**HW #2: MQTT CoAP and HTTP Protocol Analysis Group 8**

**Team Members:**

1. Stepan Kalinin skalini@ncsu.edu
2. Connor Smith cpsmith6@ncsu.edu
3. Sagar Hirenallur Prasannakumar shirena@ncsu.edu
4. Rishab Gujarathi rgujara@ncsu.edu

|  |  |
| --- | --- |
| **Percent Contribution** | |
| Stepan Kalinin | 25% |
| Connor Smith | 25% |
| Sagar Hirenallur Prasannakumar | 25% |
| Rishab Gujarathi | 25% |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **TASKS** | Stepan Kalinin | Connor Smith | Sagar Hirenallur Prasannakumar | Rishab Gujarathi |
| MQTT QOS1 | Code |  |  | 50% | 50% |
| Debug |  |  | 50% | 50% |
| Report |  |  | 50% | 50% |
| MQTT QOS2 | Code |  |  | 50% | 50% |
| Debug |  |  | 50% | 50% |
| Report |  |  | 50% | 50% |
| CoAP | Code | 100% |  |  |  |
| Debug | 100% |  |  |  |
| Report | 100% |  |  |  |
| HTTP | Code |  | 100% |  |  |
| Debug |  | 100% |  |  |
| Report |  | 100% |  |  |
| Report | | 25% | 25% | 25% | 25% |

**1. Objective**

For this task, we will be evaluating three communication protocols - HTTP, CoAP, and MQTT (with QoS 1 and 2) - that are commonly used in IoT. Our objective is to determine which protocol is most suitable for transferring different sizes of data, how long it takes to transfer the data, and how much overhead is generated by the header for each protocol.

**2. Description**

**2.1 MQTT**

**Development**

To test the MQTT protocol, we used three computers - one as a publisher, one as a subscriber, and one as a broker. We implemented the subscriber and publisher using the paho-MQTT library in Python and used the open-source MQTT broker Eclipse Mosquitto. Wireshark was used to sniff the packets and calculate the overhead and application-level data.

To calculate the transfer time, we recorded the start time at the publisher and subscriber ends, and then calculated the difference between them. We also captured packets at the publisher end to ensure all packets were captured. We used a Wireshark packet transfer in Python to read the packet data and calculate the total application layer data transferred. Additionally, we sent the end time from the subscriber to the publisher via MQTT to calculate the transfer time at the publisher end.

**Observation**

We discovered that increasing the file size resulted in higher throughput, as the same-sized file was sent multiple times from the sender to the receiver. However, transferring larger files fewer times caused some delays since the underlying protocol is TCP. Additionally, sending the file in a single message resulted in increased throughput with increasing file size. The overhead decreased as file size increased since the entire data was sent in a single MQTT protocol, resulting in similar overhead for each message. We also noted that the overhead of QoS 2 was higher than QoS 1 due to the Publish Release message sent from the sender to the receiver, contributing to the overall overhead. These observations provide valuable insights for selecting the most appropriate IoT protocol based on the data size, transfer time, and header overhead.

**2.2 CoAP**

**Development**

For we used a aiocoap library, both for the server and the client. The server is basically derived from the example server.py in the aiocoap repo, while the client uses aiocoap library to make requests to it in a similar fashion to the popular requests library for HTTP. Blockwise transfer was confirmed using debug logging.

Time measurement is done using time.perf\_counter\_ns, and using that data and sizes from the received message calculates the required statistics (and a bit more on top).

**Observation**

The experiment was done on two machines, with one of them acting as a hotspot, so the two were connected over WiFi. The results indicate that the protocol is good for small files, having trouble reaching throughput over 2kbps, likely capped by the fact that an RTT is spent for every block transferred. In terms of overhead, it is pretty close at ~10%, slowly growing as the number of bytes needed to represent block index increases with larger file sizes.

**2.3 HTTP**

**Development**

Testing the HTTP protocol was done in a fairly similar way to the CoAP protocol. The server is done using Python’s built-in server command to host a simple HTTP server at a given host and port number. Localhost is used for testing on port 80. The client file is ran with option command line arguments for host/port if they are different than localhost:80. A HTTP GET request is sent a specified number of times. A timer is used before and after the GET request is made to track the time required for each GET request. The size of the header and payload is determined by finding the point in the byte string where the headers end and the payload begins. This information allows us to find the size of the header and payload. The Measurement class is also used to return the time, total size, and payload size of each file that is sent. Statistics are then printed to the console.

**Observation**

Examining the results for throughput, we can see that throughput slows considerably for files of a really small and really large file size. HTTP excels with files or medium file size, with throughputs in the 10s of thousands of kbps. For large files, this may be due to network bottleneck constraints. Another possible explanation could be due to the maximum size of each packet, so the data needs to be fragmented more, which causes slower throughput. For medium sized files, we don’t reach the network bandwidth capacity, and the overhead on the packets is comparatively minimal, so we see much higher throughput values. This is also backed up by examining the overheads of the different files. In general, the smaller sized files have significantly more overhead than the larger files.

**3. Comparison**

In looking at throughputs of the 100B file for the four protocols, we can observe that CoAP does a significantly better job than the 3 other protocols. This is possible since CoAP is built in UDP instead of TCP, allowing for significantly faster throughputs. On the flip side, with the 10MB file, CoAP falls far behind the other protocols in throughput. This could be for several reasons. Firstly, since CoAP is built on UDP, there may be issues with packet loss that are exacerbated with larger files that are not present in the other protocols. Second, CoAP is designed to transfer smaller payloads, which is why it is so popular with IoT applications. Fragmentation slows down the transfer process with each fragment also needing its own header. Lastly, bandwidth constraints could also make the previous 2 reasons even worse. When examining the jumps in throughputs between the 10kB file and the 1MB file, there are significantly increases of magnitude in throughput for all the protocols, except for CoAP. This is because the overhead for all the other protocols becomes drastically smaller by comparison as the file size increases. Since CoAP is designed for small payloads, the overhead will stay relatively the same, even as the file size increases because of fragmentation. The overhead decreases for the other 3 protocols because the payloads can transmit more data. Looking at the overhead for the 4 protocols, we can see that CoAP has a much smaller maximum payload size compared to the other protocols. We can see that CoAP reaches its maximum payload size in between the 100B file and the 10kB file because the overhead stays relatively constant after the 100B file, whereas all the other protocols have their overheads decrease by magnitudes of 10 even up to the 10MB file. Our last observation we have made is that MQTT QoS1 is substantially faster than QoS2. This is because QoS1 is sending only 1 acknowledgement during data transmission, whereas QoS2 sends multiple acknowledgements to ensure the safety of the data transmission, at the cost of speed.