CSC 591/791, ECE 592/792
Spring 2022
Homework Assignment # 2

Group # 12

Team Members and Task Contributions

Task	Timothy Boushell	Alex Carruth	Andrew Sonnier	Faisal Almutairi	Arnab Datta
MQTT	0%	35%	15%	0%	50%
COAP	25%	0%	75%	0%	0%
HTTP	0%	0%	100%	0%	0%
Report	25%	25%	25%	0%	25%

Observation and Discussion

	Throughput (in kilo bits per second)								**Total application layer data transferred from sender to receiver (including header content) per file divided by the file size (in kilo bits)**			
	100B file		**10kB file**		**1MB file**		**10MB file**		**100B file**	**10kB file**	**1MB file**	**10MB file**
	Average	*Std. Dev.*	*Average*	*Std. Dev.*	*Average*	*Std. Dev.*	*Average*	*Std. Dev.*	*Average*	*Average*	*Average*	*Average*
MQTT QoS1	83539.4	1788 5.09	828635.173	141 518.142	84499 0176.227	1700 06467.085	7041 13830 1.81	3214 33080 1.88	2.35	1.0376	1.058	1.023
MQTT QoS2	1668.6 681	682. 29	17726.01	793 8.2	46784 083.53	1684 3543. 767	5114 00284 .295	2223 89616 .449	3.55	1.0496	1.097	1.044
CoAP	597.81	2029 .42	6318.01	260 75.6 4	5772. 02	1581 78.70	3513. 39	1050 9.47	1.48	1.00	1.00	1.00
HTTP	328.84	743. 64	35722.2 5	103 8535 .99	18766 59.72	4556 2676. 80	3489 190.8 6	21641 500.8 0	1.48	1.03	1.05	1.03

1. **Observations from the experiments and from the completed table in "Results File.xlsx".**

Our team used docker for all of our experiments in order to standardize the environment as well as for ease of repeatability. While docker does create a performance hit, especially when dealing with IO operations, this was a worthy trade off in order to make the experiments more repeatable and automatable.

HTTP

HTTP was the easiest of the four protocols to complete as the support online is the best. This was accomplished by using the flask library as the server and the library requests for the client. Application layer size was measured by adding the content length to the size of the headers. This will be slightly larger than the actual size due to measuring the size of python objects but the results are consistent among themselves. The results in the table show that as the file size increases, the measured throughput increases as well. This is due to, at lower file sizes, a larger percentage of the time is spent doing the handshake for the request as well as processing time on the web server to open the file and send it. These cases have lower impact at the larger file sizes. For total application layer data, we can see that as file size increases, the data sent increases as well with the header amount taking up less and less percentage-wise, with about .3 of the data being headers for each.

CoAP

CoAP initially was very confusing as there is not much support for it online. We ended up using a library known as aiocoap for python which worked perfectly. For the results we can see that throughput peaked with the 10kb file. This makes some sense as the 100B file is slow for the same reason as HTTP but at larger file size, CoAP sends the file in 1024 byte chunks and each of those requests require its own handshake, so the bigger files have to do more handshakes. When attempting to get throughput data, at 1MB and 10MB files we sometimes would get a reset message to send which then would send the program into an infinite loop. To fix this, the larger results were split across two runs and averaged the results together. For total application layer data, we used the sys package to measure the size of the python response object. The main issue with this method is python handles the files as references so it is not included in this number but this number allows us to show that the headers for CoAP are approximately 300 bits which is about the same as HTTP. The interesting thing is the header size does not change whether the file is sent blockwise or not. Initially when doing the measuring for HTTP and CoAP, our implementation was backwards having the client send the file to the server, for CoAP this increased the number of errors occurring but overall there was not a big difference in throughput which direction the files were traveling.

MQTT

MQTT QoS 1 and MQTT QoS 2 were implemented very similarly using the paho mqtt python library. Initially, there were issues setting up the docker containers with the eclipse-mosquitto docker image so the image was switched to toke/mosquitto which worked perfectly. The docker compose created three containers, one which hosted the mosquitto broker, one that hosted the publishing client, and one that hosted the subscriber. The client/publisher chunks the files, posts them to the broker on a specified topic, and then the subscriber which is subscribed to the topic receives the data of these files. The throughput was measured as the average time it took to upload each file to the broker. A sleep method was used in the loop for the file uploads to ensure the actual upload time was measured rather than the queue time. The results from the table show that the larger the file size, the higher the throughput for both MQTT QOS1 and MQTT QOS2. MQTT QOS1 throughput was higher for every file than MQTT QOS2 which makes sense because MQTT QOS2 has to transfer a larger header size and has to receive an additional ACK (Pub Release and Pub Complete) compared to QOS1. The additional ACK makes sure QOS2 does not retransfer a file. MQTT throughput was also higher than CoAP due to the time it takes for the CoAP handshake.

Application layer size was measured via WireShark. The packets were transferred to a local broker at 127.0.0.1 to measure the header size. For MQTT QoS

1, the application layer data per file was highest for the first file and then about the same for the next three file sizes. This makes sense because the header for MQTT is about the same size for every file so proportionally it affects the size of the first file the most. The application layer data per file was about the same for the rest of the file sizes because the header did not significantly change the size of the files sent. For MQTT QoS2, there was the same trend of the application layer data per file being a lot larger for the first file and then being around the same for the rest of files, but the application layer data per file was larger for every file since the QoS2 header size is bigger.

2. **Discuss which protocols perform better in what scenarios, investigate and describe why, and provide convincing arguments to justify your observations**

HTTP

For a single connection, client/server, HTTP provided the easiest use and wide support. At anything larger than the 100B size it had a very high throughput. The packet sizes are about the same for both CoAP and HTTP and both support modern encryption standards so the difference between them is minimal on that front. HTTP is only usable where there is a single connection between the two computers. Neither of these protocols will perform well in a subscriber/publisher situation.

CoAP

One notable feature of CoAP is its focus on constrained devices and constrained networks. It focuses on keeping the demand for clients low and also has built-in IP multicasting, a feature that other protocols typically need some form of extension to perform. CoAP is definitely tailored towards the IoT network space in its design motivations. While HTTP and CoAP are very similar in their protocol (they are both REST and use the client/server model, for example) the scenario of having multiple groups of devices (say, many smart light switches in a house, or multiple embedded sensor units all communicating to one hub) is not something HTTP supports, and this constitutes the primary use case difference between the two protocols. Another difference between CoAP and HTTP that is relevant to the course material we are currently learning is that HTTP typically uses TCP, while CoAP typically uses UDP. Again, this speaks to what each protocol is tailored towards, CoAP uses UDP, so it needs pings frequently to keep the connection open, which makes sense in an IoT space, while HTTP only needs a ping every 15 minutes or so due to TCP, making it more efficient for connections that need to last longer.

MQTT

MQTT is the best in scenarios in which one device needs to communicate with multiple devices in a publish/subscribe model. This is because MQTT is quick and it does not need a handshake. It has a broker which can deliver a message from a client to multiple other subscribers with its publisher, broker, subscriber system, and can implement a hierarchy of topics which allow subscribers to access different levels. It is lightweight and binary compared to HTTP which means it has smaller packet overhead as well. It also has time decoupling between publisher and subscriber so they do not need to be running at the same time. These are all highly desirable features in IoT operations. QoS 1 is best used when a message needs to be delivered but it does not matter if the message is delivered more than once. The message is guaranteed to be delivered because an ACK has to be sent back to the client and then it stops sending the message. This would be used over QoS 2 in situations in which speed is valuable because QoS 1 has less ACKs and a smaller header so it sends data faster. QoS 2 is used when the speed trade-off is deemed acceptable and a safer quality of service is needed since each message is guaranteed to be delivered only once. This guarantee is because the first ACK is sent as a PUBREC which allows the sender to stop storing the message and then the second ACK sent is a PUBCOMP, ending the sending process, after the sender sends a PUBREL allowing the stored state to be removed in the receiver's storage.

MQTT and CoAP have some fundamental differences. MQTT uses TCP whereas CoAP uses UDP. MQTT is a many-to-many protocol and allows multiple clients to pass messages through a central broker. Publishers and subscribers are decoupled and the broker decides the best route for transferring information. For IOT applications, MQTT proves to be very useful as a communication bus for live data.

CoAP on the other hand, is a one-to-one protocol for transferring information between a client and a server. CoAP proves to be useful as a state-transfer protocol, and is not completely event-based. CoAP also has in-built discovery features, allowing devices on the same network to find each other and devise ways of transferring data. In MQTT, the message formats must be strictly specified at both the publisher and subscriber side for communication to take place.

Comparing MQTT and HTTP, the throughput for MQTT is much higher than that of HTTP (as shown in the table above). MQTT is a data-centric protocol, whereas HTTP is a document-centric protocol. Even if one publisher goes offline in a MQTT system, the publisher-broker-subscriber model will still continue to function. This is not the case with HTTP, which runs on a client-server model. However, HTTP has much more support when it comes to the web. In the end, though HTTP is more extendable, the response time, throughput, and low battery and bandwidth consumption of MQTT make it a better protocol for IOT devices.