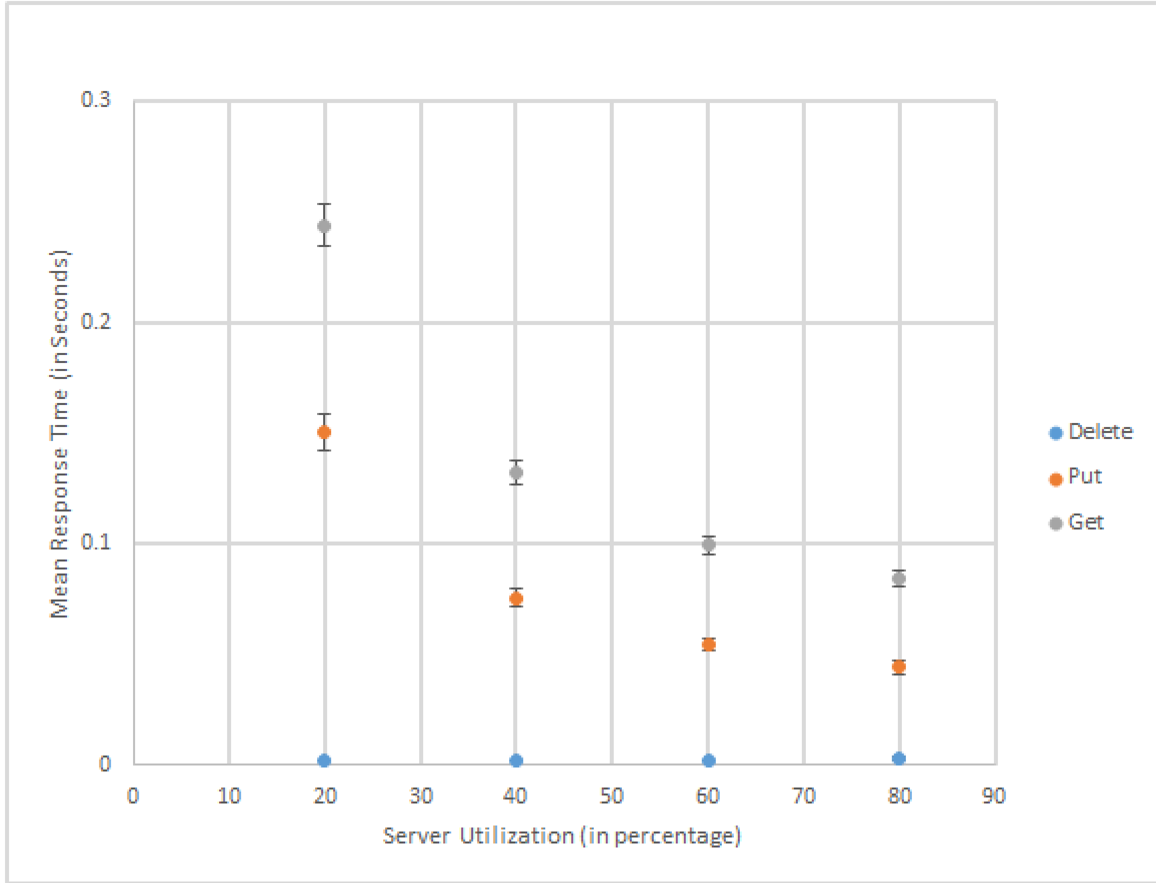

Figure 4.4 Mean Response Time vs HiveMQ Server Utilization

4.5.5 Protocol Comparison

In this section, we compare the above 4 protocols amongst each other based on the statistical results obtained previously. Figure 4.5 shows the overall mean response time comparison for the protocols for different server utilizations. It can be concluded from the results that MQTT and CoAP, since using message queues for processing client requests, results in a much higher response time as compared to the XMPP and MQTTWS.

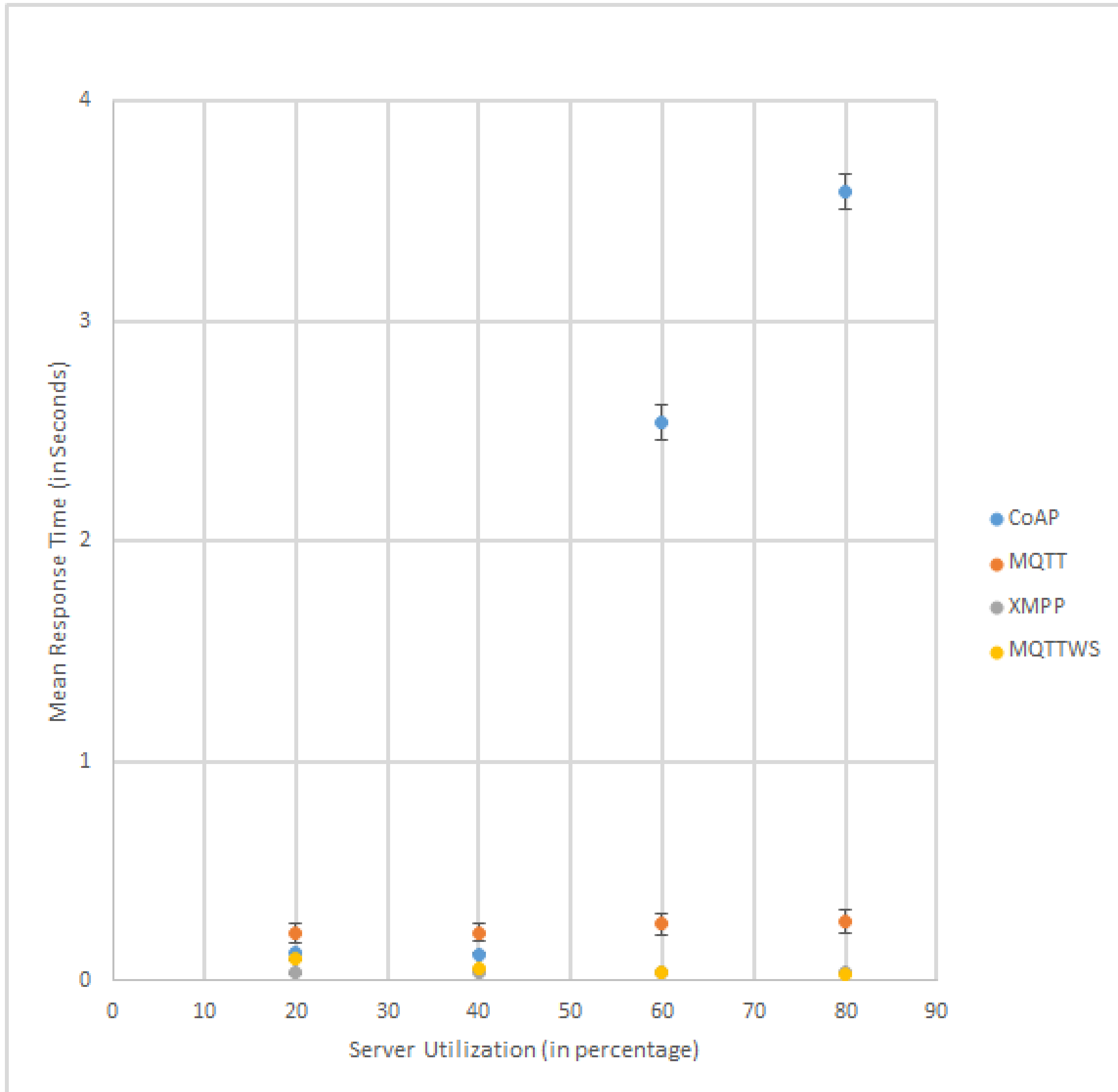

Figure 4.5 Mean Response Time for the 4 protocols vs Server Utilization

Figure 4.6 gives a closer look for the comparison obtained from Figure 4.5 by removing 2 high valued points. It can be seen that XMPP performs best in terms of mean response time at lower server utilization, while MQTTWS performs best at higher utilization. CoAP performing worst in case of higher server utilization.

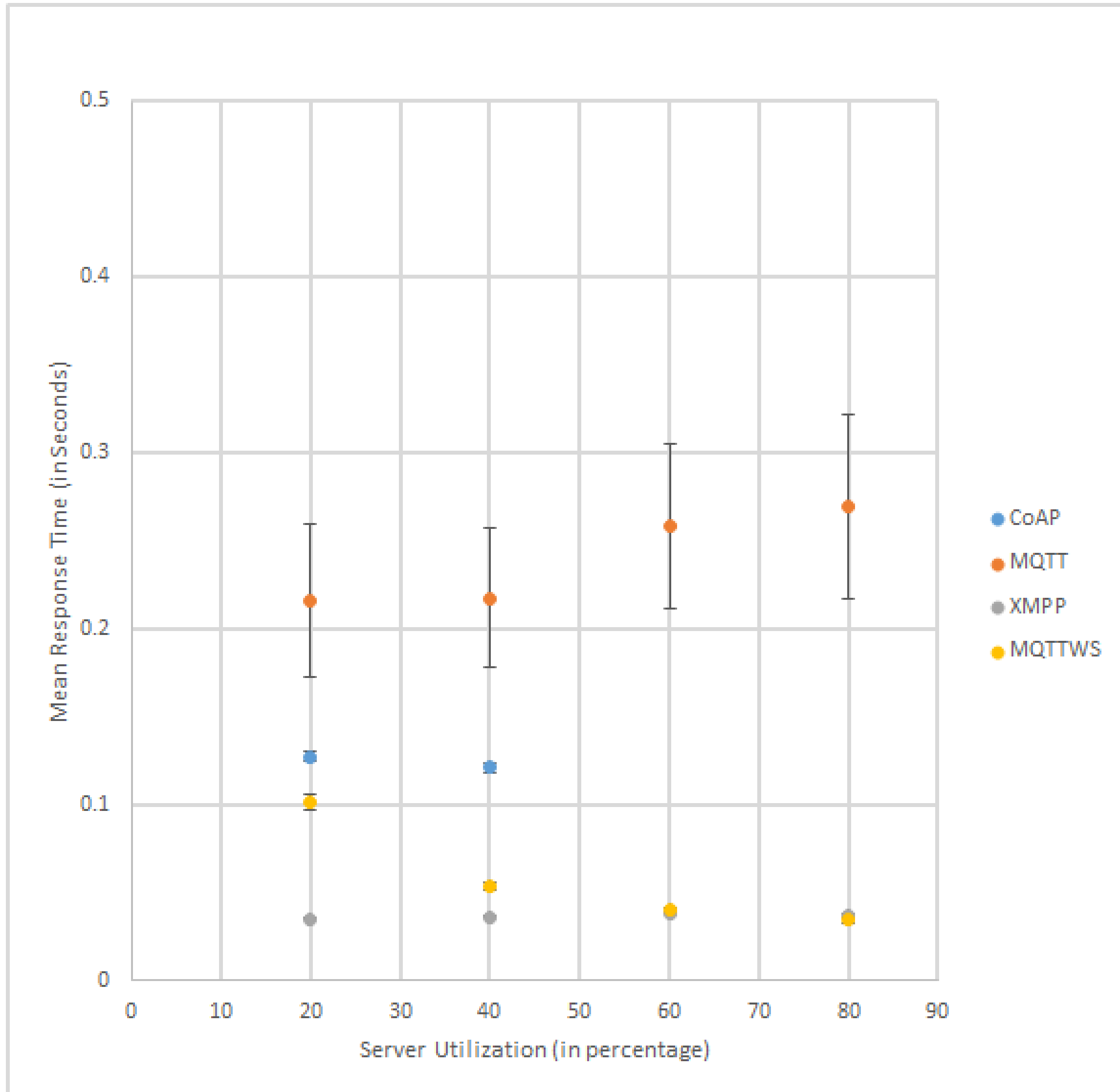

Figure 4.6 Mean Response Time vs Server Utilization a closer view

Figure 4.7 shows the comparison between two protocols, CoAP and MQTT, since they are both based on message queues. From the results it clear that CoAP performs better than MQTT with low server utilization while at higher utilization the change in mean response time is very significant as compared to MQTT. Though both CoAP and MQTT are designed for power constrained devices, MQTT still implements some advanced optimization algorithms, being a little heavier than CoAP. Also CoAP takes advantage of using UDP for communication while MQTT is based on TCP. Hence

the CoAP server performs better than MQTT at lower loads taking advantage of small header and binary format messages over UDP. MQTT performs better at higher server utilizations making use of some extra optimization features of the protocol.

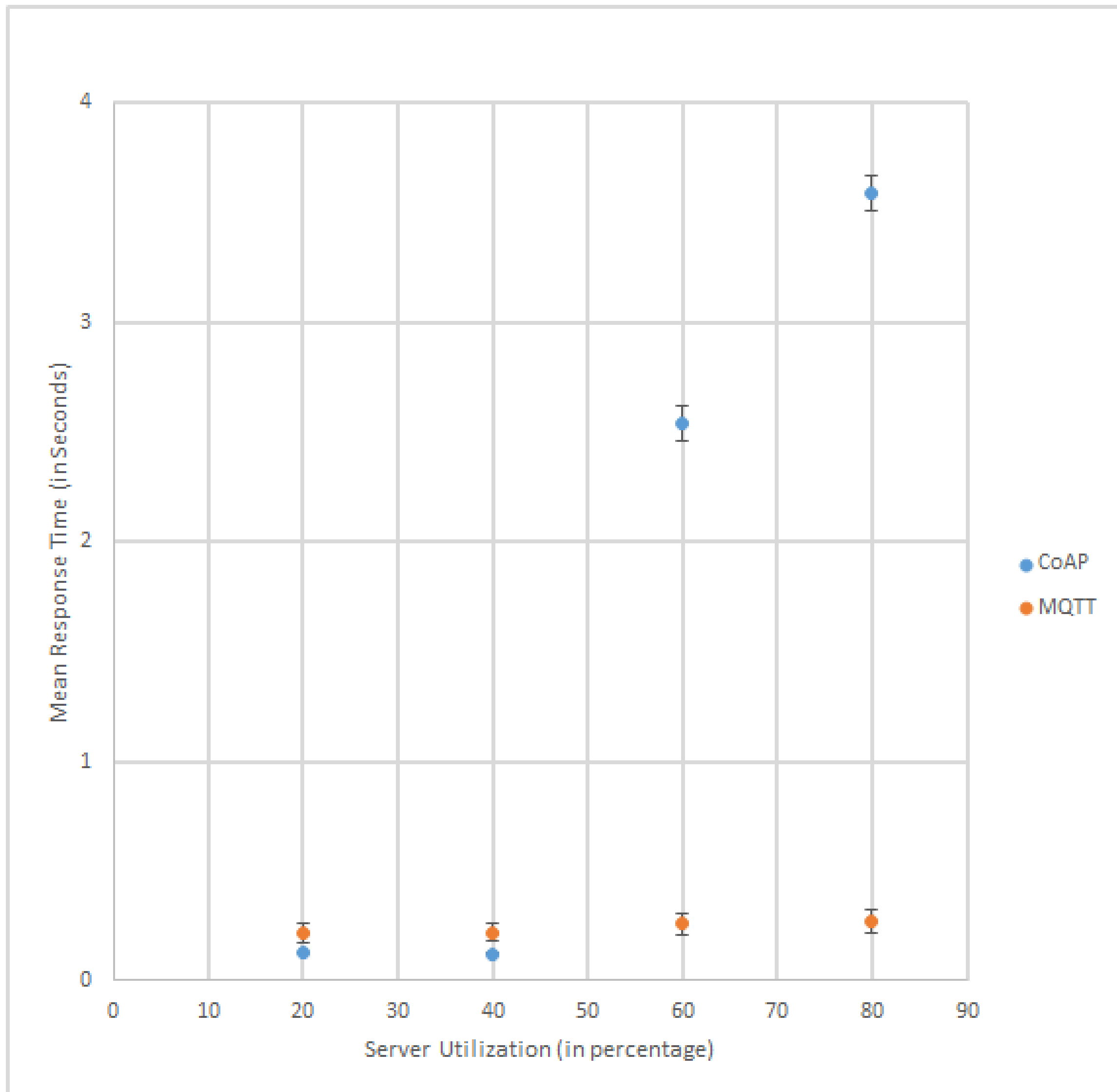

Figure 4.7 Mean Response Time for CoAP and MQTT vs Server Utilization

Figure 4.8 shows a comparison between XMPP and MQTTWS, since both are based on a multi-threaded environment. From the results we can see that XMPP performs better as compared to

websockets at lower server utilization and as load increases Websockets based hivemq server takes a lead for the mean response time. This can be explained by the fact that XMPP Smack client transfers messages directly over TCP connection while paho websocket client use websockets which require initial handshakes. As explained above, in case of websockets the number of handshakes decreases as the arrival rate increases on the server. And hence a TCP connection performs better at lower utilization, while websockets taking advantage as utilization goes higher.

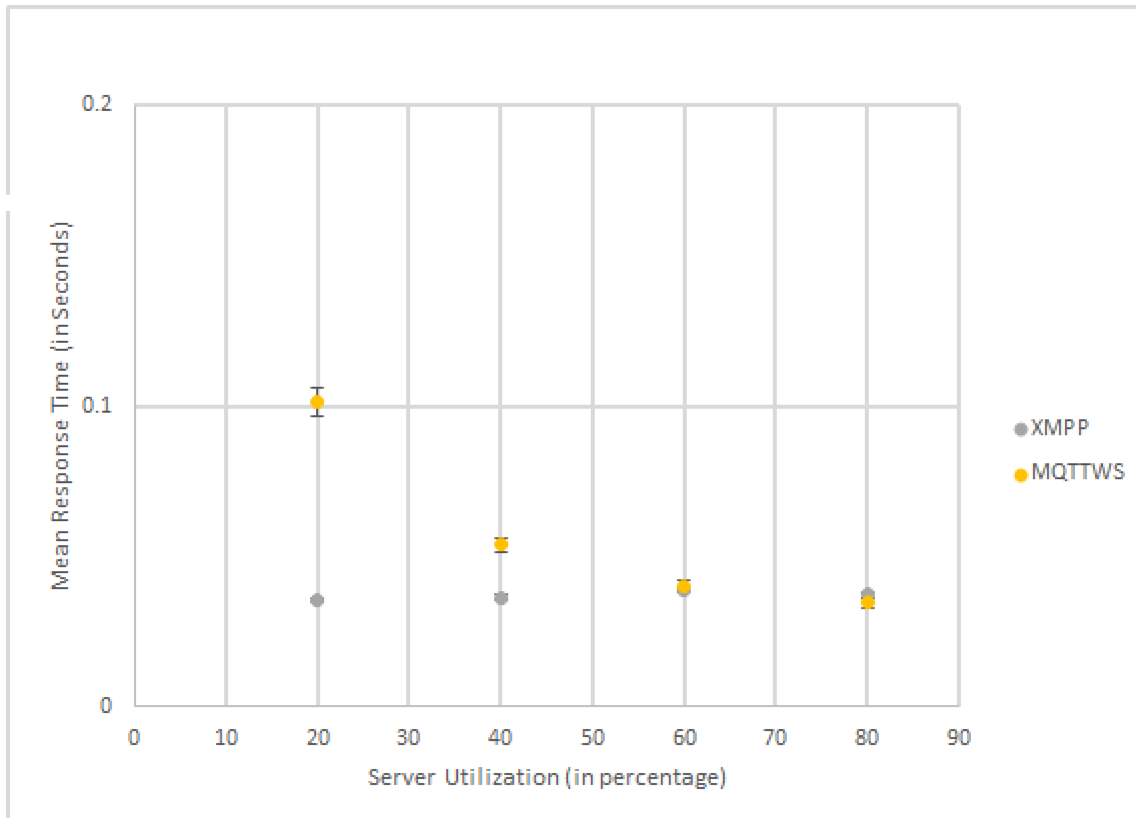

Figure 4.8 Mean Response Time for XMPP and MQTTWS vs Server Utilization

Figure 4.9, 4.10 and 4.11 shows the comparison between all 4 protocols for each of DELETE, PUT and GET messages which follows the same pattern as the overall mean response time.

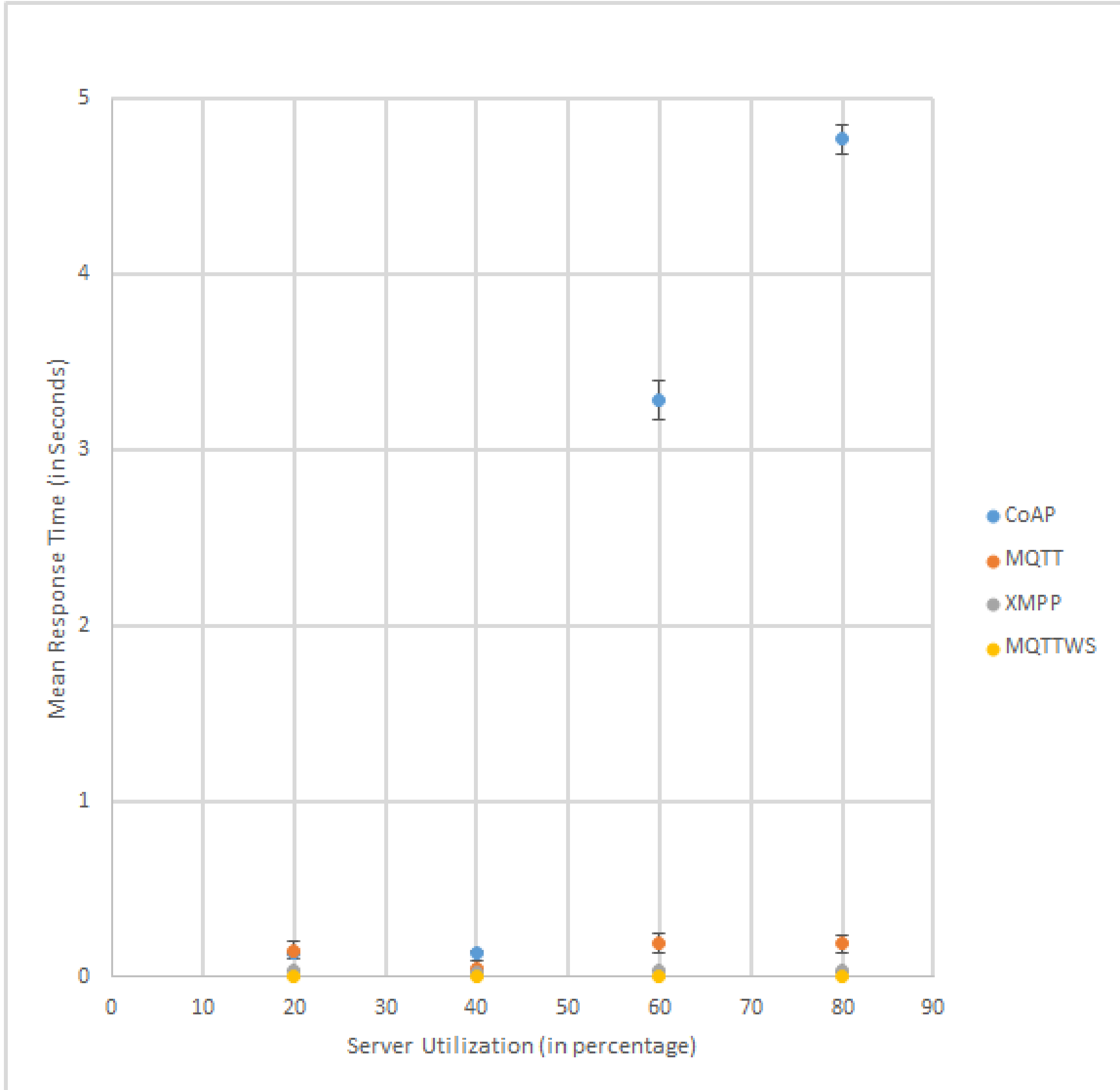

Figure 4.9 Mean Response Time vs Server Utilization for DELETE Clients

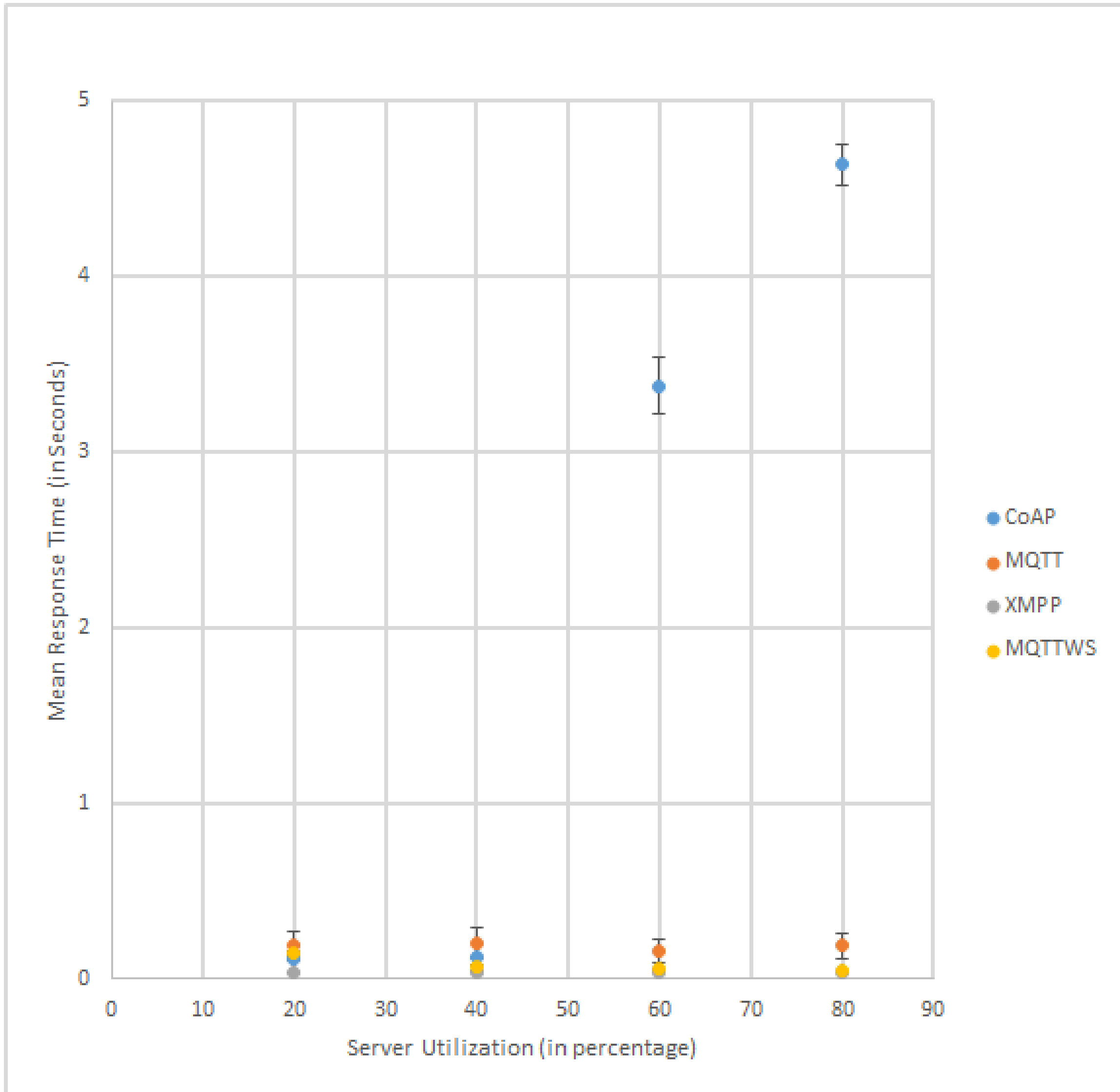

Figure 4.10 Mean Response Time vs Server Utilization for PUT Clients

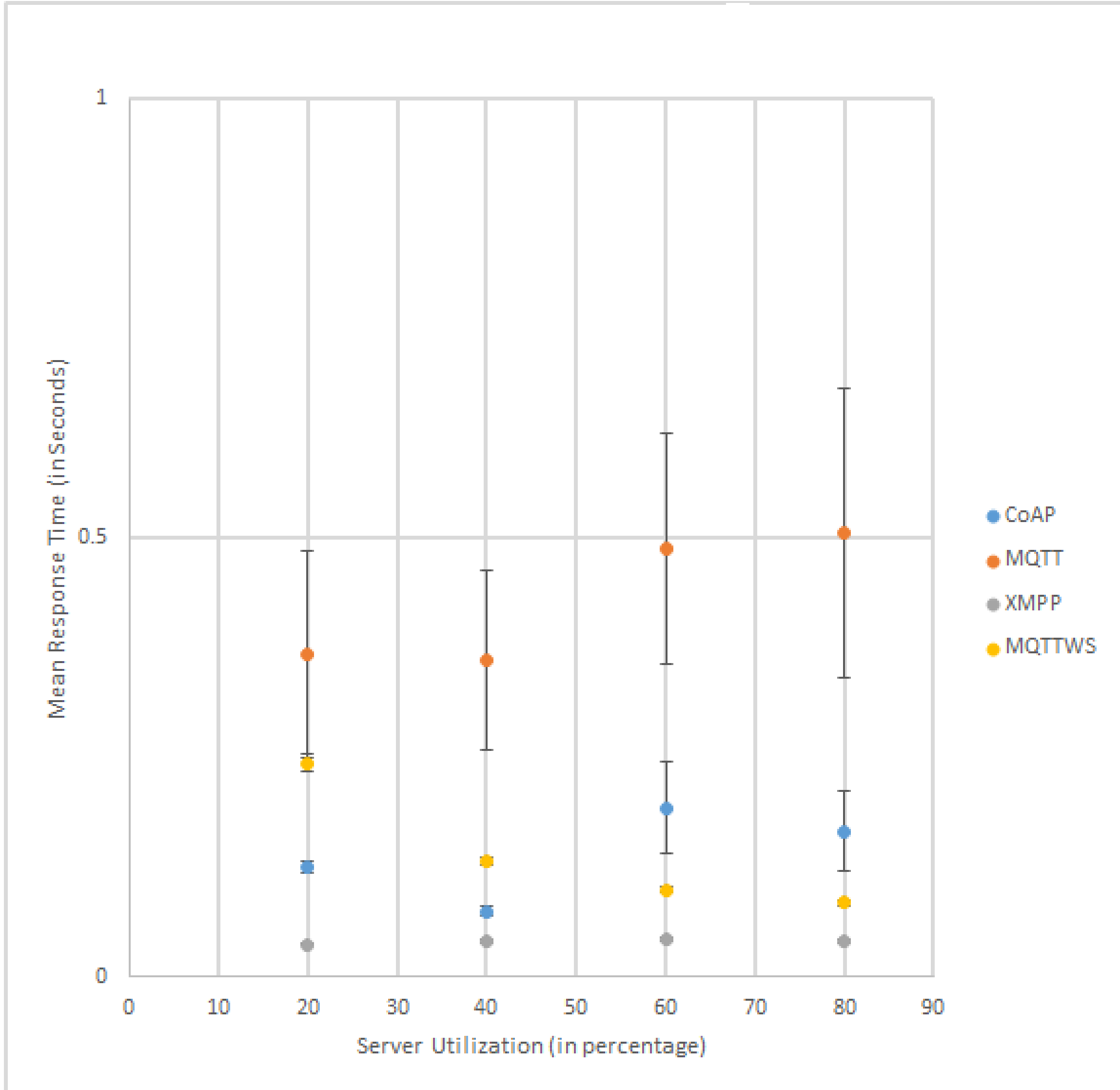

Figure 4.11 Mean Response Time vs Server Utilization for GET Clients

CHAPTER

5

CHOOSING THE APPROPRIATE PROTOCOL

Each of the protocols discussed in this thesis is widely adopted and supported in IoT. There are at least 10 implementations of each and all are used as application layer protocols. Each protocol has weaknesses: MQTT appears to be weak in security; DDS seems to be complex to scale and has version dependencies; and XMPP is considered heavy-weight. But they all have strengths too, of course: DDS has many deployments in the field to prove its relevance; XMPP supports EXI and WebSocket for efficiency and it has a proven track record; and both DDS and XMPP are extremely mature and have built-in security. Choosing the right real-time protocol depends largely on the needs of a given developer and his IoT devices. In view of this, it is important to understand the class of use that each of these important protocols addresses in order to decide which protocol suits best for a given application. In this chapter, we identify a few application requirements which are important in choosing the appropriate communication protocol. There have been many other analysis on application features and specific requirements in IoT world. We here present a more complete quantitative as well qualitative analysis on application specific requirements.

5.1 Architectural Requirements

5.1.1 Request/Response vs Publisher/Subscriber

Applications may have different communication requirements, some may require one-to-one interaction while there may be others which require a central server to maintain a connection with the devices. CoAP, XMPP and SMQ provides good solutions for applications which requires a Request/Response model. They provide an advantage for applications which are event based and need only periodic updates. MQTT, XMPP and websockets serve the publisher/subscriber architectural requirements. These protocols keep a connection to the server open for the duration of the time a user is interacting with the server. This permits constant real-time updates without the overhead of establishing a connection every time.

5.1.2 Device Solution Vs Browser Solution

IoT applications may involve both low powered devices and browser based applications at the same time. Protocols like Websockets, SMQ enable a user to interact through web browser. While protocols like CoAP, MQTT, XMPP and SMQ provides a device solution by directly opening a TCP/UDP connection to the server. CoAP, MQTT are considered lightweight and serves the purpose of power constraint devices while XMPP and SMQ comparatively heavier protocols.

5.2 Discovery and Addressing

Discovery and addressing is an important feature required specially for point-to-point communication. CoAP offers Service Discovery under which a server can be discovered by a client learning a URI that references a resource in the name space of the server. The CoAP resource directory provides a way to discover the properties of the nodes on your network. Similarly, CoSIP also allows smart objects to register the services they provide to populate a CoSIP Registrar Server, which serves as a Resource Directory. This support is missing in other IoT protocols.

5.3 Scalability

The explosion in connected devices has lead to the creation of a multitude of new and traditional applications in the Internet. The choice of protocol has to be made based on the type of traffic and number of estimated devices to be involved in communication. Each of the above described protocols support one on more of the requirements but not all.

5.3.1 Types of traffic

5.3.1.1 Long-lived vs Short-lived

Applications involving low powered devices may generate short-lived or long-lived flows of packets of different sizes. One-to-one communication protocols like CoAP and XMPP with IM and Presence, serves better for applications with short-lived traffic by avoiding the load on server. While it is worth using a long lasting MQTT or websockets connection to serve applications which sends continuous data streams, thus avoiding the overhead of connection establishment.

5.3.1.2 Small vs Large packet size

Appropriate protocol has to be chosen based on the size of data packets involved in the communication. Protocols like MQTT and CoAP sends small packets with minimal header bytes, while XMPP sends packets in XML format and hence there is no limit of the packet size.

5.3.1.3 Unidirectional vs Bidirectional

Protocol choice also depends on nature of source and direction of traffic. Some applications involve bidirectional flow of traffic between client and the server, while others involve traffic that is predominantly unidirectional.

5.3.2 Estimated number of devices

A choice of protocol has to be made keeping in mind its vertical as well as horizontal scalability and synchronization ease.

5.3.2.1 Horizontal Scalability

Some large applications involving millions of sensors typically involve multiple servers which should be able to interact and synchronize together to provide communication. XMPP provides server-to-server communication and synchronization feature which makes it horizontally scalable. MQTT over websockets creates MQTT broker cluster that is scalable in a horizontal fashion (which means you can add any number of MQTT brokers at runtime) and eliminates the single point of failure by clustering multiple broker nodes to one logical MQTT broker.

5.3.2.2 Vertical Scalability

Vertical scalability of a protocol depends on its architecture and processing model. CoAP and MQTT involves message queues for processing requests which puts a limit on the no of devices that

can be served concurrently. On the other hand XMPP and Websockets works in a multi-threaded environment by providing a separate dedicated thread to each connected device.

5.4 Latency and Throughput Requirements

IoT applications envisioned to connect small battery-cell powered devices to the Internet, are typically of low-form factor that generate/publish data in sporadic manner. However, potential large-scale deployments of such IoTs create a huge aggregated traffic to the Broker nodes that causes congestion, packet loss and delay and thereby reduced throughput (messages per second) in the network. IoT is also expected to generate large amounts of data from diverse locations that are aggregated very quickly, thereby increasing the need to have better throughput, reliability and predictability from the network.

According to our results, we found that websockets performs the best in case of high traffic on the server. Hence provides the best throughput for applications with browser support. But is case of devices which requires a TCP/UDP connection to communicate, XMPP would provide the comparable throughput. Also all the protocols mentioned above except CoAP, runs over TCP, so can be considered equally reliable.

5.5 Application level Qos

The IoT networking environment is strongly characterized by the heterogeneity of networks. Very broadly, network traffic can be categorized into two classes: throughput and delay tolerant traffic, and the bandwidth and delay sensitive (real-time) traffic, which may further be discriminated by data-related applications with different QoS requirements. For example, an environmental monitoring system, where the data traffic is more, bandwidth is the primary concern, while delay and packet loss is tolerable to some extent. Health monitoring applications may require reliable and authenticated data delivery from each node to the server. On the other hand a traffic monitoring application may be relatively stringent in terms of throughput and delay, due to the involvement of real-time data information. Applications like noise pollution monitoring or video monitoring that helps to track a person, identify suspicious activities, detect left luggage and unauthorized access, are generally delay sensitive and have strict QoS requirements. The data generation rate is independent of network congestion and degradation in bandwidth may cause serious packet drop and severe impact on performance. To ensure the QoS of such traffic, rate control and admission control is necessary to guarantee that they will receive sufficient bandwidth. A single protocol lack the ability to support all of these applications without compromising QoS for any of them. Hence a choice of protocol is to be made to handle the type of traffic generated by an application.

5.6 Identification and Authentication

Identification and authentication are two terms that described the preliminary phases of the security process in computer systems which could apply to various IoT applications. The devices must authenticate to the local gateway when sending the critical data. Then the gateway must authenticate to the cloud endpoint when forwarding this data. The applications that will analyze and render this data must also authenticate to the cloud when requesting the data.

For any kind of personally identifiable information, it is critical that the relevant users be in control of how their data is collected, shared and analyzed. A powerful mechanism to enable this sort of control is to require that the user be actively involved in the process, where the different actors above are issued the authentication tokens used for subsequent interactions. Without the user's consent, no tokens are issued and no authenticated interactions occur. Thus, no personal data can flow. One such application may be a health care system which requires an identification and authentication procedure to manage, grant access and improve medical staff morale by addressing patient safely issues. In addition, identification and authentication are essential parts to meet the requirements of security schemes and prevent thefts or losses of precious instruments and products.

5.7 Security

A great amount of data pertaining to possibly every aspect of human activity, both public and private, will be produced, transmitted, collected, stored and processed. Consequently, integrity and confidentiality of transmitted data as well as the authentication of (and trust in) the services offering that data is crucial. Hence, security is a critical functionality for the IoT. Data networks, especially wireless, are prone to a large number of attacks such as eavesdropping, spoofing, denial of service and so on. Legacy Internet systems mitigate these attacks by relying on link layer, network layer, transport layer or application layer encryption and authentication of the underlying data. Though some of these solutions are applicable to the IoT domain, the inherently limited processing and communication capabilities of IoT devices prevent the use of full-fledged security suites. Different IoT protocols provide different levels of security. Knowing what level of security is required will help you choose the best one.

End-to-end security is a mandatory security pattern for the protection of communications in the Internet. It can be ensured at different levels of TCP/IP model: application, transport or network layer. In the IoT scenario, end-to-end security is even more important, as the information being exchanged is generally sensitive and enough private. Recent research works have intensively investigated this issue, by trying to find adaptation techniques allowing the extension of existent secure protocols in the classical Internet, to WSNs while considering their severe constraints (limited energy, low memory and computational power). One of the most useful and effective approaches to

maintain data confidentiality during the data transmission process is encryption. However, some encryption algorithms may provide an easier way to attackers for tracing data and analysis of linking packets. Hence, secure communication protocols should be suitable approach to address this issue.

Table 5.1 IoT Communication Protocols Comparison

Protocol	**CoAP**	**MQTT**	**XMPP**	**Websockets**	**SMQ**	**CoSIP**
Transport	UDP	TCP	TCP	TCP	TCP	TCP/UDP
Messaging	Rqst/Response (REST)	Pub/Sub	Client/Server	Pub/Sub	Pub/Sub	Pub/Sub Rqst/Response
Discovery	Yes	No	No	No	Yes	Yes
QoS	Optional	3 levels	No	Yes	via TCP	High
Security	DTLS	TLS	TLS/SASL	WSS	HTTPS	SIPS
LowPower constraint	Excellent	Good	Fair	Fair	Fair	Fair
Message format	binary	binary	XML	XOR Encoding	binary	text
Message Header	4 bytes	2 bytes	Variable	16 bytes	3 bytes - device 1 byte - browser	4 bytes
Encryption	No	Optional	Mandatory	Optional	Optional	Optional
Device vs Browser	Device	Device	Device	Browser	Both	Both
Horizontal Scalability	No	No	Yes (server -to-server)	Yes (clustering)	Yes (multiple broker instances)	Yes (proxy -to-proxy)

CHAPTER

6

SUMMARY AND FUTURE WORK

There exists an almost bewildering choice of connectivity options for electronics engineers and application developers working on IoT related products and systems. Depending on the application, factors such as size, data requirements, authorization, security, reliability, scalability and power constraints will dictate the choice of the communication protocol. IoT application programmers are faced with the challenges of choosing an appropriate communication protocol for their resource-constrained applications. To assist the programmers in their decision process, this thesis focuses on studying some major IoT communication protocols with a view to understand their features and advantages and analyze their performance in some real world scenario.

MQTT is a great choice for multi-device networks, and CoAP is well-suited for point-to-point connections when resources are limited. Few other heavier protocols like XMPP, websockets, CoSIP provides a different level of security for applications so that they are theft safe and prevent private data from eavesdroppers. They can be used when speed and CPU usage are not critical. XMPP strengths are its addressing, security, and scalability. This makes it ideal for consumer-oriented IoT applications. Other protocols like Websockets and SMQ provides browser access to web devices. Either protocol can also be used for managing and configuring a variety of home devices. A deeper understanding of these protocols and the applications requirements is necessary to properly select which protocol is most suitable for the application at hand.

We examined the range of protocols available, analyzed specific requirements of applications that drive the features of these protocols to build a complete system. We further implemented smart

parking application using CoAP, MQTT, XMPP and Websockets as application layer protocols inorder to compare their performance on varying load. The results showed that at lower server utilization CoAP performs best among the message queue based protocols. But when the application can support multi-threading then XMPP performs best for lower server utilization. Also as we increase the server utilization the performance of all the three protocols drops but websockets follow an opposite trend because of multiplexing several streams on same connection, as higher arrival rate results in less connection terminations and thereby less handshakes again. So at higher server utilization websockets outperforms the other three protocols if application has enough CPU to allow multi-threading. Also both XMPP and websockets provide horizontal scalability while this feature is missing in CoAP and MQTT being prone to single point of failure. We also found that XMPP serves the processes in the order of most available first while others are based on FIFO queues.

In future we would want to evaluate effects of allowing clustering for XMPP and Websockets to further reduce load on single server and to support more concurrent clients. Also we would want to evaluate SMQ and CoSIP and see how their performance is affected on varying loads. In current experimental setup one of our assumptions was zero packet loss, we would like to further investigate for each protocol, the utilization at which it starts dropping packets.

BIBLIOGRAPHY

[Bab16] Babovic, Z. B. et al. “Web Performance Evaluation for Internet of Things Applications”. IEEE Access **4** (2016), pp. 6974–6992.

[BB13] Bandyopadhyay, S. & Bhattacharyya, A. “Lightweight Internet protocols for web enablement of sensors using constrained gateway devices”. Computing, Networking and Communications (ICNC), 2013 International Conference on. IEEE. 2013, pp. 334–340.

[BG14] Banks, A. & Gupta, R. “MQTT Version 3.1. 1”. OASIS standard (2014).

[Chi07] Chinrungrueng, J. et al. “Smart parking:Anapplication of optical wireless sensor network”. Applications and the InternetWorkshops, 2007. SAINTWorkshops 2007. International Symposium on. IEEE. 2007, pp. 66–66.

[Cir13] Cirani, S. et al. “CoSIP: a constrained session initiation protocol for the internet of things”.European Conference on Service-Oriented and Cloud Computing. Springer. 2013, pp. 13–24.

[Coaa] “CoAP Project”. [Online] Available: http://coap.technology.

[Coab] “CoAP Smart Parking implementation”. [Online]Available:https://github.com/paridhika/CoAP_Parking

[Cos] “CoSIP Project”. [Online] Available: http://cosip.org

[DC13] De Caro, N. et al. “Comparison of two lightweight protocols for smartphone-based sensing”. 2013 IEEE 20th Symposium on Communications and Vehicular Technology in the Benelux (SCVT). IEEE. 2013, pp. 1–6.

[FM11] Fette, I &Melnikov, A. “Rfc 6455: The websocket protocol”. IETF, December (2011).

[Jaf14] Jaffey, T. “Mqtt and coap, iot protocols”. Feb-2014.[Online]. Available: http://www. eclipse.org/community/eclipse_newsletter/2014/february/article2. php.[Accessed: 24-Mar-2016] (2014).

[Kar15] Karagiannis, V. et al. “A survey on application layer protocols for the internet of things”. Transaction on IoT and Cloud Computing **3**.1 (2015), pp. 11–17.

[LS12] Laine, M. & Säilä, K. Performance Evaluation of XMPP on theWeb. Tech. rep. Citeseer, 2012. 1995.

[Lu09] Lu, R. et al. "SPARK: a new VANET-based smart parking scheme for large parking lots".INFOCOM 2009, IEEE. IEEE. 2009, pp. 1413–1421.

[Mij16] Mijovic, S. et al. "Comparing application layer protocols for the Internet of Things via experimentation". Research and Technologies for Society and Industry Leveraging a better tomorrow (RTSI), 2016 IEEE 2nd International Forum on. IEEE. 2016, pp. 1–5.

[Weba] "MQTT overWebsocket". [Online] Available: http://www.hivemq.com/blog/mqtt-overwebsockets-with-hivemq

[Mqta] "MQTT Project". [Online] Available: http://mqtt.org/

[Mqtb] "MQTTSmart Parking implementation". [Online]Available: https://github.com/paridhika/MQTT_Parking.

[Mqtc] "MQTTWSSmart Parking implementation". [Online]Available: https://github.com/paridhika/MQTTWS_Parking

[Mun16] Mun,D.-H. et al. "An Assessment of Internet of Things Protocols forResource-Constrained Applications". Computer Software and Applications Conference (COMPSAC), 2016 IEEE 40th Annual. Vol. 1. IEEE. 2016, pp. 555–560.

[Webb] "Paho Python Client". [Online] Available: https://eclipse.org/paho/clients/python

[Pah] "Paho Python ClientUserGuide". [Online] Available: https://pypi.python.org/pypi/pahomqtt/1.1.

[SA04] Saint-Andre, P. Extensible messaging and presence protocol (xmpp): Instant messaging and presence. Tech. rep. 2004.

[SA11] Saint-Andre, P. "Extensible messaging and presence protocol (XMPP): Core" (2011).

[She14] Shelby, Z. et al. The constrained application protocol (CoAP). Tech. rep. 2014.

[Smqa] "SMQProtocol Specification". [Online]Available: https://realtimelogic.com/downloads/docs/SMQspecification.pdf.

[Sri09] Srikanth, S. et al. "Design and implementation of a prototype smart parking (spark) system using wireless sensor networks". Advanced Information Networking and ApplicationsWorkshops, 2009.WAINA'09. International Conference on. IEEE. 2009, pp. 401–406.

[SCT08] Stanford-Clark, A. & Truong, H. L. "MQTT for sensor networks (MQTT-S) protocol specification". International Business Machines Corporation version **1** (2008).

[Tha14] Thangavel, D. et al. "Performance evaluation of MQTT and CoAP via a common middleware". Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), 2014 IEEE Ninth International Conference on. IEEE. 2014, pp. 1–6.

[Hiva] "HiveMQ Plugin Developer Guide". [Online]Available: http://www.hivemq.com/docs/plugins/latest.

[Hivb] "HiveMQ Plugin Support". [Online] Available: https://github.com/hivemq/hivemq-helloworld-plugin. 55

[Hivc] "HiveMQ User Guide". [Online] Available: http://www.hivemq.com/docs/hivemq/latest.

[Smqb] "SMQ: IoT Messaging". [Online]Available: https://realtimelogic.com/products/simplemq.

[WH11] Wang, H. & He,W. "A reservation-based smart parking system". Computer Communications Workshops (INFOCOM WKSHPS), 2011 IEEE Conference on. IEEE. 2011, pp. 690–695.

[Xmpa] "XMPP Smart Parking implementation". [Online]Available: https://github.com/paridhika/XMPP_Parking

[Xmpb] "XMPP Softwares". [Online] Available: http://igniterealtime.org/projects/index.jsp.

[YS16] Yassein, M. B., Shatnawi, M. Q., et al. "Application layer protocols for the Internet of Things: A survey". Engineering & MIS (ICEMIS), International Conference on. IEEE. 2016, pp. 1–4.

[Zha12] Zhao, H. et al. "IPARK: Location-aware-based intelligent parking guidance over infrastructureless VANETs". International Journal of Distributed Sensor Networks **2012** (2012).