KU LEUVEN
Paper submitted within the scope of the Bachelor's Project
Bachelor of industrial Sciences
KU LEUVEN - Faculty of Engineering Technology, Campus Group T Leuven

Theoretical Study of Session and Application Layer Protocols

Maxim Yudayev
KU LEUVEN - Faculty of Engineering Technology, Campus Group T Leuven

Coach: Tim Stas
KU LEUVEN - Faculty of Engineering Technology, Campus Group T Leuven, tim.stas@kuleuven.be

1 Introduction

The paper aims to profile several session and application layer protocols with the goal of providing insight into the recommended use case of each within the embedded device realm. The discussed protocols include MQTT, CoAP and HTTP. The three protocols are benchmarked, then compared against each other and raw UDP/TCP packets with application-specific session and application code.

For the purpose of the profiling, a constrained device such as ESP8266 (a 32bit ARM RF MCU) and a PC are to be used to best mimic real use case wireless interfacing to a constrained embedded device over the network. The two devices take on a role of server or client, depending the application protocol employed; the setup is then tested using a multi-threaded "C" program which creates a desired number of thread-clients, each having their own socket and hence acting like an independent client as far as the constrained device is concerned.

2 Requirements & Background

The study assumes a scenario where a client, designing a solution to communicate between an embedded application and the user, seeks an optimal way to maintain sessions and present data to either side of the to-be-designed network. Hence, the choice of a standardized protocol or the architecture of an application-specific implementation may be critical to the performance of the final product. The goal of the study is to list criteria of a possible application for each aforementioned protocol, which would be optimal for the given protocol and allow the user to choose a solution based on it for their end application.

Protocols are standardized and are tested rigorously by time, making them an appealing and a logical solution for a networking application. Although due to their abundance, it is frequently overwhelming to choose an optimal one for a given use case. Despite of that, protocols offer significant advantages over custom solutions by being widely adopted, back-/future-compatible, being natively implemented in other applications (like web browsers), faster time-to-market, having existing tools and libraries useful for design and debugging of an application.

Table 1 presents the summary of the differences between the studied protocols.

	HTTP	CoAP	MQTT	Raw UDP	Raw TCP
Model	Server-Client	Server-Client w/ Proxy option	Client-Broker (PubSub)	NA	NA
Application Header	Unlimited size	4 bytes	2 bytes	None	None
Paradigm	RESTful	RESTful	Connect, Subscribe, Publish	NA	NA
Transport Layer	TCP	UDP	TCP	UDP	TCP
Port	80	5683	1883	NA	NA
Data Format	html, text, json, binary	xml, text, json, binary	binary	binary	binary

Table 1: Protocol comparison.

HTTP and CoAP use structured data formats which are inferred at runtime, aka "parsing", upon response receival. This parsing is an extra step and implies that the communicating device must check character after character (byte after byte) in the payload to understand what is requested, hence the model is intrinsically limited by the speed of parsing. It is without a doubt that depending on the size of the payload/header, this parsing step becomes troublesome for a constrained device.

Worth noting is the fact that in networks consisting of embedded devices, it is most of the time possible to

transfer only single IP frames to and from the device since most of the embedded devices employ single frame networking buffers. This frame buffer is used by the embedded device for both, Rx and Tx operations, implying lack of possibility to simultaneously store an inbound and an outbound frame. Additional aspect is the limited amount of RAM, which sets a limit on the software-based stack size required for maintaining connection information/stacks in connection-oriented networks meaning that each embedded device can only maintain a limited size of active connections. Hence, the approach to application protocols for embedded device inclusive systems is different to that of a general computer.

2.1 MQTT Background

MQTT (Message Queueing Telemetry Transport) creates abstraction in communication, allowing to relay all traffic through a broker, or a number of brokers. This allows to move heavy network traffic to the more powerful broker server instead of forcing a constrained device to directly accept connections to a potentially large number of clients. The protocol works by keeping track of publishers and subscribers to a particular "topic" (a human readable hierarchical string alike a URI), sending messages to clients interested in the particular topic when a new message is available. As the name suggests, MQTT is a telemetry-oriented protocol, that is it was developed to power things like sensor networks and targeted network implementations with simplistic constrained embedded devices and personal computing/server machines acting as clients. *Figure 1* depicts the TLD of an MQTT logical topology.

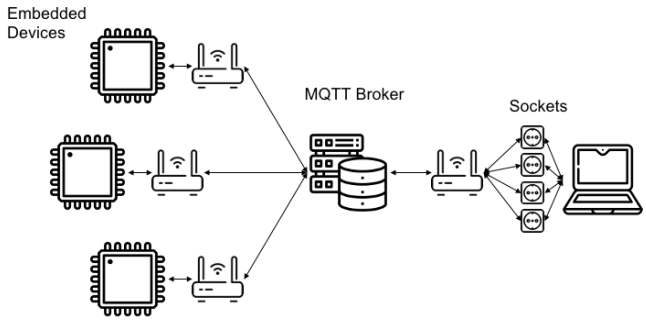

Figure 2: MQTT logical topology.

As shown in *Table 1*, MQTT is a lightweight protocol with a minimum header of two bytes and uses TCP/IP as the transport layer protocol. That means the system is connection-oriented and maintains sockets, that is a combination of the IP address and port number (i.e 192.168.0.219:56384), which is dedicated to the communication with a certain peer. In MQTT, the broker accepts all traffic from every client on the same socket, that is the one attached to port 1883. Despite of that, the broker maintains connection to all of the clients after initiating a connection via the typical TCP three-way handshake (or four-way if ACK frame is sent separately and not piggybacked). After the handshake is completed, the MQTT broker maintains an internal data structure of clients with only the port 1883 active instead of creating separate sockets corresponding to each one separate client. This is, however, internal design choice and shall depend on the system software architecture, abstract from the actual protocol implementation, and, in a way, irrelevant for the topic of this study.

2.1.1 Message Format and Implications

MQTT sends binary payload. That means that each of the members of communication must know exactly which byte is what, compared to structured payload of HTTP or CoAP discussed later, where information is inferred at runtime using "tags". Having binary payload allows more compact messages and shorter transmit times, both a desire for constrained devices, but lacks portability or compatibility in case the designer wishes to transmit different data; in which case, the firmware of the embedded devices will have to be updated in order to process/understand new data types. This is, however, not a concern in real-life scenarios as constrained device in, for instance, sensor networks are used as publishers that post updates of their sensor reading at a certain periodicity and do not require bidirectional communication. If the bidirectional communication is required, it is often preprogrammed to support processing of command/control packets via the network at design time and remains fixed for the lifetime of the device. Embedded devices are never general purpose anyway, hence this is not an issue from the engineering standpoint.

2.2 HTTP Background

HTTP (Hypertext Transfer Protocol) is a widely adopted application protocol powering web browsers and is the closest to a generic user application protocol due to its mass market penetration. HTTP is an application protocol designed for mass transfer of arbitrary structured data among network nodes. Versatility of the protocol was the priority at its conception time and is indicated by the lack of hard coded header size

constrains; although browsers typically limit header size to 16kB. Referring to the topology of *figure 2* HTTP applications may have optional Proxy servers which cache frequently used or recent data, allowing to limit traffic for identical information to the end embedded device nodes, minimizing congestion since the relaying of the constrained device born information to a potentially large number of clients is carried out by a "beefier" dedicated server on the network. In an instance that the proxy server does not have requested by the client data, based on the application architecture, it either requests itself information from the embedded device and relays later to the client or the client resends the request, but now to the end node directly. Similar to MQTT, embedded devices in HTTP push updates to the proxy server periodically via their maintained TCP connections.

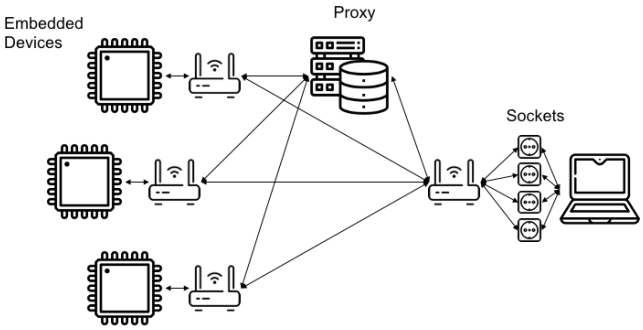

Figure 2: HTTP and CoAP logical topology.

Sensor networks are only a part of potential applications of HTTP and are by no means the only target of the protocol. As mentioned previously, HTTP is a mass volume structured data transfer protocol whose main audience is clients running web browsers and web applications, requiring vast graphics, performance and massive data flows; none of which resemble the constraints of a sensor network. In fact, constrained devices running over HTTP are simply a "porting" method (abstractions to support higher level protocols) to conveniently run web-browser-based applications using RESTful conventions as if the constrained device was a normal HTTP general purpose server. This allows simpler design stage and much beloved high-level programming languages for an engineer, and gets one quicker to market.

2.2.1 Browser Compatibility

Web-browser compatibility, however, means that a constrained device must adhere to the communication approach of the ported protocol; for a constrained device this means considerable processing trouble and overhead to deal with, potentially making such a network less performant and less efficient for the embedded device.

2.3 CoAP Background

CoAP (Constrained Application Protocol), in simplest terms, is a lightweight HTTP, oriented at specifically communication with embedded devices.

2.3.1 Differences to HTTP

Given that the protocol targets constrained devices, it has a small fixed size header of only four bytes and requires the payload combined with the application header to not exceed the size of a UDP datagram. These differences are targeted at complying with the constraints of embedded devices, mentioned previously in the MQTT section. Since payload in both HTTP and CoAP are structured, they require to be parsed by the end node, although compared to HTTP, CoAP does not require parsing of the header as it is fixed size and standard, making processing of requests potentially significantly larger than in case of HTTP. CoAP also runs over UDP, meaning that the network is best-effort and does not mind lost packages. This is potentially okay for non-critical applications, although CoAP applications may employ acknowledgements on the application layer rather than the transport layer for different levels of quality of service (QoS) in an instance that the application requires guarantee of message reception.

With improving networking technologies and wider coverage of existing network infrastructures, use of TCP/IP appears to be excessive as only a tiny fraction of messages requires retransmission, and hence may benefit from TCP. This fact acts as a standpoint for utilization of connectionless UDP as the transport layer protocol, while carrying out connection and session maintenance at higher OSI layers, among which benefits is reduced communication overhead, equaling increased network throughput, due to lack of multi-way handshakes present in connection-oriented TCP.

2.4 Research Hypothesis

Transport layer overhead and payload format of dynamic browser-compatible protocols limit the throughput of the constrained embedded device.

3 Design & Implementation

Interfacing with ESP8266 is done via ESP8266_RTOS_SDK[1] library, which provides API for all of the protocols of interest for the study. On the PC side of things: POSIX threads are used for the multi-threaded "C" client program; "curl" is used as the HTTP request client; "libcoap"[2] is the client library for CLI CoAP requests; "mosquitto"[3] is used as the MQTT broker. To profile the protocols Wireshark application was used to capture network traffic, which was then analyzed in Wireshark itself or in Microsoft Excel when needed. Devices in the study communicate on WLAN and after frame analysis in Wireshark it was noted that the libraries implemented in the study do not route packets through default gateway by default, but can do so directly to the designated peer if they have corresponding MAC address of the device, obtained at the beginning of the routine via an ARP request.

3.1 Top Level Design

The study is made using the following setup. It describes physical topology and connections; based on the protocol, the transport layer may change or there may be devices added logically in between the embedded device and the client PC. However, from the physical point of view, *figure 3* is the general layout.

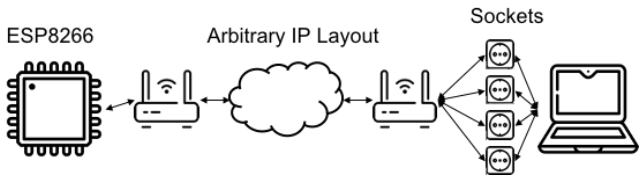

Figure 3: General study topology.

The study is profiled with a variable number of clients. Other than the ESP8266 itself, in the network, for each protocol, there is 1, 10, 100 clients. In each tested protocol, 100 single frame messages were transmitted.

3.2 Development

3.2.1 MQTT

Using mosquito broker daemon on the test PC, using the test setup of *figure 4*, a persistent MQTT server socket on port 1883 was created, now accepting network traffic.

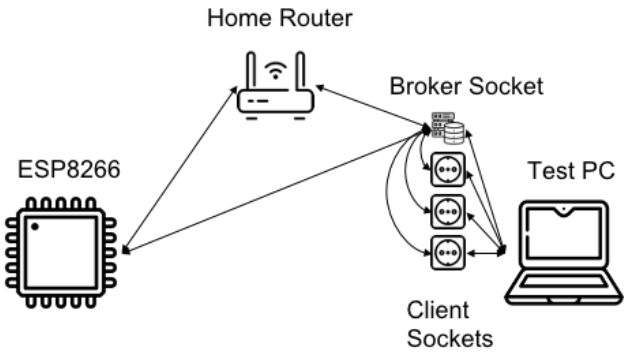

Figure 4: MQTT setup.

In order to emulate clients of a possible MQTT implementing application, a multi-threaded C program generated a number of threads, each with own associated socket using the mosquitto library; each of these threads subscribes to a particular topic and publishes to the "connections" topic to acknowledge others of their connection status. Similarly, the embedded device subscribes to the broker. For the test purposes, it was investigated how the network behaves when a certain number of clients publish a "control" message to the topic the embedded device is subscribed to, as well as how the network behaves when the embedded device publishes a message to the "sensors" topic this certain number of emulated subscribers is subscribed to.

Fixed header, present in all MQTT Control Packets
Variable header, present in some MQTT Control Packets
Payload, present in some MQTT Control Packets

Figure 5: MQTT packet composition. [mqtt-v3.1.1].

Bit	7	6	5	4	3	2	1	0
byte 1	MQTT Control Packet type				Flags specific to each MQTT Control Packet type			
byte 2…	Remaining Length							

Figure 6: MQTT fixed header byte diagram. [mqtt-v3.1.1].

At the very minimum, an MQTT header is two bytes long, corresponding to a payload plus remaining header of between 0 and 127 bytes. If application data exceeds this size, additional one, two or three "Remaining Length" bytes can be appended, where the cleared 7th bit indicates current length byte as the final "Remaining

[1] https://docs.espressif.com/projects/esp8266-rtos-sdk/en/latest/get-started/ - ESP8266 SDK

[2] https://libcoap.net - CoAP client

[3] https://mosquitto.org – MQTT broker daemon-library

Length" specifier. All length specification options are depicted in *figure 7*.

Digits	From	To
1	0 (0x00)	127 (0x7F)
2	128 (0x80, 0x01)	16 383 (0xFF, 0x7F)
3	16 384 (0x80, 0x80, 0x01)	2 097 151 (0xFF, 0xFF, 0x7F)
4	2 097 152 (0x80, 0x80, 0x80, 0x01)	268 435 455 (0xFF, 0xFF, 0xFF, 0x7F)

Figure 7: MQTT variable header "Remaining Length" specification options. [mqtt-v3.1.1].

In the test setup, only one length byte is used since the protocol profiling in this study was done using messages that fit in a single IP frame. This allowed to have results that can be compared against each other in spite of the underlying differences in the transport layer (i.e it would be hard to compare protocol performances against each other if the amount of sent frames differed). When an MQTT message that does not fit in a single IP frame (maximum message size is 268MB) is sent, it is fragmented into multiple TCP packets and later reassembled on the receiver side; there, the indication of the "Remaining Length" explained above by the 7th bit is crucial for the reassembly so that the receiver knows how to piece fragmented data back together and what frames belong together to the same transaction.

The profiling of MQTT was also done for al three QoS options, where the QoS indicates how and whether reception of messages shall be acknowledged by the parties. Each increasing level of QoS indicates additional overhead, with a minimum number of eight IP frame transactions for the QoS of two, the highest where recipient receives acknowledgement exactly once, instead of only a minimum of two for the QoS of zero. The word "minimum" here is mentioned due to the transport layer acknowledgments of TCP, whereby failure to receive acknowledgment on the transport layer for a frame within a certain time frame results in retransmission of the frame.

3.2.2 CoAP and HTTP

The CoAP and HTTP test setup did not employ a proxy. Similar to MQTT, the test PC generated threads using a C program that sent out requests at fixed periods to the constrained device using libcoap or curl, for CoAP or HTTP scenarios, respectively.

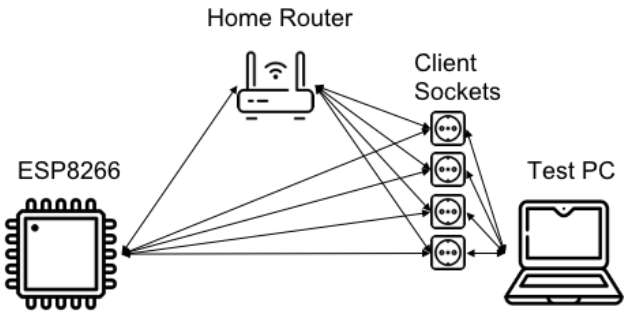

Figure 8: CoAP and HTTP setup.

Alike the MQTT scenario, both, in CoAP and in HTTP the embedded device and the sockets of the test PC communicated directly, not through the default gateway, over WLAN if they already established a record of the peer's MAC address. This step was done in case of the embedded device immediately prior to subscribing to the MQTT broker via an ARP request, yellow *figure 9.*

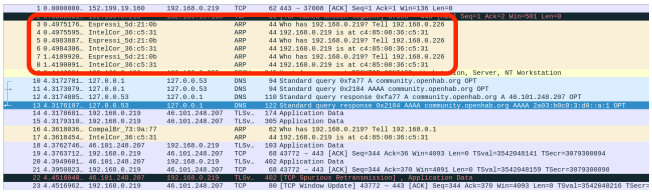

Figure 9: ARP request by ESP8266 (Espressi_5d:21:0b).

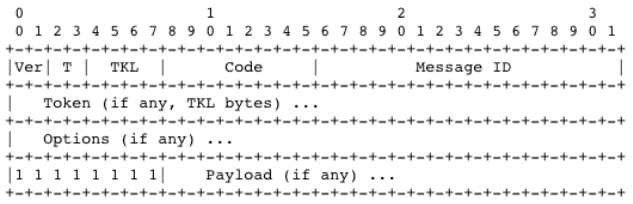

Figure 10: CoAP header byte diagram [coap-rfc7252].

The maximum size of a CoAP message is one UDP datagram payload size minus CoAP header size (header size includes the padding byte of ones to indicate non-empty payload). If a message is smaller than the limit of a UDP datagram payload, it is not problematic since the UDP header specifies the length of its payload; the size of the datagram can thus be inferred from it at runtime, *figure 11,* where the length field is the UDP total size including header and data.

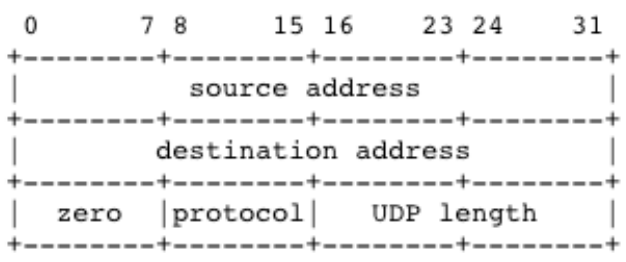

Figure 11: UDP header byte diagram [udp-rfc768].

However, due to size changes to the underlying IP frame size, as the result of insertion of additional headers during travel in the network, the actual UDP payload data may be “chopped off”, thus often in practice a 512 byte UDP payload limitation is chosen to provide sufficient margin for protection in such instances.

HTTP has practically no header limit unlike the protocols compared against in this study ([http-rfc2616]). Each field in the header is in ASCII text format, hence parsing is required to interpret the control commands and all the additional supplementary data, including the payload. Since HTTP uses TCP as underlying transport layer protocol, arbitrary size of data can be fragmented and sent over the network, implying lack of limitation to the format or quantity of the to-be-transmitted data.

In the study, to emulate a real-life sensor network, the constrained embedded device acts as the server, which can be queried using CoAP or HTTP for current readings, separately in each test setup. Hence, the device acts as a server and a test PC socket acts as a client. An advantage of using HTTP for the sensor network, as thoroughly described in section two, is an ability to communicate with the device via a standard web browser, not requiring installation of additional libraries, programs, and usage of low-level languages, both often too complicated for a general public user, or a novice engineer.

4 Testing

Analysis of protocols was done primarily in Wireshark using filters and its “Statistics” tools; functions employed included “Conversations”, “TCP Stream”, “Flow Graph”, “IO Graph”. To perform counting or logical operations on aggregated data, Microsoft Excel was used on filtered exported data after preliminary format correction in text editor.

During the tests, the initial hypothesis of delays due to significant overhead of maintaining connections in connection-oriented transport layer networks, together with overhead of numerous ACK exchanges was noticeably observed. All of the MQTT test setups ran for the expected 100 seconds, while CoAP and HTTP test took up to 36 minutes (HTTP, 10000 total requests from 100 clients). Study test summary is provided in *figure 25*.

In addition, the claim earlier in the paper of network link quality not being the main concern in today’s networks was correctly confirmed by the test results.

4.1 MQTT

In the following figures, presented are the application parameters of each MQTT QoS option. It was found based on the packet structure of *figure 12* that “mosquitto” stuffs multiple enqueued MQTT messages into single TCP frames. However, ESP8266 MQTT implementation does not account for this and reads only the very first MQTT message, while discarding the rest; this is the only reason packet loss is so high in the test. Hence, for a well-designed application, QoS 0 will be very competitive in reliability to QoS 1 and 2.

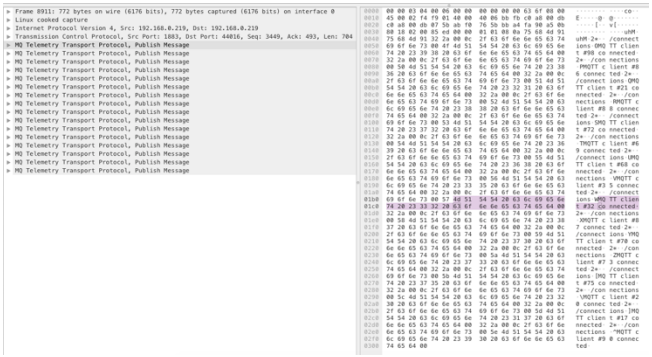

Figure 12: MQTT broker stuffing enqueued messages in single TCP frame.

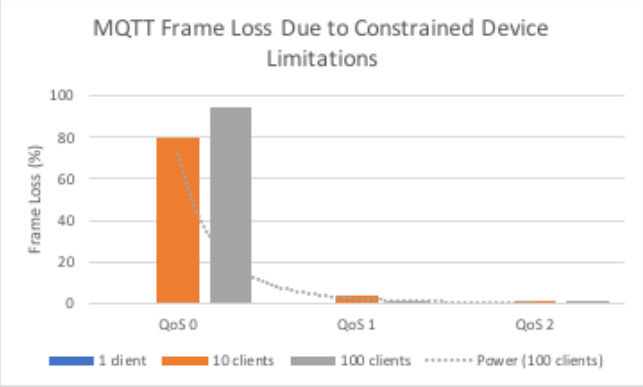

Figure 13: MQTT – ESP8266 loss of received MQTT messages appended sequentially in one TCP frame due to protocol implementation.

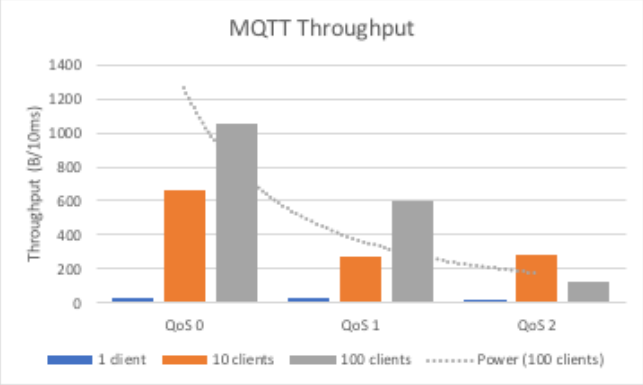

Figure 14: MQTT – Network throughput of each QoS setting.

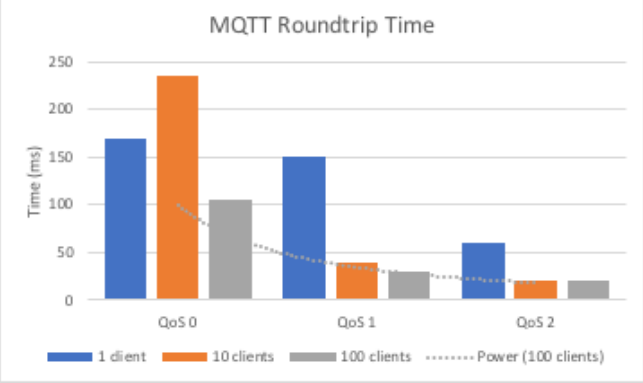

Figure 15: MQTT – time between two consecutive TCP ACK in a conversation.

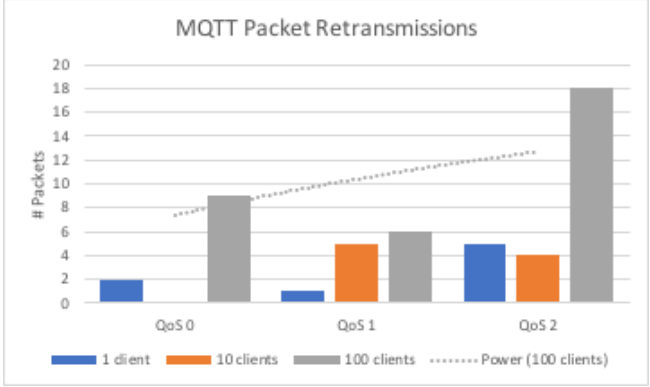

Figure 16: MQTT – Retransmitted TCP frame number for each QoS option.

Such performance and low overhead of MQTT can be explained by the fact that the broker in MQTT networks relays all the traffic through itself. Instead of every client asking the poor constrained device “Are we there yet?”, the broker settles the clients and notifies them of updates when it receives one from a publisher, the constrained device in this case. This dramatically limits the require amount of message exchange for achieving the same goal and results in lower throughput; better replaced with “better usage of network throughput” since throughput is a parameter of the underlying network, rather than an application on top of it.

Another notable factor improving the network efficiency is opening and maintaining persistent TCP connections for clients, most important in case of the embedded device, which removes the need for continuous handshaking between the peers (i.e HTTP) where this overhead is arguable in comparison with the size of useful payload.

With increased level of QoS, more of smaller sized ACK frames are sent, with each conversation being delayed by ESP’s processing and resulting in a busy wait type of situation. As the result less bytes in air are present and hence the throughput is lowered again. The time it takes to send the same number of packets in the network, when comparing across QoS, increases exponentially as shown in *figure 17*.

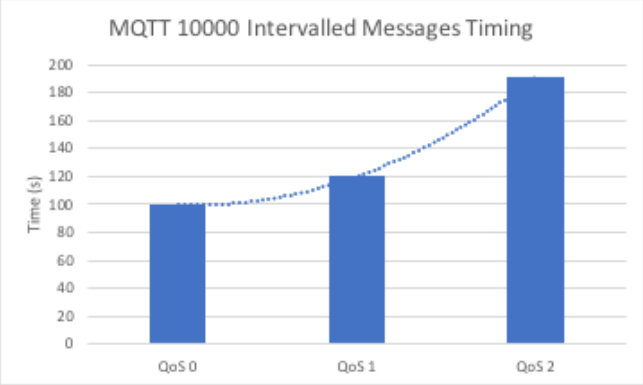

Figure 17: MQTT – Time required for sending 10000 messages by 100 clients at 1 second intervals at each QoS.

4.2 CoAP

CoAP significantly outperformed HTTP with minimal transport layer communication overhead and lack of handshakes (both attributes of using UDP over TCP) and tightly competed with MQTT though falling behind orders of magnitude on conversation times since the communication is now carried out peer to peer instead of relaying through a 3rd party (use of proxy for such enhancement was out of the scope of the study though guarantees to dramatically improve the performance of the final system). *Figure 22* indicates retransmission for CoAP; although CoAP uses UDP which is best-effort transport layer protocol, connection/session management is performed on the application layer. Flow graph in *figure 18* shows example from the study.

Figure 18: CoAP – Flow graph. Only 4 total frames in a conversation.

In case no response (ACK) was received by the client in a specified time, the application will resend the request frame up to a user specified number of times, even with variable time windows. This ability to maintain connection/session on application layer gives more freedom to the designer, but also requires more development effort than using TCP.

This supports another claim of the author of this paper at the beginning of the paper about the advantages of using UDP instead of TCP for connection-oriented applications.

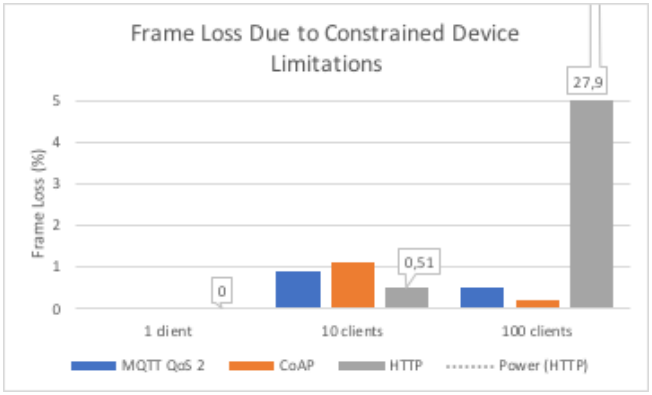

Figure 19: MQTT QoS 2 vs. CoAP vs. HTTP – Frame loss.

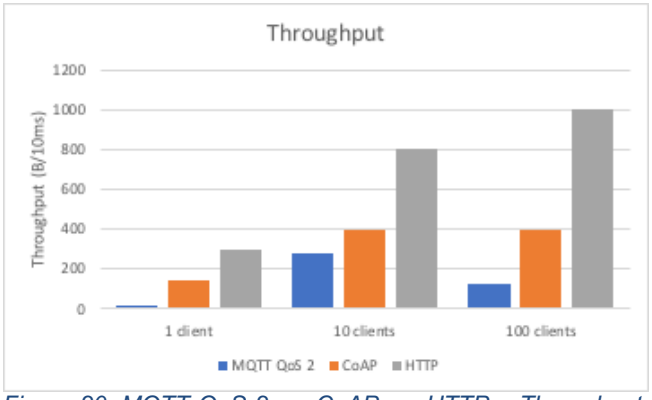

Figure 20: MQTT QoS 2 vs. CoAP vs. HTTP – Throughput. Lower value indicates less occupation of transmission medium.

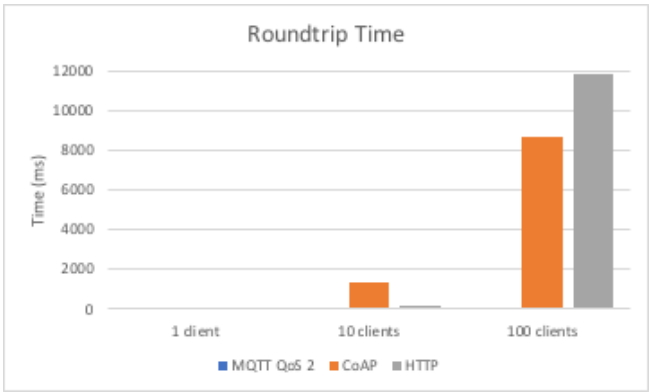

Figure 21: MQTT QoS 2 vs. CoAP vs. HTTP – Mean conversation time. Higher value indicates delays.

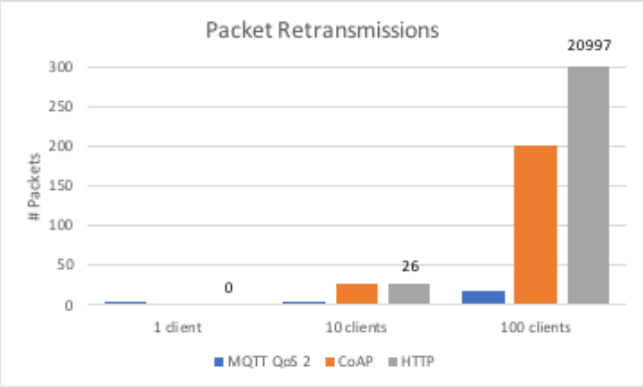

Figure 22: MQTT QoS 2 vs. CoAP vs. HTTP – Packet retransmissions.

CoAP proved to be reliable, lightweight and includes features normally offered by HTTP like caching at a proxy server, structured message data format, URI-like server structure and RESTful querying model using GET, POST, PUT, DELETE methods to interact with the CoAP server running on an embedded device.

4.3 HTTP

HTTP came in the loser in the study as its great functionality is counterbalanced by the overhead in the use case with small payloads for embedded devices where only telemetry like data is to be transferred. At worst case scenarios in the study, HTTP server and client exchanged up to 25 messages for a simple 40-50 byte payload, *figure 23*.

Figure 23: HTTP – Study's worst case single exchange flow graph.

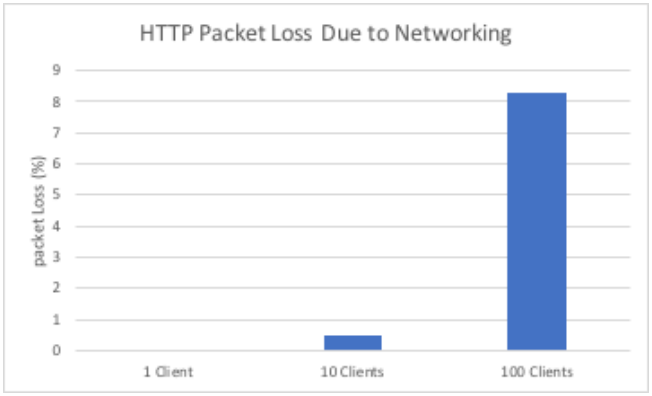

Figure 24: HTTP – Loss of frames exceeding TCP RTO due to unresponsive or faulty connections.

Despite its loss in the study, HTTP allows quick integration opportunity as it provides designer the opportunity to "speak" directly to the web-browser and program the application in high-level programming languages, making this solution the quickest to market. It also remains by far the widest used application protocol for media transfer by volume in modern networks. In addition, if targeted application requires streaming of more than telemetry data, in case of embedded IP cameras, edge computing devices or OTA updatable complex systems, HTTP is an optimal solution to feed the data hungry devices.

4.4 Protocol Comparison

			Packet Loss - Network (%)	Packet Loss - Embedded Device (%)	No. Retransmission	Throughput Mean (B/10ms)	Roundtrip Time Mean (ms)	No. Embedded Device Crashes	Embedded Device Offline (s)
MQTT	QoS 0	1 Client	0	0	2	30	170	0	0
		10 Clients	0	80,2	0	660	235	0	0
		100 Clients	0	94,75182572	9	1050	105	0	0
	QoS 1	1 Client	0	0	1	28	150	0	0
		10 Clients	0	3,74	5	270	40	1	15
		100 Clients	0	1,07	6	600	30	0	70
	QoS 2	1 Client	0	0	5	16	60	0	0
		10 Clients	0	0,9	4	280	20	0	0
		100 Clients	0	0,5	18	130	20	0	32
CoAP		1 Client	0	0	0	140	89	0	0
		10 Clients	0	1,11	28	400	1306	0	0
		100 Clients	0	0,20	200	400	8693	0	0
HTTP		1 Client	0	0	0	300	128	0	0
		10 Clients	0,51	0,51	26	800	200	0	0
		100 Clients	8,3	27,90	20997	1000	11842	0	0

Figure 25: Tabulated collected test data comparing MQTT, CoAP and HTTP

Comparing the protocols to UDP turned out not to be required as it was seen from the analysis of CoAP that it is essentially the most stripped down version of session and application layer protocol running over UDP; in ideal scenario, CoAP conversation would consist of only four packets with multiples of one additional packet for each retransmitted request attempt. The only advantage of Raw UDP is lack of header, a saving of four bytes equaling to a time saving of only 17ns at 2.4GHz and 8b/10b encoding (excluding few possible stuffing bits).

Similarly, for TCP, only two TCP frames are exchanged in MQTT QoS 0 and four in MQTT QoS 2 in each transaction, meaning that the only gain would be a saving of two bytes of MQTT header.

5 Conclusion

Having application layer connection/session managing makes it unnecessary to run over TCP. UDP can be as reliable when accompanied by session managing logic on the higher OSI layer.

MQTT was magnitude times faster than CoAP or HTTP, with HTTP being the slowest.

Performance of a secure connection, addition of encryption, inclusion of communication over WAN will all only decrease the performance of each setup from the results of the study.

Running applications over raw TCP or UDP saves you two to four bytes, while requiring to spend significant effort implementing custom solution; application protocols guarantee to be compatible to other applications on the market and remain flexible compared to a custom solution.

ESP library sends out receive window information after 40ms intervals, likely due to the software tick mechanism of the underlying cooperative scheduler in its real-time operating system. As the result, sender frequently tries to push information when ESP has not yet freed up enough space. Hence the likely reason ESP crashes.

The only case a TCP frame may be lost is if it exceeds TCP retransmission timeout (RTO) period, in which case the TCP connection closes, this is the scenario against which QoS option of MQTT protects against. However, as discussed previously in the paper and as is remarked from the tests, the need for QoS higher than 0 is largely unnecessary as TCP retransmission capabilities will almost surely guarantee delivery of a message, given that the MQTT implementation accounts for stuffed messages inside of single TCP frames unlike the case with ESP. This position is further strengthened by the fact of continuous improvement in network performance, reliability and coverage. Hence QoS in MQTT may be appealing only for “ABSOLUTELY MUST” receive networks which cannot tolerate even a single packet loss, no matter the likelihood.

MQTT enqueued messages which are not immediately transferrable to the subscriber are retransmitted when the recipient device comes back online.

Oddly from the system software standpoint that “mosquitto” MQTT broker uses single socket for communication with all clients instead of assigning separate individual sockets after accepting connections.

Based on the findings in section 4, and to answer the initial goal of the study about what criteria of a project makes it fit within optimal beneficiaries of a particular protocol, the author if the study offers his interpretation of the results and the obtained experience from playing with the protocols.

- Use MQTT when:
  - Large number of clients are interested to know the same information from an embedded device
  - End device has strict restrictions on power use and battery life
  - Application for embedded device is mainly write and not read oriented
  - Consistent demand for publisher’s information
- Use CoAP when:
  - Needed to send structured data of less than 512 bytes
  - No consistent demand or only occasional demand for embedded device’s data is needed
  - In dense networks to minimize channel occupation for inclusiveness of other devices nearby
- Use HTTP when:
  - Prototyping
  - Target application is web browser based
  - Embedded device has high data bandwidth needs
  - Lack of proficiency with hardware and low-level programming

For an interested reader, Appendix includes more captions and statistics from Wireshark for all of the discussed protocols in greater visual detail.

Network captures, as well as application code, for both ESP8266 and test PC are attached together and are available for review on Github *[study-data]*.

6 References

- [mqtt-v3.1.1] Reprinted from "*MQTT Version 3.1.1*" Edited by Andrew Banks and Rahul Gupta. 29 October 2014. OASIS Standard. http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html. Latest version: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html.

- [coap-rfc7252] Reprinted from *"The Constrained Application Protocol (CoAP). RFC 7252"* Z. Shelby, et al. June 2014. Internet Engineering Task Force (IETF). https://tools.ietf.org/html/rfc7252.

- [udp-rfc768] Reprinted from *"User Datagram Protocol. RFC 7252"* J. Postel. 28 August 1980. Internet. J. Postel. https://tools.ietf.org/html/rfc768.

- [http-rfc2616] Fielding, R., et al (June 1999). Network Working Group. *Hypertext Transfer Protocol – HTTP/1.1. RFC 2616.* Retrieved from https://tools.ietf.org/html/rfc2616.

- [study-data] Yudayev, M., (May 2019). https://github.com/maximyudayev/ee5-application-and-session-protocols

7 Appendix

7.1 MQTT QoS 0, 1 client

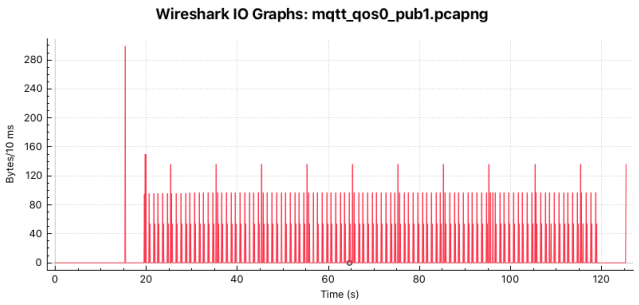

Figure 26: MQTT QoS 0, 1 client – overall traffic.

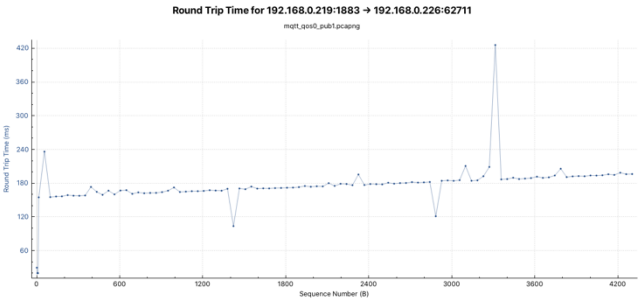

Figure 27: MQTT QoS 0, 1 client – roundtrip time.

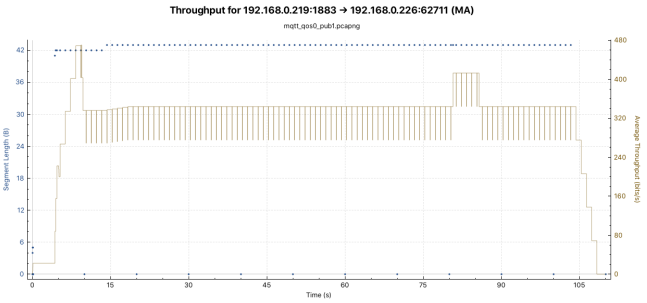

Figure 28: MQTT QoS 0, 1 client – throughput.

7.2 MQTT QoS 0, 10 clients

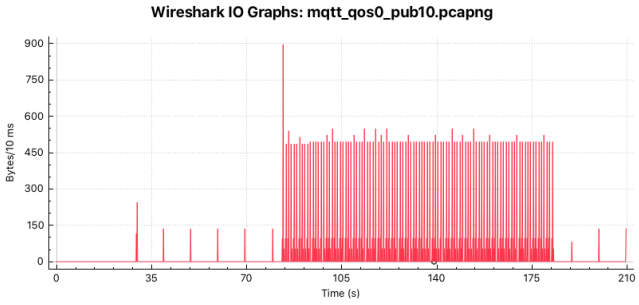

Figure 29: MQTT QoS 0, 10 clients – overall traffic.

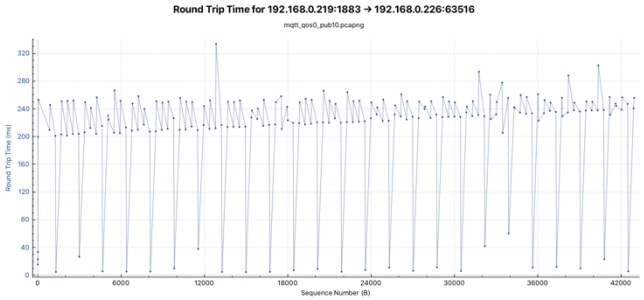

Figure 30: MQTT QoS 0, 10 clients – roundtrip time.

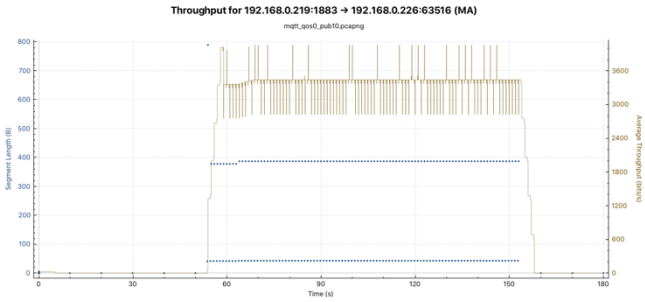

Figure 31: MQTT QoS 0, 10 clients – throughput.

7.3 MQTT QoS 0, 100 clients

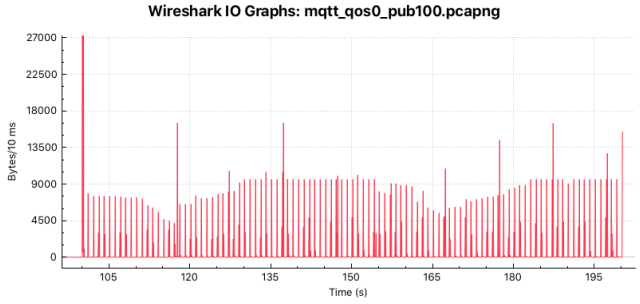

Figure 32: MQTT QoS 0, 100 client – overall traffic.

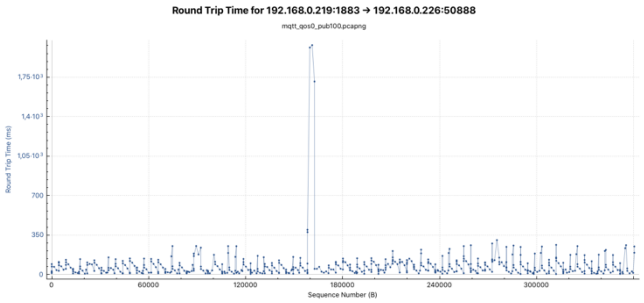

Figure 33: MQTT QoS 0, 100 clients – roundtrip time.

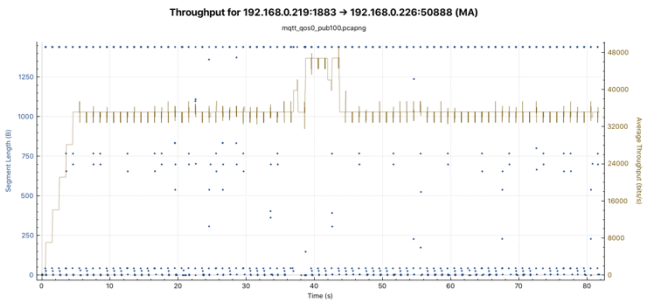

Figure 34: MQTT QoS 0, 100 clients – throughput.

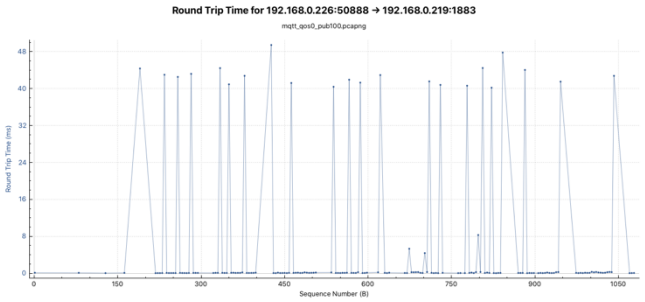

Figure 35: MQTT QoS 0, 100 clients – roundtrip time (ESP to broker).

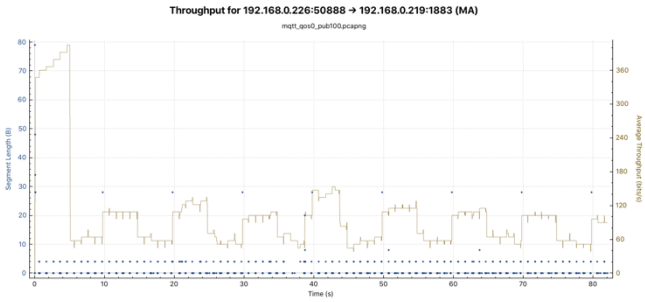

Figure 36: MQTT QoS 0, 100 clients – throughput (ESP to broker).

7.4 MQTT QoS 1, 1 client

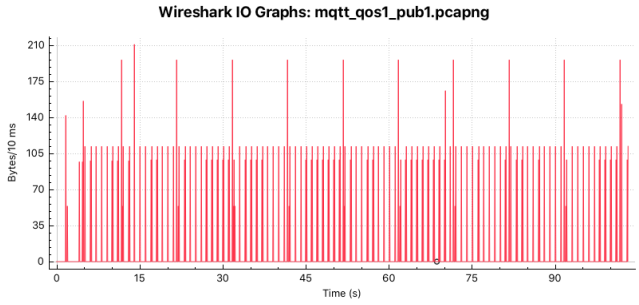

Figure 37: MQTT QoS 1, 1 client – overall traffic.

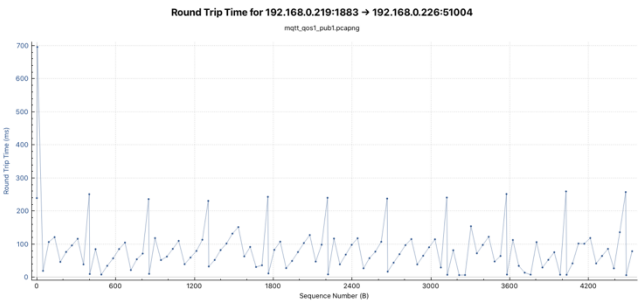

Figure 38: MQTT QoS 1, 1 client – roundtrip time.

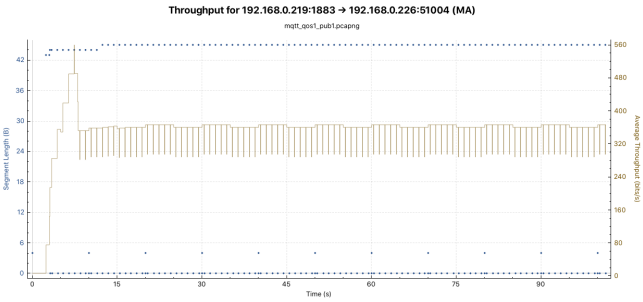

Figure 39: MQTT QoS 1, 1 client – throughput.

7.5 MQTT QoS 1, 10 clients

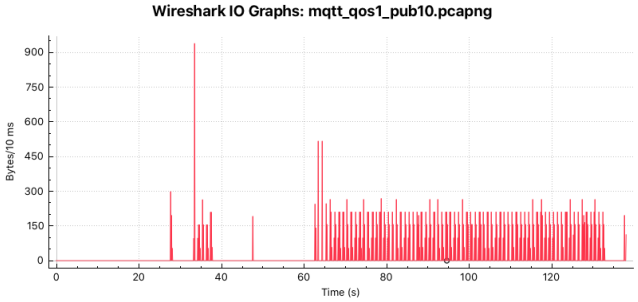

Figure 40: MQTT QoS 1, 10 clients – overall traffic.

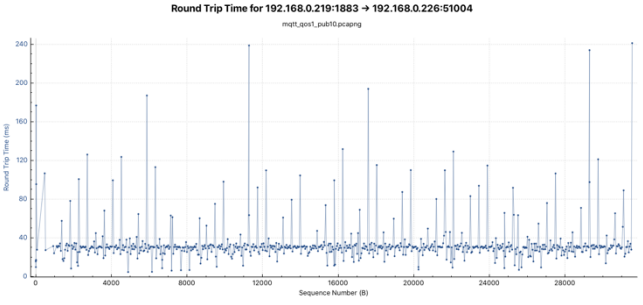

Figure 41: MQTT QoS 1, 10 clients – roundtrip time.

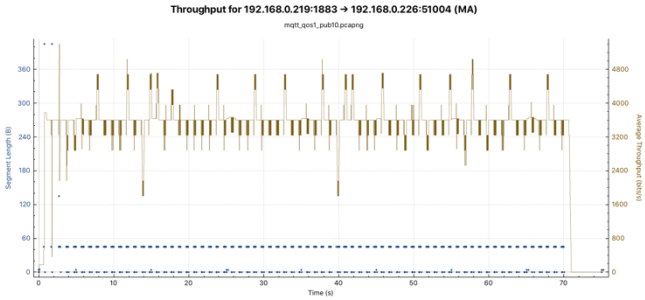

Figure 42: MQTT QoS 1, 10 clients – throughput.

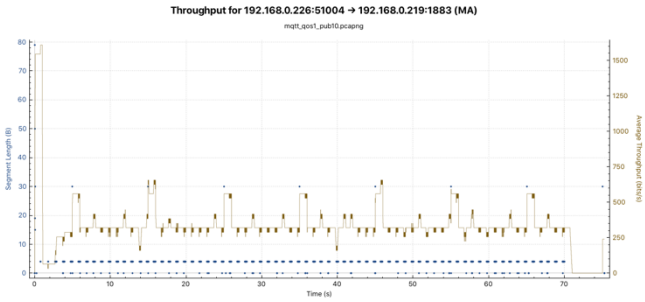

Figure 43: MQTT QoS 1, 10 clients – throughput.

Address A	Port A	Address B	Port B	Packets
192.168.0.219	48694	54.85.58.161	443	19
192.168.0.219	41616	54.69.165.95	443	3
192.168.0.219	59254	35.224.99.156	80	11
192.168.0.226	51003	192.168.0.219	1883	113
192.168.0.226	63565	192.168.0.219	1883	2
192.168.0.226	51004	192.168.0.219	1883	1.744
2a02:1811:c8c:be00:38af:2ff:5fb1:2f9	56404	2a00:1450:400e:808::200e	443	79
2a02:1811:c8c:be00:38af:2ff:5fb1:2f9	34898	2a00:1450:400e:807::2002	443	28

Figure 44: MQTT QoS 1, 10 clients – TCP conversations.

7.6 MQTT QoS 1, 100 clients

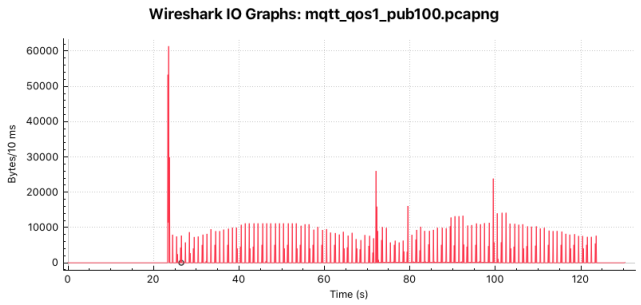

Figure 45: MQTT QoS 1, 100 client – overall traffic.

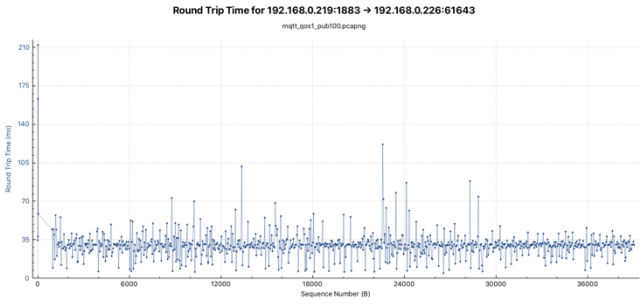

Figure 46: MQTT QoS 1, 100 clients – roundtrip time.

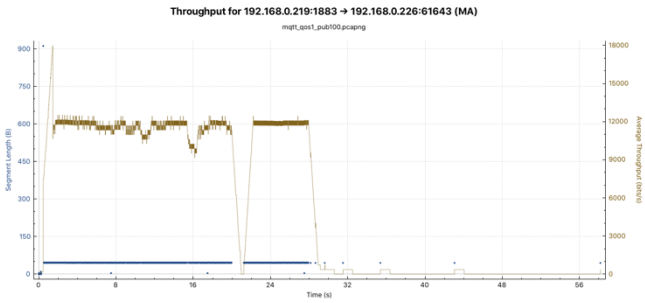

Figure 47: MQTT QoS 1, 100 clients – throughput.

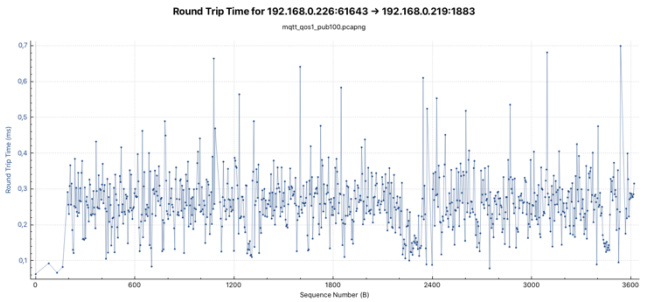

Figure 48: MQTT QoS 1, 100 clients – roundtrip time (ESP to broker).

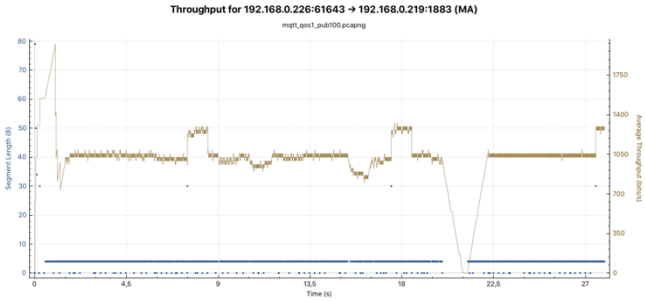

Figure 49: MQTT QoS 1, 100 clients – throughput (ESP to broker).

7.7 MQTT QoS 2, 1 client

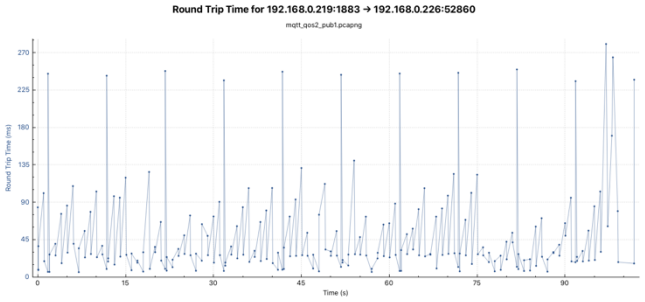

Figure 50: MQTT QoS 2, 1 client – roundtrip time..

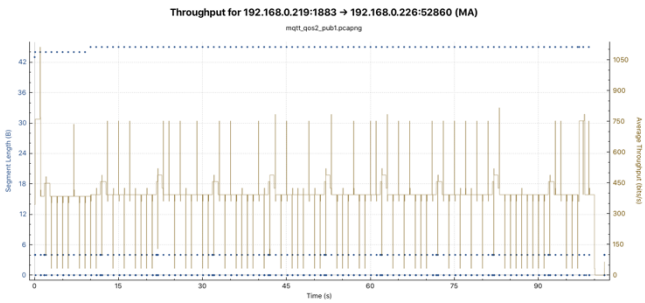

Figure 51: MQTT QoS 2, 1 client – throughput.

7.8 MQTT QoS 2, 10 clients

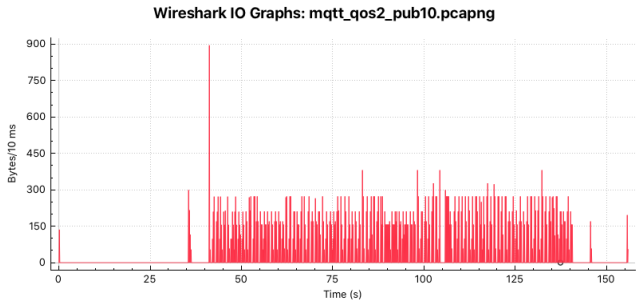

Figure 52: MQTT QoS 2, 10 clients – overall traffic.

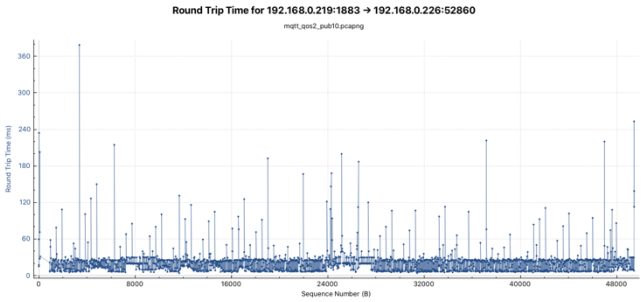

Figure 53: MQTT QoS 2, 10 clients – roundtrip time.

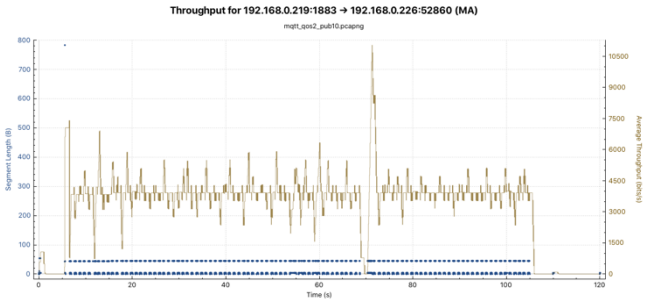

Figure 54: MQTT QoS 2, 10 clients – throughput.

7.9 MQTT QoS 2, 100 clients

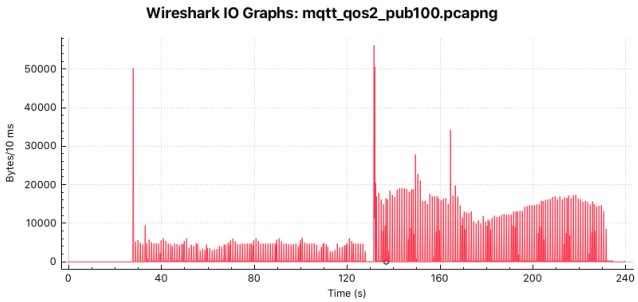

Figure 55: MQTT QoS 2, 100 client – overall traffic.

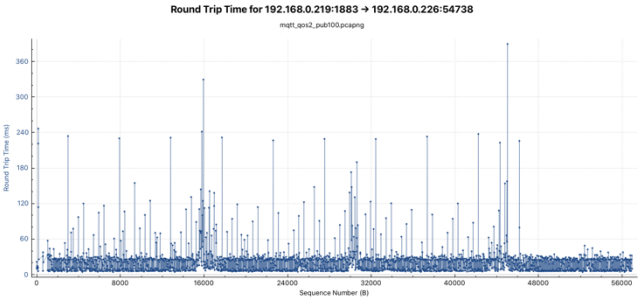

Figure 56: MQTT QoS 2, 100 client – roundtrip time.

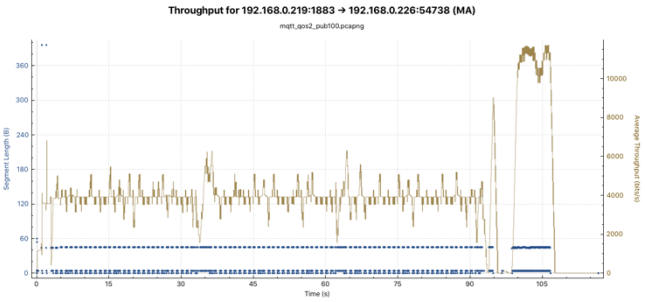

Figure 57: MQTT QoS 2, 100 client – throughput.

Figure 58: CoAP – failed request, no data returned by server.

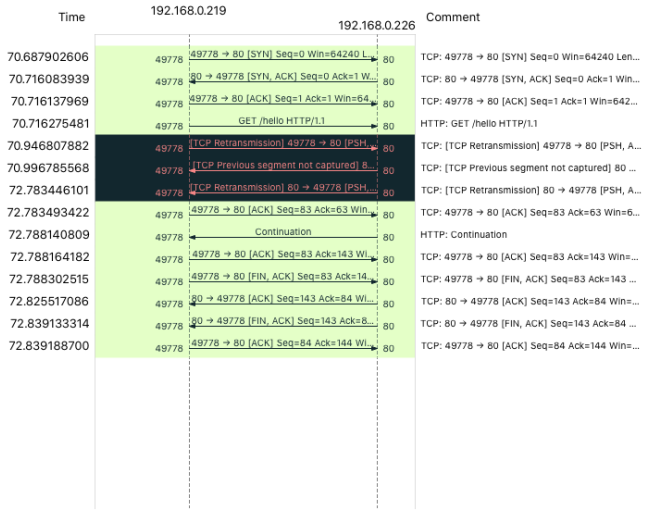

Figure 59: HTTP – failed request.

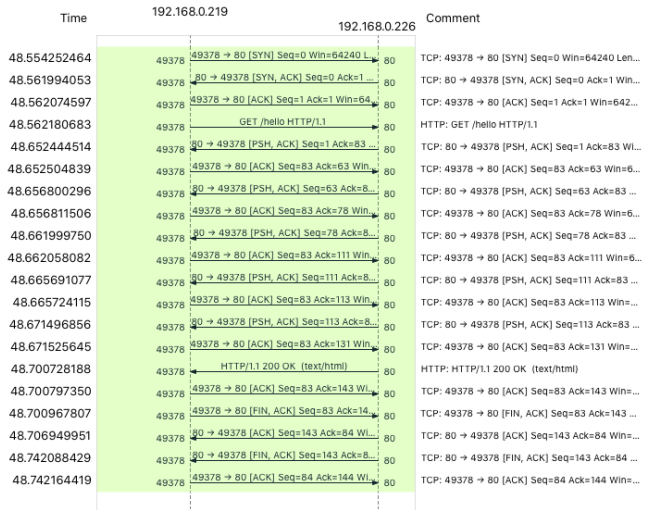

Figure 60: HTTP – successful request, a lot of ACK overhead.

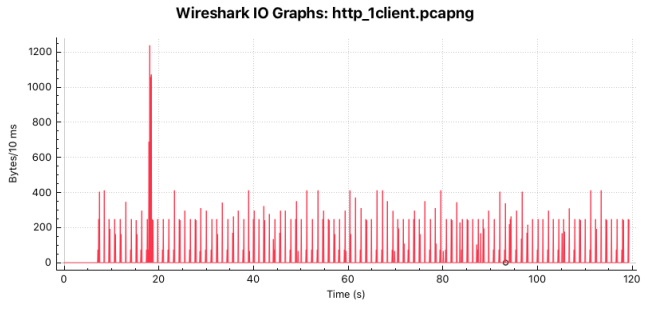

Figure 61: HTTP – overall traffic for 1 client w/ 100 requests.

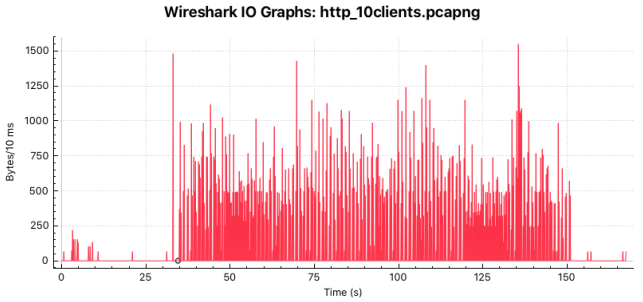

Figure 62: HTTP – overall traffic for 10 clients w/ 100 messages each.

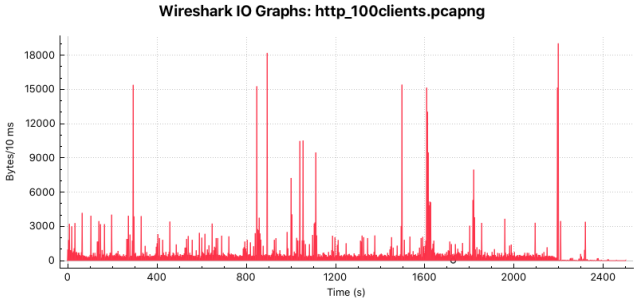

Figure 63: HTTP – overall traffic for 100 clients w/ 100 messages each.