A Comparison of IoT application layer protocols through a smart parking implementation

Paridhika Kayal and Harry Perros
{pkayal,hp}@ncsu.edu
Computer Science Department
North Carolina State University

***Abstract*—Several IoT protocols have been introduced in order to provide an efficient communication for resource-constrained applications. However, their performance is not as yet well understood. To address this issue, we evaluated and compared four communication protocols, namely, CoAP, MQTT, XMPP, and WebSocket. For this, we implemented a smart parking application using open source software for these protocols and measured their response time by varying the traffic load.**

***Keywords*—CoAP, MQTT, XMPP, WebSocket, smart parking, response time.**

I. INTRODUCTION

An IoT application typically involves a large number of deployed and interconnected sensors and gateways. The sensors measure the physical environment and send the data to a gateway. The gateway aggregates the data from various sensors 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 an IP-compatible protocol stack 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.

IoT 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. 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, QoS, interoperability, fault tolerance and security are taken into account.

In this paper, we report on the response time of the following communication protocols: CoAP, MQTT, XMPP, and WebSocket, measured for different traffic loads.

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. The WebSocket (WS) standard provides bi-directional Web communication and connection management. WebSocket is a good IoT solution if the devices can afford the WebSocket payload. Other protocols, such as, SMQ and CoSIP are also gaining traction. 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.

II. RELATED WORK

Several surveys have provided description and comparative analysis of IoT application layer protocols without providing real data measurements, see [1], [2], [3], [4]. All these surveys identified CoAP, MQTT, XMPP, AMQP and REST services as the most representative protocols for internet of things. The article in [2] provides a description of these key protocols their architectures and strengths and weaknesses. 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.

Several works experimentally tested the most popular IoT application layer protocols, typically comparing two selected protocols. In [5] the authors compared MQTT and CoAP by creating a middleware component in order to perform testing. They found that MQTT has a lower latency for smaller packet loss than CoAP, and in contrast, higher latency than CoAP for higher packet loss. Two IoT protocols CoAP and MQTT have been assessed in terms of energy consumption, bandwidth utilization and reliability [6]. According to the result, CoAP is the most efficient in terms of energy consumption and bandwidth usage while MQTT provides high reliability. [7] provide a qualitative and quantitative comparison between MQTT and CoAP when used as smartphone application protocols. In [8] performance tests are conducted by implementing the experimental design for three different techniques in a LAN for supporting real-time communication with XMPP on the Web.

In paper [9] CoAP, WebSocket and MQTT are 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. Authors in [10] evaluated the performance, energy consumption, and resource usage characteristics of IoT protocols including CoAP, MQTT, MQTT-SN, WebSocket, and TCP for varying packet size. [11] compared web performance of MQTT, AMQP, XMPP, and DDS by measuring the latency of sensor data message delivery and the message throughput rate for different message size and format.

There are only a few studies available on protocol efficiency, strictly related to overhead, RTT, packet fragmentation, QoS and retransmission. To the best of our knowledge there is no performance study of the response time with varying traffic loads. In this work we take an experimental approach by considering a real IoT scenario. We implemented a smart parking application using the following application layer protocols: CoAP, MQTT, XMPP and WebSocket. Being implemented on the same platform, the comparison between theses protocols is fair and realistic. The measured performance metrics is the average response time and its confidence interval and the 99th percentile for varying server utilizations.

III. PROTOCOLS OVERVIEW

In this section, we review briefly the four protocols under study.

A. CoAP (Constrained Application Protocol)

CoAP [12] is a one-to-one protocol for transferring state information between client and server over the Internet using UDP, and it is primarily designed for constrained devices. Clients may send GET, PUT, POST and DELETE resource requests to the server. CoAP messages are encoded in a simple binary format. Packets are simple to generate and can be parsed in place without consuming extra RAM in constrained devices.

We used the CoAP libcoap library that provides a coap-client which is a wget-like tool to generate the resource requests. Below is an example of the commands used to send a coap-uri request for a DELETE message:

./coap-client -m delete coap://[::1]/location -e "Message to be sent"

coap-URI = "coap:" "//" host [":" port] path-abempty ["?" query]

B. MQTT (Message Queuing Telemetry Transport)

MQTT [16] is a client/server publish/subscribe messaging protocol designed for lightweight M2M communications. The protocol runs over TCP/IP to provide ordered, lossless, bidirectional connections. MQTT supports three QoS levels and messages can be encrypted with SSL/TLS. MQTT brokers may require username and password authentication from clients to connect.

We used the Eclipse Mosquitto (EPL/EDL licensed) message broker that implements the MQTT protocol versions 3.1 and 3.1.1. It provides an MQTT server which can handle publish and subscribe messages sent from the clients. Below is an example.

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

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

C. XMPP (eXtensible Messaging and Presence Protocol)

XMPP [15] is a distributed client/server architecture and TCP communications protocol based on XML that enables near-real-time exchange of structured data between two or more connected entities. XMPP provides a wide range of applications, such as, instant messaging, multi-party chat, voice and video calls, collaboration, and lightweight middleware. It provides SASL authentication and has built-in TLS encryption.

We used the Openfire server licensed under the Open Source Apache License. It uses the only widely adopted open protocol for instant messaging, XMPP (also called Jabber).

D. MQTT over WebSocket

WebSocket [18] operate over TCP as an upgrade to a standard HTTP connection allowing for full-duplex, low-latency communication between a server and a client. In MQTT over WebSocket, an MQTT message is encapsulated within a WebSocket packet. Messages over WebSocket are sent in frames, which have only 2 bytes overhead. WebSocket are suitable as transport for MQTT because the communication is bidirectional, ordered and lossless.

We used the following software: a) HiveMQ Broker, a simple and secure, highly scalable enterprise MQTT broker designed for lowest latency and high throughput; b) the HiveMQ Plugin, which allows plugin development on top of the broker to achieve required functionality, and c) the Paho python client library**,** which provides a client class with support for both MQTT v3.1 and v3.1.1 on Python 2.7 or 3.x.

IV. SMART PARKING SYSTEM

In this paper, we use a smart parking testbed to evaluate and compare the above four communication protocols. Smart parking makes it easier for drivers to find a spot in a car parking to park their car and avoids the overhead of moving around in the parking looking for an empty slot.

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) as a PUT or a DELETE message to update their status at the server.

Also, there is a sensor attached to every car and as a car arrives at the parking lot, the sensor sends a GET request to reserve an empty space in the parking. If there is an available spot, the server reserves it and sends the exact location of the reserved spot back to the car sensor. Cars can also come in and park at an empty position without requesting the server to reserve a spot. In this case, the sensor at the parking spot sends a PUT request to the server to indicate that it has become occupied. The server stores the information of all the occupied and empty spots in the parking, so that it can serve requests from GET clients.

The following assumptions were made:

- The car 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 the communication protocol server queue is assumed to be large enough to accommodate all messages pending execution.
- The communication between clients and server is assumed to be fully reliable with zero packet loss.

In order to implement the testbed, we considered a parking space of 100 cars depicted as a 10x10 array with a sensor attached at every location. The server stores the status of every spot 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 requests for reservation according to a Poisson distribution with rate λ_1. The PUT thread generates messages due to cars occupying a parking spot 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 mean $1/\xi$. The server synchronizes the parking map according to the requests it receives.

The three client threads and the server 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. A programmable propagation delay can be easily introduced, but we were only interested in the response time of the protocols without including two-way propagation delays.

We also note that the exponential assumption could be easily replaced by a non-bursty general distribution. The service time for the GET, PUT and DELETE message was first calculated by issuing a single request to the server which was idle. Let $1/\mu_1$, $1/\mu_2$, and $1/\mu_3$ be the time required by the idle server to process a single GET, PUT and DELETE request respectively.

The testbed can be seen as consisting of 2 queueing systems. The first one is an M/M/s/K queue, where s=K=100 and it depicts the parking lot consisting of the 100 parking spots with an arrival rate of $\lambda_1 + \lambda_2$ and a mean service time of each server equal to the mean waiting time of a car in parking lot, i.e., $1/\xi$. 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. p_b can be obtained using the Erlang-B formula.

The second queueing model is an M/M/1 queue and it depicts the server as implemented with the given protocol where GET, PUT and DELETE messages queue up to get processed. In order for this queue to be stable, $U< 1$, where $U= \lambda_1/\mu_1 + \lambda_2/\mu_2 + \eta/\mu_3$. U can be seen as the workload offered to the server. CoAP and MQTT maintain a single server, and therefore U is the server utilization. However, this is not the case in XMPP and WebSocket because they spawn multiple threads depending on the arrival rate. For presentation purposes we will refer to U as the server utilization.

We picked a value for $1/\xi$ for each protocol, and then used the above M/M/100/100 queue to calculate all possible combinations of λ_1 and λ_2 so that $p_b = 0$. In this case $\eta = \lambda_1 + \lambda_2$. Subsequently, we chose a pair of λ_1 and λ_2 that corresponds to 20%, 40%, 60% and 80% utilization of the server. We could have used any combination of λ_1 and λ_2, since we can calculate η, and thus the total arrival rate to the server. Then, vary λ_1 and λ_2 so that to obtain a given server utilization. However, this does not make any difference since we run the experiments 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. Once a car occupies a position, its parking time is generated from an exponential distribution with a mean $1/\xi$. The departure time 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 and λ_2 were such that they were as close as possible, i.e., $\lambda_1/\lambda_2 \sim 1$. This can be varied, but we do not expect any difference in the 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, i.e., the time when a process thread 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 sets of samples of response times, 1000 GET samples, 1000 PUT samples and 2000 DELETE samples. For each sample we calculated the mean, standard deviation and the 99th of the response time. The three samples were also combined to calculate similar overall statistics. The order in which they were combined is immaterial as far as these statistics are concerned. The results of the interval estimation of the mean response time are presented in the next section. The unit of time is seconds.

V. PERFORMANCE EVALUATION

A. CoAP:- libcoap library

We implemented the three types of messages using the GET, PUT and DELETE commands provided by CoAP. The service time for a PUT and a DELETE request is found to be the same, but the service time of a GET request is significantly less. 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.

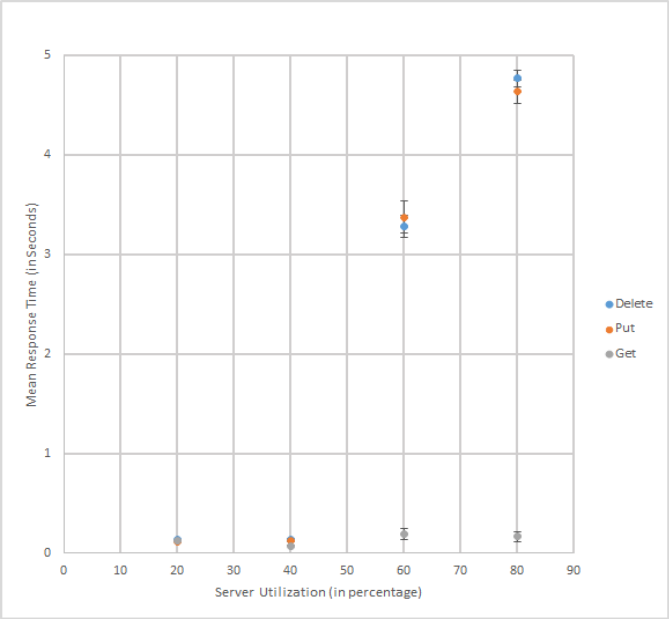

Fig. 1 Mean Response Time vs libcoap Server Utilization

As can be seen in Fig. 1, as the server utilization is increased, the mean response time of DELETE and PUT also goes up as expected. In the case of 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. A validation and freshness mechanism is used for this purpose [12]. The PUT and DELETE commands of CoAP have non-cacheable response codes while the GET command's response codes are cached. 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. In view of this, 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 caching.

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. This requires buffer reallocation, which is 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.

B. MQTT:- Mosquitto Broker

With MQTT's publisher/subscriber model, we used the publish message to implement PUT and DELETE requests, and the subscribe message followed by a publish message to implement a GET request. Hence, we found that the service time of the PUT and DELETE requests were equal but that of the GET request was higher.

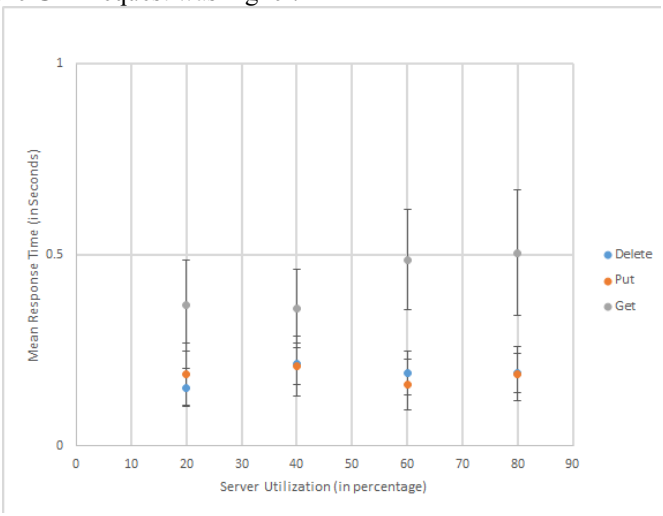

Fig. 2 Mean Response Time vs Mosquitto Broker Utilization

From Fig. 2, we can 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.

C. XMPP:- Openfire Server & Smack Client

We used XMPP's Instant Messaging feature to implement the three different requests and hence the service time for all the three messages was found to be equal. From Fig. 3, we see that as the load increases the mean response time increases for each individual DELETE, PUT and GET client requests but the change is not significant. This is because it opens a new TCP connection for each client.

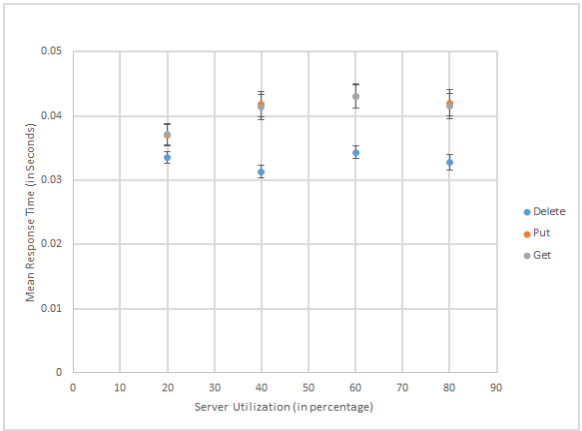

Fig. 3 Mean Response Time vs Openfire Server Utilization

From Fig. 3, we can see that the mean response time for the GET and PUT processes are similar as they have the same arrival rate while that of the DELETE is lower. This is because the server gives priority to messages with the highest presence priority as "most available"(M) resource [13]. 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" (i.e., the last arrived client first). This results in a lower mean response time for DELETE messages because of their higher arrival rate.

D. WebSocket:- HiveMQ Broker and Paho Python Client

We used the publish request to implement PUT and DELETE messages while a subscribe request to implement the GET message. The service time of the PUT and DELETE messages was found to be equal while that of the GET message was greater.

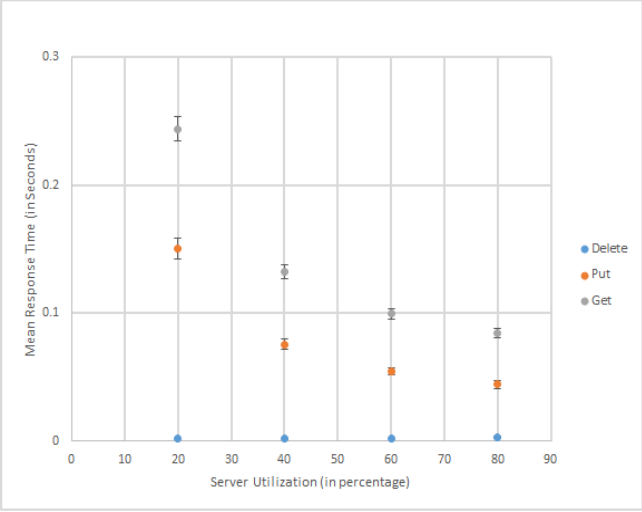

Fig. 4 Mean Response Time vs HiveMQ Server Utilization

From Fig. 4, we see that the mean response times of each individual process goes down on increasing server utilization. This is because the WebSocket protocol also has the ability to multiplex several streams simultaneously over the same connection. Hence, as the arrival rate increases, it skips few handshakes and the new client connects over the same websocket, resulting in a decrease in the mean response time.

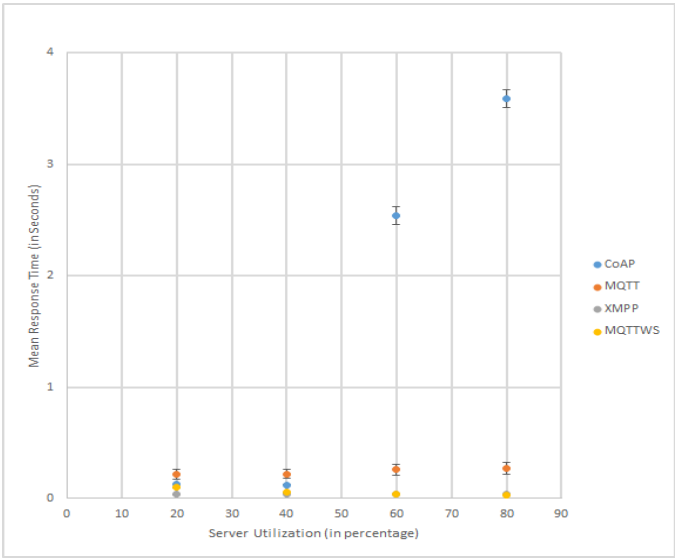

Fig. 5 Mean Response Time of the Four Protocols vs Server Utilization

E. Comparisons of the Protocols

Fig. 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 have a much higher response time, since they use a message queue to process client requests, as compared to XMPP and MQTTWS which are multi-threaded.

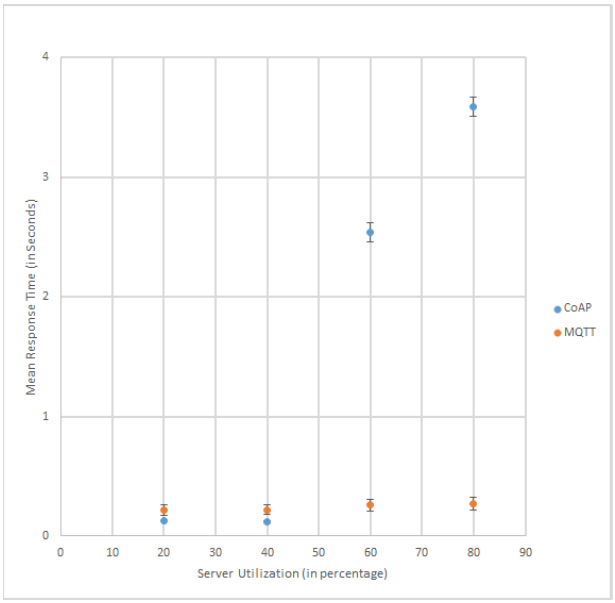

Fig. 6 Mean Response Time for CoAP & MQTT vs Server Utilization

Fig. 6 shows the comparison of the two message queue based protocols. We see that CoAP takes advantage of using UDP for communication and hence performs better at lower utilization. MQTT performs better at higher server utilizations making use of some extra optimization features of the protocol.

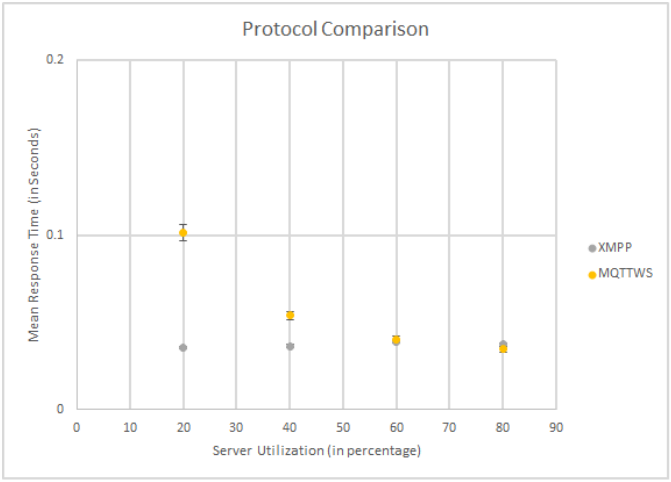

Fig. 7 Mean Response Time for XMPP & WebSocket vs Server Utilization

Fig. 7 shows the comparison of the two multi-threaded protocols XMPP and WebSocket. We can see that XMPP performs better at lower server utilization as it transfers

messages directly over a TCP connection while WebSocket ues an extra WebSocket handshake. As load increases WebSocket takes advantage of multiplexing and hence the mean response time goes down.

VI. CONCLUSIONS AND FUTURE WORK

We implemented a smart parking application using CoAP, MQTT, XMPP and WebSocket as application layer protocols in order to measure and compare their response time for different loads. The results showed that at lower server utilization, CoAP performs best between the two message queue based protocols. When the application can support multi-threading, then XMPP performs the best for lower server utilization. As we increase the server utilization, the mean response time of the protocols increases except that of WebSocket which follows an opposite trend. This is because it multiplexes several streams onto the same connection due to higher arrival rates, resulting in less connection terminations and consequently fewer handshakes. So, at higher server utilization WebSocket outperforms the other three protocols given that the application has enough CPU to allow multi-threading. Also, both XMPP and WebSocket 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 scheduling.

The 99th percentiles are not reported due to lack of space, but they follow the same pattern as the mean response times. They can be found along with the interval estimates of the mean response times and the input values to the experiments in [19].

REFERENCES

[1] T. Jaffey. MQTT and CoAP, IoT protocols. Feb-2014.[Online]. Available:http://www.eclipse.org/community/eclipse_newsletter/2014/february/article2.php, 2014.

[2] V. Karagiannis, P. Chatzimisios, F. Vazquez-Gallego, and J. Alonso-Zarate. A survey on application layer protocols for the internet of things. Transaction on IoT and Cloud Computing, vol 3, pages 11–17, 2015.

[3] M. B. Yassein, M. Shatnawi, et al. Application layer protocols for the Internet of things: A survey. In Engineering & MIS (ICEMIS), pages 1–4, 2016.

[4] Z. B. Babovic, J. Protic, and V. Milutinovic. Web performance evaluation for Internet of things applications. IEEE Access, vol 4, pages 6974–6992, 2016.

[5] D. Thangavel, X. Ma, A. Valera, H.-X. Tan, and C. K.-Y. Tan. Performance evaluation of MQTT and CoAP via a common middleware. In ISSNIP, 2014, pages 1–6, 2014.

[6] S. Bandyopadhyay and A. Bhattacharyya. Lightweight Internet protocols for web enablement of sensors using constrained gateway devices. In ICNC, pages 334–340, 2013.

[7] N. De Caro, W. Colitti, K. Steenhaut, G. Mangino, and G. Reali. Comparison of two lightweight protocols for smartphone-based sensing. In 2013 IEEE 20th Symp. on Comm. and Vehicular Technology in the Benelux (SCVT), pages 1–6, 2013.

[8] M. Laine and K. Säilä. Performance evaluation of XMPP on the web. Technical report, Citeseer, 2012.

[9] S. Mijovic, E. Shehu, and C. Buratti. Comparing application layer protocols for the Internet of things via experimentation. In Research and Technologies for Society and Industry Leveraging a Better Tomorrow (RTSI), pages 1–5, 2016.

[10] D.-H. Mun, M. Le Dinh, and Y.-W. Kwon. An assessment of internet of things protocols for resource-constrained applications. In COMPSAC, vol. 1, pages 555–560, 2016.

[11] Z. B. Babovic, J. Protic, and V. Milutinovic. Web performance evaluation for Internet of things applications. IEEE Access, vol 4, pages 6974–6992, 2016.

[12] Z. Shelby, K. Hartke, and C. Bormann. The constrained application protocol (CoAP). Technical report, 2014.

[13] P. Saint-Andre. Extensible messaging and presence protocol (XMPP): Instant messaging and presence. Tech. report, 2004.

[14] S. Cirani, M. Picone, and L. Veltri. CoSIP a constrained session initiation protocol for the Internet of things. In European Conference on Service-Oriented and Cloud Computing, pages 13–24. Springer, 2013.

[15] P. Saint-Andre. Extensible messaging and presence protocol (XMPP): Core. [Online] Available: xmpp.org/rfcs/rfc3920.html.

[16] A. Banks and R. Gupta. MQTT 3.1. 1. OASIS standard, 2014.

[17] SMQ protocol specification. [Online] Available: https://realtime.com/downloads/docs/SMQ-specification.pdf.

[18] I Fette and A Melnikov. RFC 6455: The WebSocket protocol. IETF, Dec, 2011.

[19] P. Kayal, A Comparison of IoT application layer protocols through a smart parking implementation. MS thesis, North Carolina State University, Dec. 2016.