UNIVERISTY OF PIRAEUS - DEPARTMENT OF INFORMATICS

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ – ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

MSc «Cybersecurity and Data Science»

ΠΜΣ «Κυβερνοασφάλεια και Επιστήμη Δεδομένων»

MSc Thesis

Μεταπτυχιακή Διατριβή

Thesis Title: Τίτλος Διατριβής:	**Theoretical and experimental security study of communication protocols in IoT environments (MQTT, CoAP, HTTP, XMPP)** Θεωρητική και πειραματική μελέτη ασφαλείας πρωτοκόλλων επικοινωνίας για περιβάλλον IoT (MQTT, CoAP, HTTP, XMPP)
Student's name-surname: Ονοματεπώνυμο φοιτητή:	**Maria Karamina** Μαρία Καραμηνά
Father's name: Πατρώνυμο:	**Naoum** Ναούμ
Student's ID No: Αριθμός Μητρώου:	ΜΠΚΕΔ21017
Supervisor: Επιβλέπων:	**Panayiotis Kotzanikolaou, Professor** Παναγιώτης Κοτζανικολάου, Καθηγητής

November 2024/ Νοέμβριος 2024

3-Member Examination Committee

Τριμελής Εξεταστική Επιτροπή

Constantinos Patsakis
Associate Professor

Πατσάκης Κωνσταντίνος
Αναπληρωτής Καθηγητής

Michael Psarakis
Associate Professor

Μιχαήλ Ψαράκης
Αναπληρωτής Καθηγητής

Panayiotis Kotzanikolaou
Professor

Παναγιώτης Κοτζανικολάου
Καθηγητής

Περίληψη

Το διαδίκτυο των πραγμάτων (Internet of Things – IoT) είναι ένα δίκτυο διασυνδεδεμένων συσκευών και αισθητήρων που συλλέγουν, επεξεργάζονται και μεταδίδουν πληροφορία μεταξύ τους και του περιβάλλοντος. Τα τελευταία χρόνια, στο πεδίο του IoT έχει υπάρξει μεγάλη εξέλιξη με όλο και περισσότερες χρήσεις σε πεδία όπως η γεωργία, η υγεία, η έξυπνη κατοικία και οι βιομηχανικές εφαρμογές. Κάθε ένα από αυτά τα πεδία έχει διαφορετικές ανάγκες και απαιτήσεις από την υποκείμενη τεχνολογία, που περιλαμβάνει διάφορα πρωτόκολλα στα διαφορετικά επίπεδα της στοίβας του IoT. Η παρούσα μελέτη εστιάζει στο επίπεδο εφαρμογής και σε επιλεγμένα πρωτόκολλα επικοινωνίας που μπορούν να χρησιμοποιηθούν σε μια υποδομή IoT. Συγκεκριμένα, τα MQTT, CoAP, HTTP, και XMPP μελετώνται σε επίπεδο λειτουργιών και αρχιτεκτονικής με μια σύγκριση σε επιχειρησιακό και τεχνικό πεδίο. Πραγματοποιείται επίσης μια πειραματική υλοποίηση, εξετάζοντας και συγκρίνοντας τα προαναφερθέντα πρωτόκολλα με βάση επιλεγμένα κριτήρια, εστιάζοντας στις δυνατότητες ασφαλείας τους και τα λειτουργικά τους χαρακτηριστικά.

Abstract

The Internet of Things (IoT) is a network of interconnected devices and sensors collecting, processing and transmitting information between them and the environment. In the latest years, the field of IoT has seen great advancements with more and more uses in fields such as agriculture, health-care, smart home and industrial applications. Each of these fields has different needs and requirements from the underlying technology, comprised of multiple protocols in the different layers of the IoT stack. This study focuses on the application layer and a selection of communication protocols that can be used in an IoT setup. Specifically, MQTT, CoAP, HTTP, and XMPP is studied on a functional and architectural level with a comparison on the operational and technical scope. An experimental implementation is also realized, examining and comparing the aforementioned protocols based on selected criteria, focusing on their security capabilities and functional characteristics.

TABLE OF CONTENTS

1 Introduction 7
1.1 Objectives 7
1.2 Related Work 7
2 Theoretical background 9
2.1 IoT architecture 9
2.2 Application layer protocols 10
3 Protocol Breakdown 12
3.1 MQTT 12
3.1.1 Functions and Features 12
3.1.2 Architecture 13
3.2 CoAP 15
3.2.1 Functions and Features 15
3.2.2 Architecture 17
3.3 HTTP 18
3.3.1 Functions and Features 18
3.3.2 Architecture 19
3.3.3 Differences Between HTTP/2 and HTTP/3 21
3.4 XMPP 22
3.4.1 Functions and Features 22
3.4.2 Architecture 23
3.5 Comparison 24
3.5.1 Criteria Definition 24
3.5.2 Operational Scope 24
3.5.3 Technical Scope 25
4 Experimental Implementation 27
4.1 Infrastructure Configuration for Testing 27
4.1.1 MQTT 27
4.1.2 CoAP 27

4.1.3 HTTP 27
4.2 Encryption and Authentication 28
4.2.1 MQTT 28
4.2.2 CoAP 29
4.2.3 HTTP 30
4.3 Connection Assessment 31
4.3.1 MQTT 31
4.3.2 CoAP 38
4.3.3 HTTP 43
4.3.4 Speed comparisons 49
4.3.5 Assessment with Security Configurations 51
4.4 Comparison 57
4.4.1 Security Scope 57
4.4.2 Functional Scope 58
5 Conclusions and Future Work 60
5.1 Conclusions 60
5.2 Future Work 61
6 Bibliography 62

1 Introduction

In the last few years, one of the fastest growing sectors in the field of information technology is the Internet of Things. The Internet of Things (IoT) is a global network of smart devices communicating with each other to exchange information, through private networks and the Internet. These devices are typically used to gather data or perform computations and often include sensors that allow the collection of information from the environment so it can then be transmitted to a central node or communicated to other devices for further processing. The Internet of Things has applications in many different fields, such as health-care, agriculture and monitoring, smart home and city, and commercial and industrial systems. [1]

Nowadays, the advancement of technological solutions has allowed a huge growth in the sector of IoT. Smart devices and sensors are used more and more in commercial environments but also in everyday life, constantly collecting and transmitting data through public and private networks. This development has produced an ever-growing need for secure protocols to be used in the communication between IoT devices but also for their communication with other devices of the Internet.

However, the IoT field is large, and applications in different sectors do not have the same needs for features while also having different requirements when it comes to technical limitations and security levels. This is reflected in the technical characteristics of the protocols used during the transfer of data between devices. As an example, a network of sensors designed to collect data about soil properties in an agricultural application may have very different requirements from the smart card authentication system of a large company building when it comes to features, energy consumption and security. At the same time, the specific devices used in each application could pose additional restrictions to the features supported by the system, through the use of limited memory, processing power and need for low power consumption.

As a result, to cover every need, multiple underlying technologies have been developed and proposed for the communication between devices in IoT networks. These provide different functionality to the devices and applications that use them and can result in different technical and functional characteristics like power and processing requirements, use under low-bandwidth conditions and different security features. As a result, when designing and developing an IoT solution, the selection of the most appropriate communication protocols for the occasion can prove particularly hard.

1.1 Objectives

Some of the most popular communication protocols used in applications of the IoT field are MQTT, CoAP, HTTP and XMPP. The goal of this thesis is to study the aforementioned protocols, understand and analyze their features, on a functional and architectural level, and make a comparison based on operational and technical criteria. We'll also explore an experimental client-server implementation using selected protocols, with the purpose of studying those protocols in practice, and comparing their security characteristics in the context of Confidentiality, Integrity and Availability. The ultimate objective is a presentation of the aforementioned protocols and a final comparison of the functions and security level they provide, aiming at simplifying the selection of the appropriate protocol depending on the domain and requirements of each application.

1.2 Related Work

There are multiple studies providing an overview of IoT communication protocols and performing comparisons. MQTT, CoAP and XMPP are usually part of those comparisons. Most of them focus on their general characteristics on a theoretical level, without an experimental comparison. There

are also studies performing practical evaluations on performance and energy consumption in low-bandwidth setups.

Bayilmis et al. [2] gave a theoretical overview of multiple protocols and executed performance tests on MQTT, CoAP and Websocket based on average delay time, throughput and energy consumption at different payload sizes, quality of service levels and methods (GET, POST, PUT). Al-Fuqaha et al. [3] provided an overview of the IoT stack and made comparisons between CoAP, MQTT, XMPP, AMQP, DDS and HTTP. Reddy et al. [4] compared the characteristics and use cases of MQTT, XMPP, AMQP, DDS, CoAP, MQTT-SN and analyzed their performance based on tests on latency, bandwidth and data loss. Karagiannis et al. [5] presented a detailed description of COAP, MQTT, XMPP, Restful Services, AMQP, Websocket and their state at the time of writing. Silva et al. [6] performed several tests with MQTT, CoAP and OPC-UA evaluating performance and packet loss.

Not many studies focus on the security scope of IoT applications. At the same time, the protocol standards keep evolving so older studies may not take into account changes in the latest versions of the protocols, especially MQTTv5 and HTTP/3. In this paper we will go more in-depth on the architecture of each protocol and give a larger focus on the security side of IoT, while also presenting the differences introduced in their latest versions. An experimental comparison will also provide better insights on the current state of these protocols, the performance of their latest versions, and tools that are available, focusing on the challenges an engineering team may face in their integration. The result is a more well-rounded study, making a comparison on both theoretical and practical level, with a major focus on the aspects of confidentiality, integrity and availability.

2 Theoretical background

2.1 IoT architecture

IoT is a combination of several technologies and protocols that work together across a universal network. It includes devices – nodes that interact with the environment, remote servers that manage incoming data, storage solutions and network infrastructure. IoT devices can be sensors or actuators. Sensors function as inputs, collecting information from their internal state or their environment and supplying it to the IoT network. There are no size or power limitations on the definition of sensor, a thermometer or a mobile phone can both function as sensors but usually small devices with limited resources are used. Actuators function as outputs, effecting a change to their environment based on IoT requests, for example turning on/off lights or adjusting air conditioner temperature. [7]

There is no single IoT architecture that is agreed upon by researchers. Various architectures have been proposed while specific IoT systems may differ in their implementation details. The most common architectures are the Three-, Four- and Five-Layer Architectures.

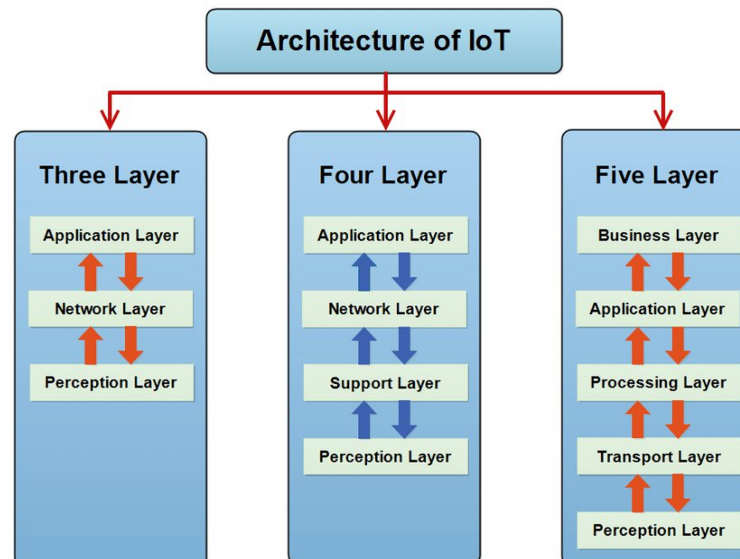

Figure 1. IoT architectures [8]

The Three-Layer Architecture is the simplest one, comprising of the Perception Layer, the Network Layer and the Application Layer.

The *Perception Layer* operates as the senses of the network. It detects physical parameters in the environment or other smart devices in the network. Devices used in this layer can include RFID tags, barcodes and various sensors for collecting information about the environment, such as location data, temperature and humidity or motion.

The *Network Layer* is responsible for transmitting and processing data in the IoT network and connecting smart devices. It works as the intermediary layer between the perception and application layer. The communication between devices can be either wired or wireless.

The *Application Layer* defines the applications where IoT has been deployed. It provides specific services to the user depending on the type of application (eg. smart home, smart health, animal tracking).

The Four-Layer Architecture adds a separate Support Layer responsible for storing and processing data coming from the perception layer. Data in IoT often requires high reliability where cloud computing can play a crucial role. [9]

In the Five-Layer Architecture three different layers are added.

The *Transport Layer* is similar to the Network Layer of the Three-Layer Architecture, transferring data between the perception and processing layer through networks.

The *Processing Layer* acts as a middleware. It processes and stores large amounts of data coming from the transport layer before passing it to the higher layers. Cloud computing and database solutions can be used when processing the data.

The *Business Layer* is at the top of the architecture with control of the whole IoT system. It manages applications and services, along with user privacy and other business logic of the specific application.

2.2 Application layer protocols

Various technologies and protocols are used in the different layers of the IoT architecture. The protocols that define application-specific behavior when processing and transmitting collected data can be found in the Application Layer in all proposed architectures. These control the format of the transmitted data, how it should be transported between devices and the role of different components in the network. As a result they can vary a lot depending on the requirements and capabilities of the specific system.

As a general rule, devices act as clients or servers in a network. Clients may communicate with servers or with each other through messages sent in a format predefined by the communication protocol. The communication can be stateful or stateless and provide some quality of service selected by the protocol or the user. Depending on the specifics of the connection process and payload formats, some protocols may result in more lightweight communication while others may include a larger overhead.

Some of the most popular protocols used in this layer are listed below and will be the subject of this paper.

- **MQTT** (MQ Telemetry Transport) is an OASIS standard messaging protocol designed as a lightweight transport for the Internet of Things. It uses a publish/subscribe architecture for communication between clients and a central MQTT broker. Built for efficiency, it's mostly used in environments with limited resources where reliability is important. It's currently used in a variety of industries, including automotive, manufacturing, smart home and transportation.
- **CoAP** (Constrained Application Protocol) is another lightweight transfer protocol for use in constrained networks in the Internet of Things. It uses a RESTful (Representational State Transfer) architecture with standard HTTP methods (GET, POST, PUT etc.) for communication between low-power devices. It's designed to use minimal resources with small messages in stateless communication over UDP, with optional reliability. Use cases include smart home automation, industrial applications, wearables and energy management.
- **HTTP** (Hypertext Transfer Protocol) is a communication protocol for the transfer of hypermedia documents, such as HTML. Its main architecture is client-server communication, where clients send requests (eg. GET, POST, DELETE) to a server and wait for a response in a stateless environment. It operates over a TCP connection (HTTP/2 or lower) or a UDP connection (HTTP/3). It is most commonly used for the communication between web browsers and web servers but can also be used in IoT applications.
- **XMPP** (Extensible Messaging and Presence Protocol) is a protocol designed for real-time communication, focused on applications around instant messaging, presence and

collaboration, based on the XML standard. It uses XML streams through which messages (XML stanzas) are sent for client–server or server–server communication. Its most typical use case is instant messaging but it has various uses in IoT as well, for smart home automation, remote control of devices and connectivity of microcontrollers.

3 Protocol Breakdown

3.1 MQTT

3.1.1 Functions and Features

The MQTT protocol relies on a publish/subscribe architecture, using topics to communicate between clients and a central MQTT broker. Clients can act as a publisher or a subscriber. Publishers push messages to specific topics of the MQTT broker and subscribers subscribe to topics which then receive those messages. MQTT uses TCP as the underlying protocol with default ports 1883 (mqtt) and 8883 (mqtts).

There are four versions of MQTT:

- **1.2:** the initial version
- **3.1:** adding most functionality used today
- **3.1.1:** became an OASIS standard
- **5:** latest version, adding more functionality

Most applications today use either version 3.1.1 or version 5.

As an IoT protocol MQTT includes some basic communication features and extra features related to its architecture.

Pub/Sub Communication: Due to its topic-based structure, MQTT allows only one-way communication between client and server, that is, a client that publishes to a topic can only send messages from that topic without receiving anything back. However, a client can function both as a subscriber and a publisher and a broker can specify a response topic for the publisher to subscribe to and receive relevant messages. Each client has a unique Client ID which identifies it to the broker.

Statefulness: MQTT is a stateful protocol, relying on a persistent TCP connection. Clients can also specify if their connection to the broker should be persistent, in which case the broker stores some information about the client so that it can continue on the same connection after reconnecting in case the initial connection was disrupted due to network issues. Clients that do not need a persistent session can set the clean session flag to disable this feature and start a new connection.

Reliability: MQTT supports three different levels of QoS (Quality of Service), defined by the clients.

- QoS 0 (at most once) offers no guarantee of message delivery. A sender sends its message once to the receiver without expecting an acknowledgment. It is the most efficient and can be used when the network is expected to be stable or in cases where some data loss is acceptable.
- QoS 1 (at least once) guarantees that the message will reach the receiver at least once. A PUBACK packet is used to confirm the receipt and, if not sent, the sender retransmits the message to ensure successful delivery. This method can produce duplicate messages, in case the PUBACK packet was lost, which are handled by the receiver.
- QoS 2 (exactly once) ensures that the message is delivered exactly once, without producing duplicates. A four-part handshake is required where, after receiving the initial PUBLISH packet, the receiver sends a PUBREC packet, the sender responds with a PUBREL packet and the receiver finishes with a PUBCOMP packet. In this way both parties can be sure that the message has been delivered. If something is lost in transmission, the sender is responsible for retransmitting the lost packet while the receiver stores the initial packet identifier to avoid processing the same message again.

Security: MQTT supports the use of TLS (mqtts) for encrypted communication. When it comes to authentication, clients can authenticate using a username and a password and most MQTT brokers support workflows for authentication and authorization based on username, password, client id and requested topic. In version 5, a special AUTH packet was added, supporting more advanced authentication methods such as OAuth. Brokers may also support x.509 certificates for client authentication.

Error Handling: Negative acknowledgments, or reason codes, can be used by the broker to notify the client of potential errors and unsuccessful operations. Version 5 introduced a much bigger number of reason codes and the ability to use them in more types of packets that previously lacked them (eg. PUBACK). It also added a bi-directional DISCONNECT packet, notifying the recipient of the reason of disconnection.

Retained Messages: In the general case, topics are not persisted and messages are lost if there are no subscribers to receive them. To combat this, publishers can specify a retained message per topic that will be stored and sent as a first message to any client that subscribes to that topic, even if the publisher isn't currently sending any messages.

Last Will and Testament (LWT): When working in unreliable networks, a client may disconnect ungracefully, without sending a DISCONNECT packet. To deal with this scenario, clients can include a last will message and topic in their initial CONNECT packet. If the server detects an abnormal disconnection, it broadcasts this message to all clients subscribed to that topic, letting them know of the disruption.

User Properties: Version 5 of MQTT added an extra field to its messages, called user properties, containing a collection of key – value pairs. These can be used by clients to provide extra metadata about the client or the message itself. Developers can use this information for application-specific purposes such as making decisions about the packet's routing and next steps.

3.1.2 Architecture

As mentioned, MQTT uses a publish/subscribe architecture. The components include a single server, called MQTT broker, and multiple clients, acting as publishers or subscribers. Clients connect to the MQTT broker and send/receive messages through topics. A topic is an entity with a unique name to which clients can subscribe or publish messages. When a client subscribes to a topic, the MQTT broker keeps track of this subscription. When a client publishes a message to a topic, the MQTT broker is responsible for forwarding that message to the clients subscribed to this topic. A topic can have subtopics specified by a forward slash, so a topic name is of the format *topic/subtopic/subsubtopic/...* with a variable number of subtopics. Wildcards can be used to subscribe to multiple topics in a single subscription. "#" matches any number of topic levels under a specified topic and "+" matches a single level under a specified topic. So a client subscribing to *topicname/#* will receive any messages published in *topicname* or any of its subtopics. Topics are temporary and only defined by the clients when publishing or subscribing to them. A client can act as both a publisher and subscriber and can publish and subscribe to messages of a single or multiple topics. Clients do not have addresses but are identified by the broker by their unique client ids.

An example of the general flow of messages can be seen in the following diagram.

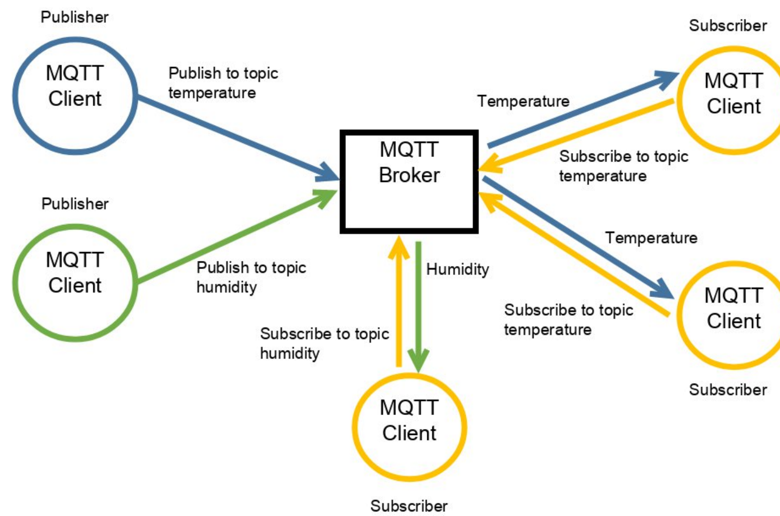

Figure 2. MQTT architecture

When a client wishes to publish or subscribe to a topic, the following procedure is executed. First, the client communicates with the MQTT broker and exchanges some initial connection information. Authentication information is also sent by the client if required. Once a connection is established, the client can send a subscription request or a publish packet with a payload to the broker. Depending on the Quality of Service level, one or more acknowledgments are then exchanged between server and client. An unsubscribe request by the client is also possible, with its own acknowledgment. The server can also send a publish packet to the client, when a message has been published to one of their subscribed topics. Finally, a disconnect packet is sent by either the client or the server when they wish to terminate the connection.

The communication between nodes is carried out via control packets. A control packet adheres to the following format.

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

Table 1. MQTT Control Packet format [18]

In the part of the fixed header, the control packet type is specified. There are 15 control packet types which are used for the entire flow of communication in MQTT.

Name	Value	Direction of flow	Description
Reserved	0	Forbidden	Reserved
CONNECT	1	Client to Server	Connection request
CONNACK	2	Server to Client	Connect acknowledgment
PUBLISH	3	Client to Server or Server to Client	Publish message
PUBACK	4	Client to Server or Server to Client	Publish acknowledgment (QoS 1)
PUBREC	5	Client to Server or Server to Client	Publish received (QoS 2 delivery part 1)
PUBREL	6	Client to Server or Server to Client	Publish release (QoS 2 delivery part 2)
PUBCOMP	7	Client to Server or Server to Client	Publish complete (QoS 2 delivery part 3)
SUBSCRIBE	8	Client to Server	Subscribe request
SUBACK	9	Server to Client	Subscribe acknowledgment
UNSUBSCRIBE	10	Client to Server	Unsubscribe request
UNSUBACK	11	Server to Client	Unsubscribe acknowledgment
PINGREQ	12	Client to Server	PING request
PINGRESP	13	Server to Client	PING response
DISCONNECT	14	Client to Server or Server to Client	Disconnect notification
AUTH	15	Client to Server or Server to Client	Authentication exchange

Table 2. MQTT Control Packet types [18]

The variable header and payload are optional and depend on the control packet type. One or more reason codes, indicating the result of an operation can be present in the variable header or the payload. The payload can also contain other values depending on the control packet type. The message in PUBLISH packets is sent in the payload field.

3.2 CoAP

3.2.1 Functions and Features

The CoAP protocol follows a client – server architecture, where a client requests an action on a resource and the server responds with a response code and a payload. The payloads are binary and can follow any format. Unlike other protocols, nodes can result in both client and server roles. Its underlying protocol is UDP and default ports are 5683 (coap) and 5684 (coaps). The main protocol is defined in RFC 7252 [20] and has been extended in separate specifications.

Its basic features include the following.

REST: CoAP uses a RESTful approach in its messages. The client makes a request to a resource specified by a URI, making use of one of the methods GET, POST, PUT or DELETE. These methods are designed to function similar to corresponding HTTP methods, so a GET request retrieves a resource while POST, PUT and DELETE affect resources on the server.

Statelessness: CoAP is stateless and due to the use of the UDP protocol all messages are asynchronous. In order to match requests to responses, a client-generated token is used in the request header which is then repeated in its corresponding response from the server.

Reliability: Messages can be transmitted in two modes: Confirmable and Non-Confirmable. Messages marked as Confirmable (CON) expect an acknowledgment message (ACK) from the receiver. If an acknowledgment is not received, the message is retransmitted using a default timeout and exponential back-off between retransmissions. The messages include a Message ID in the header which is matched by the acknowledgment message. Messages marked as Non-Confirmable (NON) do not provide any reliability mechanisms but still use a Message ID for duplicate detection. In both cases, if the server cannot process the request a reset (RST) response is sent.

Security: CoAP supports the use of DTLS over UDP (coaps). Four security modes are supported.

- NoSec: DTLS is disabled.
- PreSharedKey: a list of pre-shared keys is used which includes which nodes every key can be used to communicate with.
- RawPublicKey: the device has an asymmetric key pair without a certificate and a list of nodes it can communicate with.
- Certificate: the device has an asymmetric key pair with an X.509 certificate, signed by a common trust root, and a list of root trust anchors used for validating certificates.

Matching requests and responses share the same DTLS session. CoAP does not provide further authentication or authorization options on the protocol level.

Error Handling: CoAP provides error handling through response codes. These follow a similar format to HTTP response codes and can describe successful requests or client or server errors. Response codes follow the format of a 3-digit code and are separated into three categories: 2.xx, 4.xx and 5.xx.

Discovery: Clients can discover servers by learning the URI of one of its resources. Alternatively, multicast CoAP can be used for server discovery. The Constrained RESTful Environments (CoRE) Link Format (RFC 6690) [21] is used for referencing hosted resources and can be used for constructing a resource directory through which clients can discover resources on the network's servers.

Block-Wise Transfer: Block-wise transfer is an extension that allows big payloads to be fragmented on the protocol level, avoiding IP fragmentation which can be problematic in constrained networks. (RFC 7959) [22]

Resource Observing: Resource observing is an extension that allows CoAP to operate in a publish/subscribe mode. A client registers its interest to a resource with a GET request with the observe option enabled. The server then adds this client to a list of observers of this resource and, whenever the state of this resource changes, sends a notification to the client. The notification message is structured as an additional response to the original GET request made by the client. (RFC 7641) [23]

End-to-end protection: When proxies are used between endpoints, an extra protection against eavesdropping and modification of messages can be used, called Object Security for Constrained RESTful Environments (OSCORE). OSCORE provides encryption and integrity protection for any CoAP options not relevant to proxies as well as message contents. It ensures that only the sender and receiver nodes can read and modify the messages between them, regardless of any operations in intermediary nodes. (RFC 8613) [24]

3.2.2 Architecture

CoAP is designed to work with a network of devices acting as clients or servers. Its architecture usually involves multiple devices – servers that provide data based on sensors or other sources and one or more devices acting as clients that request and then process the data from the servers. Devices can also act as both servers and clients, forming a network of nodes that communicate sensor information between them.

The communication between clients and servers is done through RESTful requests to resources. A resource is identified by its URI which follows the format *"coap:" "//" host [":" port] path-abempty ["?" query]*. The coap or coaps protocol is used, followed by the host and optionally a port number if the default is not used, and an absolute path to the resource is specified, along with an optional query string. An example URI is *coap://example.com/temperature?unit=c.* When a request is sent to a resource by the client, a response with a response code and payload is returned by the server, after applying the relevant actions to the resource.

Another way of getting information about a resource is by using the resource observing extension, where a subscription is created to a resource specified by a URI. The server keeps track of those subscriptions and sends a response with information to the subscribed clients every time the state of this resource changes.

The following diagram shows the general flow of messages in CoAP.

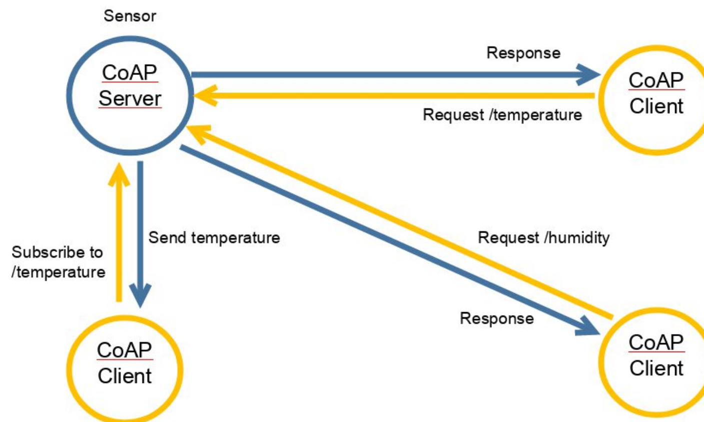

Figure 3. CoAP architecture

A client that wishes to make a request to an endpoint sends a message to the specific URI, using one of the available request methods. If security is desirable, a DTLS handshake takes place first and all messages are then sent as DTLS "application data". The message sent includes a client generated token that identifies that request and may be reliable (CON) or non-reliable (NON). In the case of a non-reliable message the server sends another non-reliable message with the response to the request. In the case of a reliable message, the server sends an acknowledgment to the client's message with the response embedded in the acknowledgment message or a separate acknowledgment and response message. In both cases the same client generated token is used in the response to identify the corresponding request.

All communication in CoAP is carried out via messages, with the same format for both requests and responses. A message consists of a 4-byte header, a token value, and optionally more CoAP options and a payload. The header field includes a version number, an integer specifying the type of message (NON, CON, ACK, or RST), the token length, a code identifying the request method or response code carried by the message and a unique message id.

The code part of the header follows the format c.dd where the first digit denotes the code class and the two digits after the dot denote the detail. For requests, the class part is a 0 while the details specify the request method. The code can take one of the following values.

Code	Name
0.01	GET
0.02	POST
0.03	PUT
0.04	DELETE

Table 3. CoAP Request Methods [20**]**

For responses, the code part of the header specifies the response code. Successful responses use a class digit of 2, client errors a class digit of 4 and server errors a class digit of 5. Response codes using a class digit of 3 are not defined but are reserved for future use. The full list of response codes can be found in [20].

Further optional options and a payload may follow the header depending on the type of message being sent.

3.3 HTTP

3.3.1 Functions and Features

The HTTP protocol uses a client – server architecture with a request – response model. Clients request resources or actions from servers through methods sent in requests and servers respond with appropriate response codes and payloads. The message payload is binary and can follow different formats. It is designed for use in web environments, between web servers and browsers. Its underlying protocol can be TCP or QUIC (in the case of HTTP/3) and its default ports are 80 (http) and 443 (https).

HTTP has been through the following versions over the years:

- **HTTP/0.9 and HTTP/1.0:** are older versions obsoleted by HTTP/1.1
- **HTTP/1.1:** defines the basic protocol and semantics
- **HTTP/2:** adds features improving efficiency and performance
- **HTTP/3:** transfers the same HTTP semantics over the new QUIC protocol

Versions HTTP/1.1, HTTP/2 and HTTP/3 are all in use today, with HTTP/3 slowly being adopted by more applications while already supported by most browsers [26].

The main features across versions are the following:

Request Methods: HTTP requests make use of methods to fetch, post or affect changes on resources of the server. The client makes a request to a resource identified by a URI using one of the available HTTP methods (eg. GET, POST, PUT). The server then applies that change to its state and returns the appropriate response and response code. These methods are standardized and the list of available methods can be expanded by separate specifications.

Response Codes: HTTP responses reply to HTTP requests with a response code and optionally a payload. These response codes are used to notify the client about the result of the

request or any possible errors that have occurred. Response codes follow the format of a 3-digit code and are separated into five categories: 1xx, 2xx, 3xx, 4xx, 5xx.

Statelessness: HTTP is a stateless protocol, meaning that each request is processed in isolation and there is no persistent connection between a server and a client. Any information required for identifying or authenticating a client should be included in the request payload or header.

Security: HTTP supports the use of TLS (https). Authentication can be implemented through fields in request and response headers, for example through Basic Authentication, JWT tokens or Cookies.

Efficiency: Since HTTP/2, the protocol uses binary frames in its messages, greatly increasing efficiency. TCP connections can be reused for multiple requests and headers are compressed through the use of HPACK (in HTTP/2) or QPACK (in HTTP/3). In HTTP/3, the QUIC protocol is used which implements multiple streams over UDP connections and handles retransmission of lost packets per stream, meaning that multiple streams are not blocked when sending packets reliably. This makes HTTP/3 much more efficient and friendlier to the network as retransmissions take up less bandwidth.

3.3.2 Architecture

HTTP follows an architecture where clients make requests to servers, which then send back responses. One side of the communication has the role of client and the other side has the role of server, while usually there are fewer servers than clients in a network.

The communication is done similar to CoAP, through requests to resources on the server, identified by a URI. The URI follows the format of *"http" "://" authority path-abempty ["?" query]* where *authority* defines the hostname and optionally a port, *path-abempty* the path to the resource and *query* extra parameters. When a client sends a request to the server, the appropriate actions are taken by the server and a response with a response code and optionally a payload is returned to the client.

The general flow of messages can be seen in the following diagram.

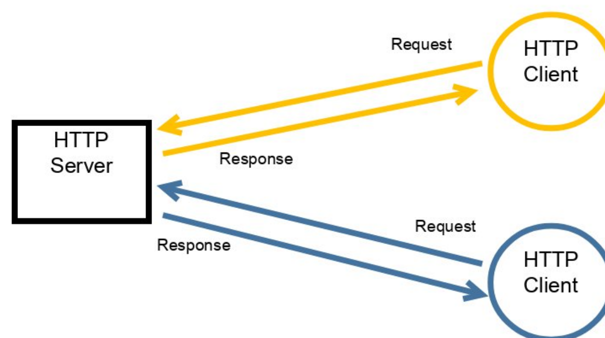

Figure 4. HTTP architecture

A client that wishes to make a request to the server, first identifies the HTTP version that will be used and can then send a message using one of the available request methods.

In the case of HTTP/2, a connection preface is sent by both sides, with a SETTINGS frame defining the settings of the connection and confirming the use of HTTP/2. If TLS is desired, the TLS negotiation must be completed before this step. Afterward, the client is free to start sending additional frames.

In the case of HTTP/3, the client first establishes a QUIC connection with the server, which includes a TLS handshake. After the QUIC connection is established, both sides need to send a SETTINGS frame to establish the settings of the HTTP connection before sending any other frames.

Since HTTP/2, requests are made in the form of frames in a stream. Multiple frames are sent through the same stream, structuring the client's request and the server's response. The request and response are sent in the same stream. A frame consists of a type, some flags and a stream identifier for specifying which stream the frame is part of. Frames of different types control the stream or are used for sending requests and responses. A request or response consists of a single HEADER frame followed by some CONTINUATION frames containing the header section, zero or more DATA frames containing the message content and optionally a HEADERS and some CONTINUATION frames for the trailer section. The message itself is sent in one or more DATA frames which contain the request method, options and payload, or the response code in the case of a response. Request and response data is the same as defined in HTTP/1.1.

The request methods defined by the main HTTP protocol are the following.

Method Name	Description
GET	Transfer a current representation of the target resource.
HEAD	Same as GET, but do not transfer the response content.
POST	Perform resource-specific processing on the request content.
PUT	Replace all current representations of the target resource with the request content.
DELETE	Remove all current representations of the target resource.
CONNECT	Establish a tunnel to the server identified by the target resource.
OPTIONS	Describe the communication options for the target resource.
TRACE	Perform a message loop-back test along the path to the target resource.

Table 4. HTTP Request Methods [27]

The response sent by the server includes a response code as a 3-digit number, belonging to one of five types of responses.

Code	Type	Description
1xx	Informational	Request received, continuing process.
2xx	Success	The action was successfully received, understood, and accepted.
3xx	Redirection	Further action must be taken in order to complete the request.
4xx	Client Error	The request contains bad syntax or cannot be fulfilled.
5xx	Server Error	The server failed to fulfill an apparently valid request.

Table 5. HTTP Response Types [27]

3.3.3 Differences Between HTTP/2 and HTTP/3

HTTP/3 made a transition from TCP to UDP, porting HTTP semantics defined in HTTP/1.1 and HTTP/2 to the QUIC transport protocol. The main reason for this transition is to bypass the overhead of TCP connections and provide a much more flexible protocol that is better suited for the needs of the modern web. Many of the features of HTTP/2 are delegated to QUIC while others are implemented on top of QUIC. The main differences between the two versions of the protocol are presented below.

QUIC streams: In HTTP/3 request – response pairs are defined through QUIC streams. Within each stream, HTTP/3 uses frames similar to HTTP/2 frames. Streams are independent of each other and concurrency and multiplexing are handled by QUIC, resulting in efficient request multiplexing since a single request does not block others when dealing with packet loss. Flow control is also provided per stream, allowing flow control on all types of frames, unlike HTTP/2 where only the DATA frame payload is subject to flow control. Both the client and server can initiate a QUIC stream. They can be unidirectional, with content flowing only from sender to receiver, or bidirectional, with data flowing in both directions. These changes greatly increase efficiency without sacrificing reliability.

Frames: HTTP/3 frames are mostly based on HTTP/2 frames. However, because many of the features implemented by HTTP/2 are handled directly by QUIC in HTTP/3, several frame types are not re-implemented in HTTP/3 while others have slightly different semantics. Some examples include the PRIORITY, PING, WINDOW_UPDATE and CONTINUATION frames which are missing from HTTP/3 and the DATA, HEADERS, PUSH_PROMISE and GOAWAY frames of HTTP/3 which do not contain some of the fields defined in HTTP/2.

In-order delivery: HTTP/2 streams guarantee in-order delivery of streams. Several of its features, such as field compression and prioritization, rely on this quality. In HTTP/3, QUIC guarantees in-order delivery within each stream, but not across different streams. As a result, HTTP/3 redefines the way some HTTP/2 features work so in-order delivery of frames is not assumed, while priority signaling is removed from frames.

Use of TLS: A secure connection can be established in HTTP/2 through the use of TLS over TCP. HTTP/2 allows the use of TLS 1.2 or higher while some TLS 1.2 cipher suites are prohibited due to inadequate security. The use of TLS is highly encouraged but not required by the HTTP/2 protocol. In contrast HTTP/3 incorporates TLS 1.3 on the transport layer through QUIC. That means

that TLS is required for any HTTP/3 connections, providing comparable integrity and confidentiality in all uses of the protocol.

3.4 XMPP

3.4.1 Functions and Features

XMPP is designed for an environment with multiple clients, communicating with each other through one or more federated servers. Similar to email, each client registers with a server and gets a unique address that allows it to send and receive messages with other clients that may be registered on the same or a different server. It is mainly used for instant messaging applications but can also be useful for the communication between multiple devices in a network. XMPP uses TCP as its underlying protocol on the default port 5222.

The XMPP protocol is designed to be extendable. The core features are defined in RFCs 6120 [30], 6121 [31] and 7622 [32] while further functionality is added through XMPP Extension Protocols (XEPs). Including proposed and rejected XEPs, there are 484 XEPs as of this writing. Each application is free to implement any of those extensions.

The following features are provided by the core protocol and most common XEPs.

Asynchronous Communication: In order to send and receive messages, a client needs to connect to an XMPP server. The client opens a long-lived TCP connection to the server and negotiates an XML stream. The same process happens from the server side resulting in two XML streams for each client – server connection. Those streams are in the format of an incrementally built up XML document and each message contains an XML snippet (eg. <message/>) that gets added to it. Whenever one of the sides needs to send a message, it can push a snippet through the already open connection, resulting in asynchronous communication between the entities.

JabberID: Each node in the network is identified by a unique id called JabberID or JID. This is a string following the format of an email address (eg. name@example.com). The first part of the JID is a string selected by the user and the second part is the domain name of the XMPP server on which they are registered. The JID is used as an address when a client wishes to communicate with another, similar to email. If a user is connected to their server from multiple devices/clients, each device requests a resource that is appended to the JID and identifies that particular device (eg. name@example.com/phone). This address is known as a Full JID.

Security: XMPP provides support for message encryption and client authentication. The core protocol supports the use of TLS for encrypting communication and uses SASL (Simple Authentication and Security Layer) for authentication purposes through username and password. Different SASL mechanisms, such as PLAIN and DIGEST-MD5 are supported. Of them, the EXTERNAL mechanism allows for authentication through a certificate and can be used for communication between servers. An ANONYMOUS mechanism is also available when there is no need for authentication of users. Further security features can be added through extensions, such as end-to-end encryption defined by XEP-0384 [33].

Presence: XMPP allows clients to broadcast their availability status to a group of other clients on the network. Through the implementation of presence and rich presence, a client can let others know if it is currently online along with further details about its status. This information is generally shared with the users that the client has selected to add to its contact list (roster). At the same time the client also subscribes to the presence information of the other clients in the contact list.

Publish/Subscribe: A publish/subscribe mechanism is also possible in XMPP through the use of XEP-0060 [34]. A client can subscribe to a resource (node) of another client. When that client publishes something, specifying that node, a notification is sent to all clients that have subscribed to this node.

3.4.2 Architecture

XMPP relies on an architecture with a big number of clients and smaller number of servers. Each client communicates with a specific server after, optionally, registering and authenticating on it and servers are federated with each other in order to receive messages and forward them to their connected clients. Each client has a unique address in the form of a JabberID, following the format of name@host where the host is the server they receive messages from. Clients communicate with each other using those JIDs as addresses and relying on their servers to forward their messages to the recipient's server which will then pass it on to the target client.

The general flow of messages is the following.

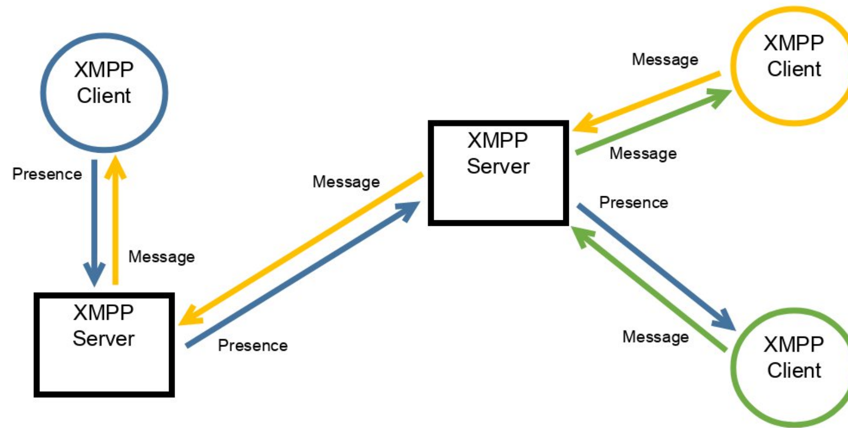

Figure 5. XMPP architecture

A client that wishes to come online first opens a connection to their own server by opening a stream. This step includes a stream negotiation where the server imposes certain conditions on the client such as authentication via SASL or encryption through TLS and available stream features are presented. After the connection has been established the client and server can keep sending messages to the open stream until one of them closes it to disconnect. The communication between servers is similar but usually imposing different conditions such as using different authentication mechanisms.

The open stream between two endpoints is in the format of an XML document that is incrementally built up by the messages sent by the two sides. Every message is in the form of an XML stanza. XML stanzas are in the format of an XML element with a number of attributes and, optionally, a body. The core protocol defines three types of stanzas: <message/>, <presence/>, and <iq/> while extensions can define the specifics of elements used within them.

The <message/> stanza provides a way to push information and is the main way of sending a message to another entity. The <presence/> stanza functions as a broadcast mechanism that sends information to any entities that have subscribed to the entity broadcasting. The <iq/> or info/query stanza provides a way of making a request and receiving a response similar to other request-response protocols.

Some attributes are sent with each stanza. The five common attributes defined by the main protocol are the following:

- **to:** identifying the recipient of a message by their JID.
- **from:** identifying the sender of a message by their JID.
- **id:** set by the originating entity for tracking responses or error messages related to a stanza.
- **type:** specifying the purpose or context, depending on the type of stanza.
- **xml:lang:** specifying the language of the stanza if it contains data that will be presented in a human-readable format.

In the case of an error, the receiving entity sends back the same kind of stanza using *type="error"* with an <error/> child element specifying the error that occurred.

In an IoT environment <iq/> stanzas can be mapped to requests – responses while <presence/> stanzas can be used as a publish/subscribe mechanism.

3.5 Comparison

3.5.1 Criteria Definition

We will proceed to compare the above protocols based on some criteria that are important when building an application. These are divided in the operational scope, ie related to the business side of the application, and the technical scope, related to technical details of the protocol and the application that uses it.

On the operational side the examined criteria are the *scope of application, extensibility,* and *adoption.* The *scope of application* defines the usual and intended uses of the protocol and for which type of application it will be best suited. The *extensibility* of the protocol describes any existing extensions or the ease with which it can be extended so it can be applied to further use cases. Finally, the *adoption* it has globally, including current popularity, support and integrations shows how easily it can be set up and combined with other tools and services.

On the technical side, the selected criteria will be *performance, reliability, efficiency,* and *security. Performance* affects the speed by which messages are transmitted. *Reliability* includes the availability of options for reliable transmission of messages, so no messages are lost if that is required by the application. *Efficiency* includes the consumption of memory, CPU and power as well as the protocol's behavior in constrained IoT environments. Finally, *security*, includes any encryption options and algorithms provided by the protocol as well as possible built-in mechanisms for authentication and authorization.

3.5.2 Operational Scope

Looking at the *scope of application* of the examined protocols it's clear that MQTT and CoAP are more geared towards IoT use cases as that is their primary design with a bigger focus on low-power devices as clients. MQTT is better suited for an environment where multiple devices-sensors need to communicate with a single server while CoAP may be better when dealing with many small devices with different roles. HTTP, on the other hand, is designed for basic client-server applications without a focus on IoT or constrained devices and XMPP is geared towards messaging applications where it is assumed that clients belong to different end users.

Examining *extensibility*, MQTT doesn't appear to provide any options. CoAP has some defined extensions beside the base protocol which provide significant help in IoT environments. HTTP includes many features and is generic enough that applications can build their own mechanisms on top of it, which may, however, require more engineering effort. Finally XMPP is designed with extensibility in mind, with many extensions already developed and covering different use cases and the option for new applications to roll out their own if not covered by the existing.

Finally, when looking at global *adoption*, MQTT seems widely supported and used in multiple technologies, with a good selection of software and libraries for it. A small limiting factor could be that not all tools support version 5, but it is steadily getting more adoption. CoAP has wide support through multiple open source libraries both for constrained devices and larger servers. HTTP/1.1 and HTTP/2 are widely adopted and supported by most programming languages and tools. HTTP/3 however may not have as much support as adoption is slower. XMPP has wide support and adoption by different applications but most of them have a focus on instant messaging, and not all tools implement the same extensions so some research is important when selecting tools for working with it.

	MQTT	**CoAP**	**HTTP**	**XMPP**
Scope of Application	IoT	IoT	Generic Client-server	Instant messaging
Extensibility	Only base protocol	Some defined extensions	Extra functionality can be built on top of the protocol	Many extensions Option to roll out own
Adoption	Wide adoption and support Smaller support for version 5	Wide support through libraries	Wide adoption and support for HTTP/2 Small support for HTTP/3	Wide adoption in instant messaging Support varies for different extensions

Table 6. Protocol comparison on operational scope

3.5.3 Technical Scope

On the *performance* side, according to their specifications and previous studies, MQTT and CoAP are both pretty performant, even in environments with significant constraints, with messages flowing fast between entities. CoAP should generally be more performant due to the use of UDP. In both cases performance varies based on the quality of service selected which affects the number of requests [2]. HTTP is not designed to be performant on constrained devices, however, HTTP/3 has made significant improvements over HTTP/2. XMPP is also not designed for constrained devices but its asynchronous nature or the use of relevant extensions can make it relatively performant.

For *reliability* MQTT provides different options of Quality of Service (QoS) that allow the user to specify the level of reliability that they need with guarantees of message transmission at least or exactly once. CoAP also allows the user to select whether communication will be reliable or not, through the use of server acknowledgments and duplicate detection. HTTP relies on the reliability of the underlying protocol, TCP in the case of HTTP/2 and QUIC in the case of HTTP/3, with QUIC taking care of retransmitting lost packets and detecting duplicates on the stream level. The core of XMPP doesn't provide any mechanisms of reliability itself but that can be added through extensions. Specifically XEP-0198 [35] adds a stream management mechanism with acknowledgments to XMPP.

For *efficiency*, MQTT and CoAP are both built to support constrained devices that may be low in CPU and memory resources but are also designed to work under low bandwidth environments. However, in the case of MQTT, a broker with higher consumption needs is required while in CoAP, smaller IoT devices can operate without a central node. Consumption of network bandwidth may also be greater in the case of MQTT due to the use of the TCP protocol and its overhead. HTTP/2 has improved on the efficiency of HTTP/1.1 with the use of frames and streams but is still fairly heavy on resource and network usage due to the larger packages and underlying TCP connection.

HTTP/3 improves on it with the use QUIC and its own, more efficient, streams. However, the features of HTTP remain large and resource-heavy for IoT environments [10]. XMPP has a larger overhead due to the use of XML in XML stanzas. However this has been addressed by extensions such as XEP-0322: Efficient XML Interchange (EXI) Format [36].

Examining *security* options, all protocols provide encryption, through either TLS or DTLS but with differences in the implementation as well as in other security features. MQTT provides optional TLS encryption along with built-in methods for authentication and authorization. The protocol itself allows for different options but it's important to note differences between broker implementations as not all methods are always supported. CoAP only provides optional encryption through DTLS without any built-in mechanisms for authentication and authorization but also supports end-to-end protection through OSCORE which can be combined with DTLS. HTTP provides encryption through TLS. In the case of HTTP/2 it is optional and limited only to the newest algorithms while in the case of HTTP/3 it is required and only allows use of the latest TLS 1.3. Both HTTP/2 and HTTP/3 provide the authentication mechanisms of the HTTP specification, which make use of header fields such as the Authorization field for use with Basic Authentication and JWTs as well as Cookies. XMPP also uses optional TLS and provides multiple ways of authenticating. As it is designed for use with user accounts, it makes use of SASL for authenticating clients to servers with several different options, while servers are usually authenticated to each other using certificates. Extensions can also add more security options such as end-to-end encryption for messages sent between clients.

	MQTT	**CoAP**	**HTTP**	**XMPP**
Performance	Performant even in constrained environments Depends on quality of service	Performant even in constrained environments Depends on quality of service	Not designed for constrained environments HTTP/3 has improved performance	Not designed for constrained environments Better performance can be achieved through extensions
Reliability	3 quality of service levels	Option for reliable messages	Underlying TCP and QUIC reliability	Through extensions
Efficiency	Clients designed for constrained devices Broker may require more resources	Both clients and servers can run on constrained devices	Resource-heavy	Heavy but can be improved through extensions
Security	Optional TLS Multiple authentication and authorization mechanisms	Optional DTLS Optional end-to-end protection through OSCORE No extra authentication and authorization mechanisms	Optional TLS for HTTP/2, required TLS 1.3 for HTTP/3 Multiple authentication and authorization mechanisms	Optional TLS SASL for client authentication or certificates for server authentication End-to-end encryption through extension

Table 7. Protocol comparison on technical scope

4 Experimental Implementation

4.1 Infrastructure Configuration for Testing

To examine the above characteristics in practice, an experimental implementation has been made featuring client – server connections for MQTT, CoAP, HTTP/2 and HTTP/3. For this purpose, a testing environment has been set up where different scripts were implemented, acting as the clients and servers for each protocol. For those scripts, Python 3.12 was used along with a collection of open-source libraries providing the base functionality of each protocol. The implementation was tested on machines running Linux where, for the simplest cases, a script simulating the server and a script simulating the client were running on the same machine and communicating with each other. For other cases, one or more scripts ran on two separate machines, connected through a WiFi network, to simulate a more realistic environment.

A set of tests was performed for each protocol. As a starting point, the simple case of a single request was tested, using a publish/subscribe architecture where possible, where a client requests and/or subscribes to a resource on a server that returns the current time as a string. After the simple case, further tests were performed, exploring the available features of the protocols and measuring the transfer speed of connections under different scenarios.

4.1.1 MQTT

For MQTT a pub/sub architecture was used, using a broker and two clients – one acting as a publisher and one as a subscriber. The two clients connect to the MQTT broker using MQTTv5. The publisher posts a payload with an ISO timestamp to a specific topic (metrics/time) every 10 seconds. The client subscribes to this same topic and receives and prints the messages posted by the publisher. The two clients and the broker are different entities on the same network and the clients connect to the broker on port 1883 using its IP address.

The libraries used for the above implementation are:

- Paho MQTT for the implementation of publisher and subscriber
- Mosquitto Broker as an out-of-the box configurable broker

4.1.2 CoAP

For CoAP an intermediate server is not required, so the end providing the resource works as a CoAP server with different clients requesting the resource. Both a request – response and a pub/sub architecture were tested, with a script working as a CoAP server and two clients, one requesting the current time with a GET request to the URI coap://coap-server/time, and another subscribing to changes of the resource at the same URI using an observation. The server provides a route that answers to normal GET requests with an ISO timestamp of the current time and also pushes updates to observations on the same route at random intervals. The clients connect to the server on port 5683 using its IP address.

For the above implementation the aiocoap library was used.

4.1.3 HTTP

For HTTP a server – client architecture was used with a server running and listening to a /time route, and two clients, one working as a publisher and making a POST request to that route with the time payload every 10 seconds and another making a GET request to the server to get the latest posted time. The server supports both HTTP/2 and HTTP/3 and separate clients were created for the two HTTP versions. The clients connect to the server on port 8000 using its IP address.

The libraries used for the above implementation are:

- httpx for the main client HTTP functionality
- aioquic for HTTP/3 support on the clients
- Fastapi for the server implementation
- Hypercorn as a web server configured with HTTP/3 support

For HTTP/3, the client examples of aioquic were used as a baseline.

4.2 Encryption and Authentication

There are multiple attacks that can be applied to IoT applications and their networks, many of which are related to a lack of encryption and proper authentication and authorization [11]. The above implementations are simple examples that create connections without taking into account other nodes in the network. As such, any actors in the network are free to read their packets and possibly modify them. At the same time, within the scope of the application, all clients have the same rights to send and receive packages without authenticating with the server. These points can compromise the integrity and confidentiality of the application. To combat this, the connections should be encrypted and some authentication mechanism applied where the protocol supports it.

4.2.1 MQTT

MQTT supports encryption through TLS which can be enabled on the broker by specifying a certificate and key file. By default the mosquitto broker selects TLS 1.3 if it's supported by the host machine and the clients, with default ciphers TLS_AES_256_GCM_SHA384, TLS_CHACHA20_POLY1305_SHA256 and TLS_AES_128_GCM_SHA256. This is configurable through its configuration file. A dhparamfile can also be provided for the DH exchange parameters in the ciphers that utilize them. Pre-shared-key based TLS support is also provided.

To test the encrypted behavior, a Certificate Authority (CA) was created and an X.509 certificate for the server was created and signed, along with a 4096-bit RSA key. The broker's listener was also changed to the mqtts port 8883. The default configuration of TLS 1.3 was used. On the client side, the new port was also specified and the CA's certificate added to the trusted CAs list. MQTT traffic is now encrypted and using the mqtts protocol with encrypted TLS packets appearing now on the network.

5	0.0001195…	127.0.0.1	127.0.0.1	TCP	66	59417 → 1883 [ACK] Seq=1 Ack=1 Win=33280 Len=0 TSval=3761165398 TSecr=3761165398
6	0.0001880…	127.0.0.1	127.0.0.1	MQTT	98	Connect Command
7	0.0001993…	127.0.0.1	127.0.0.1	TCP	66	1883 → 59417 [ACK] Seq=1 Ack=33 Win=33280 Len=0 TSval=3761165398 TSecr=3761165398
8	0.0002376…	127.0.0.1	127.0.0.1	MQTT	77	Connect Ack
9	0.0002514…	127.0.0.1	127.0.0.1	TCP	66	59417 → 1883 [ACK] Seq=33 Ack=12 Win=33280 Len=0 TSval=3761165398 TSecr=3761165398
10	0.0005800…	127.0.0.1	127.0.0.1	MQTT	86	Subscribe Request (id=1) [metrics/time]
11	0.0006459…	127.0.0.1	127.0.0.1	MQTT	72	Subscribe Ack (id=1)
12	0.0407585…	127.0.0.1	127.0.0.1	TCP	66	59417 → 1883 [ACK] Seq=53 Ack=18 Win=33280 Len=0 TSval=3761165439 TSecr=3761165398

Figure 6. MQTT packets

3	0.0000406…	127.0.0.1	127.0.0.1	TCP	66	35453 → 8883 [ACK] Seq=1 Ack=1 Win=33280 Len=0 TSval=3761284631 TSecr=3761284631
4	0.0002519…	127.0.0.1	127.0.0.1	TLSv1.3	583	Client Hello
5	0.0002637…	127.0.0.1	127.0.0.1	TCP	66	8883 → 35453 [ACK] Seq=1 Ack=518 Win=33024 Len=0 TSval=3761284631 TSecr=3761284631
6	0.0050260…	127.0.0.1	127.0.0.1	TLSv1.3	2183	Server Hello, Change Cipher Spec, Application Data, Application Data, Application Data, Application Data
7	0.0050362…	127.0.0.1	127.0.0.1	TCP	66	35453 → 8883 [ACK] Seq=518 Ack=2118 Win=33280 Len=0 TSval=3761284636 TSecr=3761284636
8	0.0056055…	127.0.0.1	127.0.0.1	TLSv1.3	146	Change Cipher Spec, Application Data
9	0.0056840…	127.0.0.1	127.0.0.1	TLSv1.3	337	Application Data
10	0.0057012…	127.0.0.1	127.0.0.1	TLSv1.3	120	Application Data
11	0.0057174…	127.0.0.1	127.0.0.1	TLSv1.3	337	Application Data
12	0.0457756…	127.0.0.1	127.0.0.1	TCP	66	35453 → 8883 [ACK] Seq=652 Ack=2660 Win=33280 Len=0 TSval=3761284677 TSecr=3761284636
13	0.0458024…	127.0.0.1	127.0.0.1	TLSv1.3	99	Application Data

Figure 7. MQTTS packets

For authorization purposes, the mosquitto broker supports the use of an access control list (ACL), specifying which clients have read or write access to which topics, by username.

user publisher

topic write metrics/time

user subscriber
topic read metrics/time

To identify the clients, two authentication mechanisms are supported, username – password authentication and certificate authentication, implemented on the broker side. For the password authentication, an encrypted password file is created, which is checked by the broker every time a client connects, using MQTT's username and password fields. For the certificate authentication, a TLS certificate – key pair is created for each client and signed by a CA which is added to the broker's trusted CA list. A certificate revocation list can also be provided. With a setting on the broker, the CN value of the certificate is used as a username to identify the client for authorization purposes. It is also possible to block the connection of any client who does not provide an identity through a certificate. Certificate client authentication and ACL were used in conjunction in the experiments.

```
mqttc.tls_set(ca_certs="../CA/ca.crt", certfile="keys/subscriber.crt", keyfile="keys/subscriber.key")

mqttc.connect("mqtt-broker", 8883)
```

Figure 8. MQTT secure connection to the broker

4.2.2 CoAP

CoAP provides two mechanisms for the assurance of integrity and confidentiality. Through OSCORE, end-to-end protection can be applied between nodes when intermediary proxies are used, so that message contents as well as any internal options are not exposed to the proxy. At the same time, DTLS can be used on the connection for encrypting the messages on the network. Use of DTLS and OSCORE can be combined when use of proxies is required. A built-in authorization mechanism for does not exist in CoAP but can be implemented on a higher level in the application.

In the aiocoap library, DTLS can be enabled on the server through the AIOCOAP_DTLSSERVER_ENABLED environment variable. Current support is limited to the use of pre-shared keys (PSKs) between the server and the client and different clients can be configured with different keys and ids. The PSKs can be loaded from json files and added to the configuration of both servers and clients and clients can now communicate with servers through the coaps protocol for an encrypted connection. By default DTLS 1.2 is used and the PSK is defined by the user in ascii format.

1	0.0000000…	::1	::1	CoAP	84	CON, MID:40321, GET, TKN:c1 76, coap://localhost/time
2	0.0012844…	::1	::1	CoAP	96	ACK, MID:40321, 2.05 Content, TKN:c1 76, coap://localhost/time
3	2.0041778…	::1	::1	CoAP	97	CON, MID:17518, 2.05 Content, TKN:c1 76, coap://localhost/time
4	2.0047299…	::1	::1	CoAP	66	ACK, MID:17518, Empty Message
5	3.0060000…	::1	::1	CoAP	97	CON, MID:17519, 2.05 Content, TKN:c1 76, coap://localhost/time
6	3.0065605…	::1	::1	CoAP	66	ACK, MID:17519, Empty Message
7	7.0098885…	::1	::1	CoAP	97	CON, MID:17520, 2.05 Content, TKN:c1 76, coap://localhost/time
8	7.0103322…	::1	::1	CoAP	66	ACK, MID:17520, Empty Message

Figure 9. COAP packets

1	0.0000000…	127.0.0.1	127.0.0.1	DTLSv1.2	109	Client Hello
2	0.0007211…	127.0.0.1	127.0.0.1	DTLSv1.2	86	Hello Verify Request
3	0.0010388…	127.0.0.1	127.0.0.1	DTLSv1.2	125	Client Hello
4	0.0013417…	127.0.0.1	127.0.0.1	DTLSv1.2	105	Server Hello
5	0.0013989…	127.0.0.1	127.0.0.1	DTLSv1.2	67	Server Hello Done
6	0.0017463…	127.0.0.1	127.0.0.1	DTLSv1.2	75	Client Key Exchange
7	0.0019006…	127.0.0.1	127.0.0.1	DTLSv1.2	56	Change Cipher Spec
8	0.0019597…	127.0.0.1	127.0.0.1	DTLSv1.2	95	Encrypted Handshake Message
9	0.0024363…	127.0.0.1	127.0.0.1	DTLSv1.2	56	Change Cipher Spec
10	0.0025085…	127.0.0.1	127.0.0.1	DTLSv1.2	95	Encrypted Handshake Message
11	0.0029513…	127.0.0.1	127.0.0.1	DTLSv1.2	83	Application Data
12	0.0051345…	127.0.0.1	127.0.0.1	DTLSv1.2	105	Application Data

Figure 10. COAPS packets

OSCORE can be similarly enabled using the module OscoreSiteWrapper and setting the client id and a secret on both client and server. Each client can have its own id and use a different secret

string for the encryption. The packets now appear as OSCORE packets and only expose routing-related packets and options required for proxying. For the time being, use of OSCORE and DTLS at the same time is not possible through the aiocoap library.

```
 1 0.0000000… 127.0.0.1    127.0.0.1  OSCORE   69 CON, MID:20007, FETCH, TKN:fe 75, /time
 2 0.0109200… 127.0.0.1    127.0.0.1  OSCORE   90 ACK, MID:20007, 2.05 Content, TKN:fe 75
 3 4.0221219… 127.0.0.1    127.0.0.1  OSCORE   93 CON, MID:43046, 2.05 Content, TKN:fe 75
 4 4.0229410… 127.0.0.1    127.0.0.1  CoAP     46 ACK, MID:43046, Empty Message
 9 9.0232716… 127.0.0.1    127.0.0.1  OSCORE   93 CON, MID:43047, 2.05 Content, TKN:fe 75
10 9.0239642… 127.0.0.1    127.0.0.1  CoAP     46 ACK, MID:43047, Empty Message
11 18.032292… 127.0.0.1    127.0.0.1  OSCORE   93 CON, MID:43048, 2.05 Content, TKN:fe 75
12 18.032952… 127.0.0.1    127.0.0.1  CoAP     46 ACK, MID:43048, Empty Message
13 23.037962… 127.0.0.1    127.0.0.1  OSCORE   93 CON, MID:43049, 2.05 Content, TKN:fe 75
```

Figure 11. COAP with OSCORE packets

An authorization mechanism was not used in the experiments as one is not provided by the protocol but it could be implemented by each application using the client identity set in the DTLS configuration as separate PSKs can be used for each client.

```
server = await aiocoap.Context.create_server_context(bind=("coap-server", 5683), site=root)
server.server_credentials.load_from_dict({
    ":client1": { "dtls": client1_credentials },
    ":client2": { "dtls": client2_credentials }
})
```

Figure 12. COAP server DTLS configuration

4.2.3 HTTP

HTTP provides an encrypted connection through https, using TLS. Hypercorn enables https when providing a certificate and key for the server. Those were created and signed with a local CA, as before, which was then added to the clients' trusted CA list. The clients also use the https protocol when contacting the server. For HTTP/3, the use of TLS is required. The same certificate and key are needed and the QUIC protocol is bound to a port with the option of —quic-bind on the server.

For HTTP/2 TLS 1.3 is selected by default if the clients support it, using the cipher TLS_AES_256_GCM_SHA384, while more insecure ciphers and versions of TLS are also supported if required. HTTP/3 requires the use of at least TLS 1.3 as it is part of the QUIC protocol so enabling more insecure connections is not possible.

```
 3 0.0000411… 127.0.0.1    127.0.0.1  TCP    66 38744 → 8000 [ACK] Seq=1 Ack=1 Win=33280 Len=0 TSval=3721235627 TSecr=3721235627
 4 0.0007106… 127.0.0.1    127.0.0.1  TCP   148 38744 → 8000 [PSH, ACK] Seq=1 Ack=1 Win=33280 Len=82 TSval=3721235628 TSecr=3721235627
 5 0.0007237… 127.0.0.1    127.0.0.1  TCP    66 8000 → 38744 [ACK] Seq=1 Ack=83 Win=33280 Len=0 TSval=3721235628 TSecr=3721235628
 6 0.0012509… 127.0.0.1    127.0.0.1  TCP   117 8000 → 38744 [PSH, ACK] Seq=1 Ack=83 Win=33280 Len=51 TSval=3721235628 TSecr=3721235628
 7 0.0012627… 127.0.0.1    127.0.0.1  TCP    66 38744 → 8000 [ACK] Seq=83 Ack=52 Win=33280 Len=0 TSval=3721235628 TSecr=3721235628
 8 0.0014830… 127.0.0.1    127.0.0.1  TCP    75 8000 → 38744 [PSH, ACK] Seq=52 Ack=83 Win=33280 Len=9 TSval=3721235629 TSecr=3721235628
 9 0.0014919… 127.0.0.1    127.0.0.1  TCP    66 38744 → 8000 [ACK] Seq=83 Ack=61 Win=33280 Len=0 TSval=3721235629 TSecr=3721235629
10 0.0021420… 127.0.0.1    127.0.0.1  TCP   149 38744 → 8000 [PSH, ACK] Seq=83 Ack=61 Win=33280 Len=83 TSval=3721235629 TSecr=3721235629
11 0.0022646… 127.0.0.1    127.0.0.1  TCP   113 38744 → 8000 [PSH, ACK] Seq=166 Ack=61 Win=33280 Len=47 TSval=3721235629 TSecr=3721235629
```

Figure 13. HTTP/2 packets

```
 3 0.0000405… 127.0.0.1    127.0.0.1  TCP       66 55090 → 8000 [ACK] Seq=1 Ack=1 Win=33280 Len=0 TSval=3721114249 TSecr=3721114249
 4 0.0010984… 127.0.0.1    127.0.0.1  TLSv1.3  583 Client Hello
 5 0.0011123… 127.0.0.1    127.0.0.1  TCP       66 8000 → 55090 [ACK] Seq=1 Ack=518 Win=33024 Len=0 TSval=3721114250 TSecr=3721114250
 6 0.0057807… 127.0.0.1    127.0.0.1  TLSv1.3 3638 Server Hello, Application Data, Application Data, Application Data, Application Data
 7 0.0057993… 127.0.0.1    127.0.0.1  TCP       66 55090 → 8000 [ACK] Seq=518 Ack=3573 Win=33280 Len=0 TSval=3721114255 TSecr=3721114254
 8 0.0065336… 127.0.0.1    127.0.0.1  TLSv1.3  146 Change Cipher Spec, Application Data
 9 0.0067008… 127.0.0.1    127.0.0.1  TLSv1.3  576 Application Data, Application Data
10 0.0069992… 127.0.0.1    127.0.0.1  TLSv1.3  170 Application Data
11 0.0070128… 127.0.0.1    127.0.0.1  TLSv1.3  139 Application Data
```

Figure 14. HTTPS packets

```
 1 0.0000000… 127.0.0.1    127.0.0.1  QUIC  1242 Initial, DCID=27048b15bf2ea01b, SCID=a6ab926602b3fe6c, PKN: 0, CRYPTO, PADDING
 2 0.0081277… 127.0.0.1    127.0.0.1  QUIC  1242 Handshake, DCID=a6ab926602b3fe6c, SCID=d232d73f2b6774f9
 3 0.0081601… 127.0.0.1    127.0.0.1  QUIC  1166 Handshake, DCID=a6ab926602b3fe6c, SCID=d232d73f2b6774f9
 4 0.0081707… 127.0.0.1    127.0.0.1  QUIC    95 Protected Payload (KP0), DCID=a6ab926602b3fe6c
 5 0.0093067… 127.0.0.1    127.0.0.1  QUIC   141 Handshake, DCID=d232d73f2b6774f9, SCID=a6ab926602b3fe6c
 6 0.0163816… 127.0.0.1    127.0.0.1  QUIC   399 Protected Payload (KP0), DCID=d232d73f2b6774f9
 7 0.0166306… 127.0.0.1    127.0.0.1  QUIC    77 Protected Payload (KP0), DCID=d232d73f2b6774f9
 8 0.0170442… 127.0.0.1    127.0.0.1  QUIC   274 Protected Payload (KP0), DCID=a6ab926602b3fe6c
 9 0.0171447… 127.0.0.1    127.0.0.1  QUIC   177 Protected Payload (KP0), DCID=d232d73f2b6774f9
```

Figure 15. HTTP/3 packets

For authentication and authorization, multiple mechanisms are supported by using HTTP headers. In the case of Basic Authentication, a username and password are passed to the server through the Authorization header, or in the case of JWTs, a token is passed through the same header. Other mechanisms such as OAuth or session ids stored in Cookies are also available. In any case, when using authorization, the connection should be encrypted, so credentials are not exposed to the rest of the network. For these experiments Basic Authentication was used.

```
# Read Basic Auth credentials from file
f = open("keys/credentials.json")
credentials = json.load(f)
auth = (credentials["username"], credentials["password"])
f.close()

response = await client.post('https://http-server:8000/time', json=payload, auth=auth)
```

Figure 16. HTTP connection with credentials

4.3 Connection Assessment

Availability is another important aspect of an application. After creating and testing the basic connections, we can assess them on the amount of data they can transport, number of messages and their speed, and see if there is any loss when dealing with higher volumes. We will also enable the different quality of service options on the protocols that provide that functionality. Due to differences in architecture, MQTT and CoAP will be compared, using the pub/sub architecture, with a publisher/server sending multiple messages with big payloads at different speeds, and a subscriber/client calculating the transfer speed and detecting possible data losses, while HTTP will be tested with GET requests initiated by the client and big responses returned by the server.

4.3.1 MQTT

For testing the connection, a publisher script was created, which sends multiple payloads of a specified size to a topic, at an interval. A subscriber script listens to the same topic, validating the payload size of each packet to detect possible losses and calculating the transfer speed. The payload size, number of packets and interval are configurable, and different combinations were tested. When no interval is specified, the script tries to push packets at the maximum possible speed, which is then calculated on the subscriber side.

For minimizing data loss, MQTT provides three quality of service levels. Level 0 is a simple transmission, level 1 guarantees the packet is received at least once, and level 2 guarantees the packet is received exactly once without duplicates. Higher levels use more packets for the transmission of acknowledgments and retransmission of messages in cases of packet loss. To minimize data loss and ensure correct tracking of received messages, quality of service 2 was used for the main tests.

The selected security configuration from the previous section was also applied. That is, the following tests were conducted using the default TLS attributes provided by the broker (TLS 1.3, cipher TLS_AES_256_GCM_SHA384) with certificate authentication for both the broker and clients, using 4096-bit RSA keys.

Testing with different transfer speeds no data loss was detected at speeds of 1, 100 and 1000 messages/second. For smaller payloads of 1 to 1000 bytes, there is no data loss or big loss in transfer speed measured by the client. However, at higher frequencies of sending about 100 messages/second the requested speed cannot be maintained for larger payloads.

Payload Size	Speed
1 byte	94.683 messages/second
10 bytes	94.733 messages/second
100 bytes	94.942 messages/second
1000 bytes	95.215 messages/second
10000 bytes	93.579 messages/second
100000 bytes	84.330 messages/second
1000000 bytes	47.525 messages/second
10000000 bytes	8.357 messages/second

Table 8. MQTT 100 messages/second test per payload size

During the transfer, the recorded transfer speed may change. Here, a higher speed is recorded at the beginning of the transfer for smaller payloads that then decreases and stays mostly stable, while for larger payloads, there is a smaller speed from the start that then fluctuates a lot.

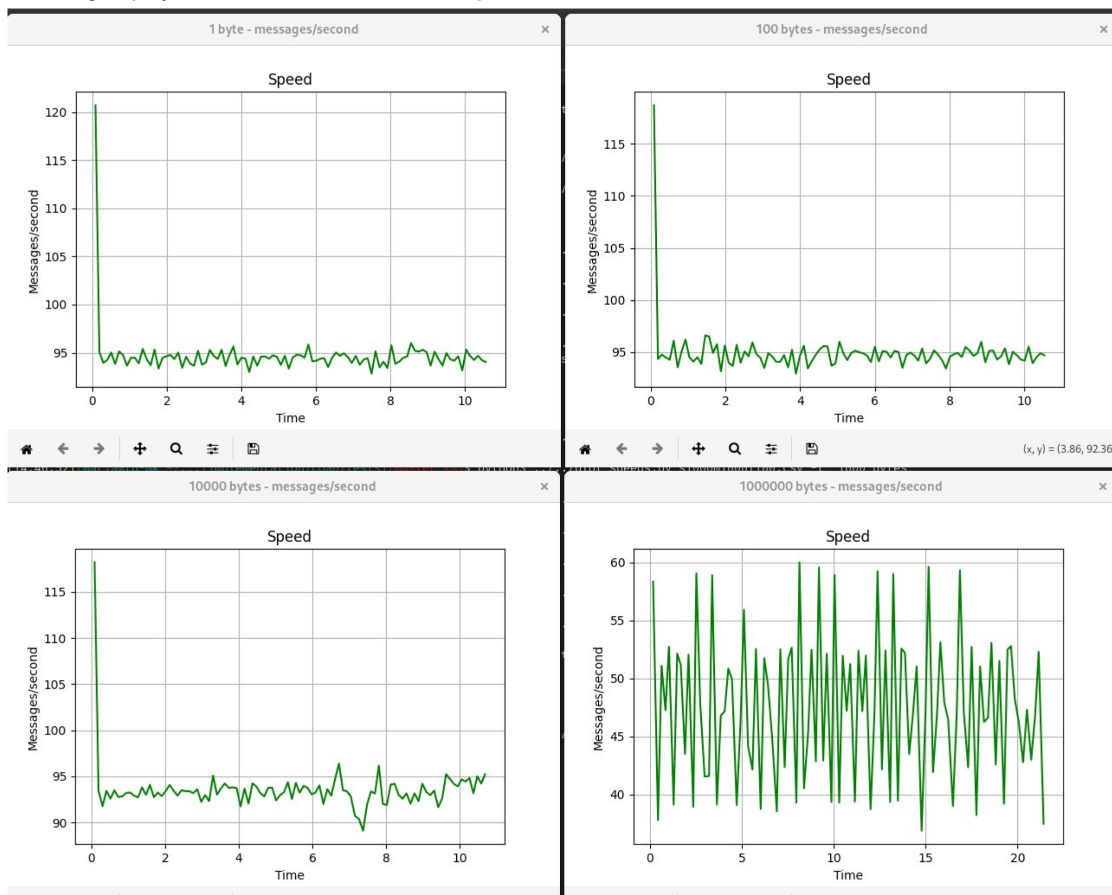

Figure 17. MQTT speeds in time for 100 messages/second per payload size

For even larger payloads some data loss is also observed, with the maximum safe payload at around 25000000 bytes, and very low transfer speeds.

When not specifying a transfer speed, the publisher will try to send messages as fast as possible and the subscriber will receive them as soon as they pass through the broker. This defines the maximum transfer speed and is affected by the payload size. Higher quality of service also means lower transfer speeds. Specifically for QoS 2 (highest level), the below speeds are recorded.

Payload Size	Speed
1 byte	7283.225 messages/second
10 bytes	7592.324 messages/second
100 bytes	7193.931 messages/second
1000 bytes	6878.299 messages/second
10000 bytes	4719.285 messages/second
100000 bytes	2354.147 messages/second
1000000 bytes	67.170 messages/second
10000000 bytes	3.237 messages/second

Table 9. MQTT max transfer speeds per payload size

At smaller payload sizes transfer speed is not greatly affected but with larger payloads, the speed decreases significantly. Some big drops may also be experienced during the transfer, even in smaller payloads, possibly due to lost packages that get retransmitted. The speed also decreases at the end of the transfer which may depend on the method of measuring it. For payloads over 5000000 bytes per message, data loss is observed.

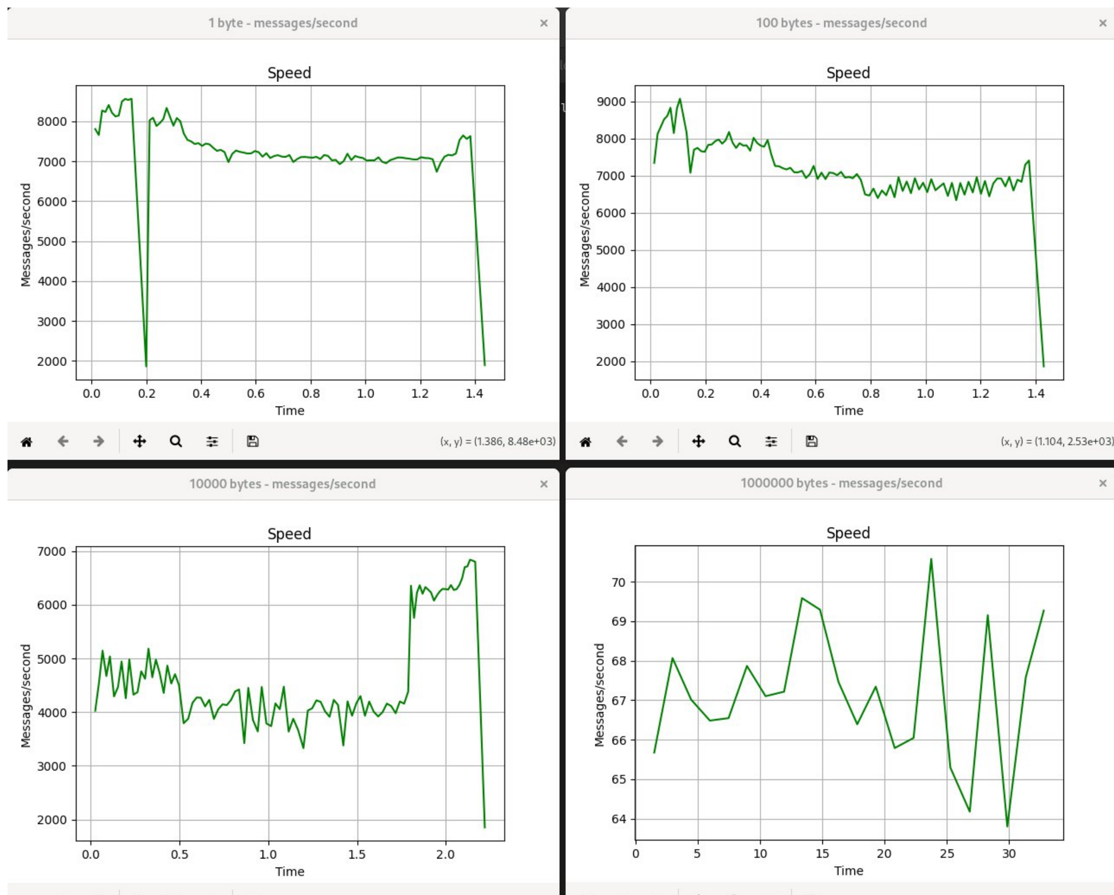

Figure 18. MQTT max speeds in time per payload size

Using a small payload size (10 bytes), increasing the quality of service level results in a decrease in speed as more packets are now exchanged during the transfer of one message.

QoS	Speed
0	13419.334 messages/second
1	8608.695 messages/second
2	7176.055 messages/second

Table 10. MQTT max transfer speeds per Quality of Service level (payload size 10)

The observed speed for the duration of the transfer shows a lot of variation in all cases, with QoS 2 being generally more stable but slower due to the increase in packets for each message, except for a big decrease at the end. Higher quality of service also results in less frequent data loss.

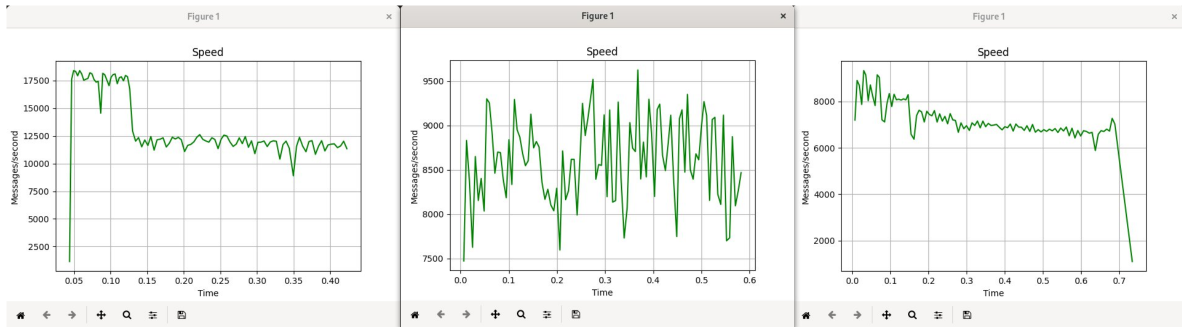

Figure 19. MQTT QoS 0,1,2 speeds in time

When multiple clients subscribe to the same topic, all of them receive the messages published on that topic. And when multiple clients publish to the same topic, subscribers of that topic receive the messages from all publishers. The number of clients running at the same time affects the speed with which messages are transmitted to subscribers. Tests bellow were conducted on the same machine. When clients run on separate machines, the speed decrease is less dramatic but still noticeable.

Number of subscribers	Speed
1	7621.712 messages/second
2	6796.350 messages/second
5	3250.519 messages/second
10	1463.447 messages/second

Table 11. MQTT max transfer speeds per number of subscribers

Number of publishers	Speed	Speed per client
1	7376.656 messages/second	7376.656 messages/second
2	7675.428 messages/second	3837.714 messages/second
5	11619.026 messages/second	2323.805 messages/second
10	12142.995 messages/second	1214.300 messages/second

Table 12. MQTT max transfer speeds per number of publishers

Number of publishers/subscribers	Speed	Speed per client
1	7250.901 messages/second	7250.901 messages/second
2	7872.237 messages/second	3936.119 messages/second
5	4870.221 messages/second	974.044 messages/second
10	2121.840 messages/second	212.184 messages/second

Table 13. MQTT max transfer speeds per number of publishers and subscribers

Examining the transfer speed during the transfer in one of the subscribers, the plots are similar to the tests with a single subscriber, with a drop in speed at the end of the transfer, but as the number of subscribers increases the general transfer speed decreases, both due to the increase in process power required in the host machine and due to the extra load on the broker.

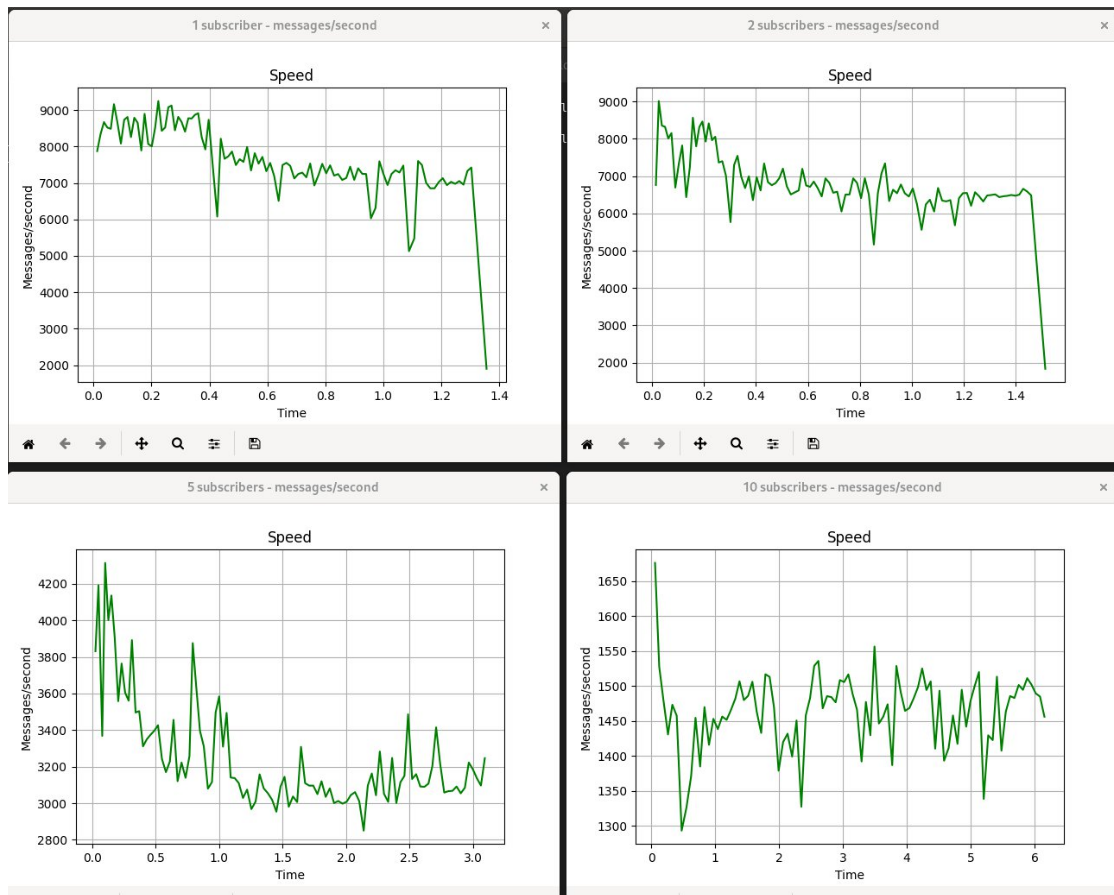

Figure 20. MQTT max speeds in time per number of clients (10 bytes payload)

When the subscriber, publisher and broker need to communicate through a network that may be less stable, the speed can be affected significantly. Placing the subscriber, publisher or both on a network where they communicate with the broker through Wi-Fi greatly reduces the max transfer speed.

Payload Size	Speed
1 byte	306.546 messages/second
10 bytes	307.129 messages/second
100 bytes	304.222 messages/second
1000 bytes	249.009 messages/second
10000 bytes	227.382 messages/second
100000 bytes	180.568 messages/second
1000000 bytes	27.315 messages/second
10000000 bytes	14.185 messages/second

Table 14. MQTT max transfer speeds per payload size on wifi

The transfer starts at a relatively high speed which is decreased later. Some spikes in speed may also be observed, possibly due to the medium being less reliable and causing retransmissions, as quality of service 2 was used, to minimize data loss.

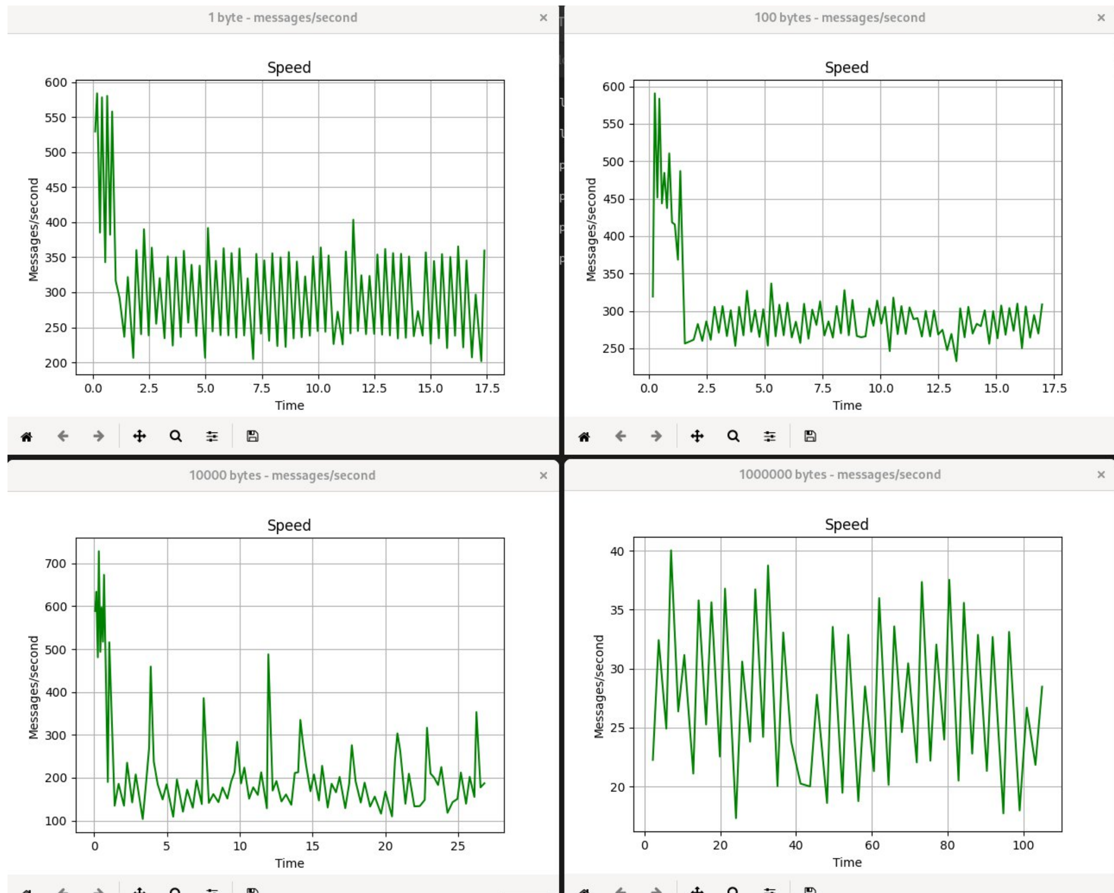

Figure 21. MQTT max speeds in time per payload size on wifi

When messages are sent at a steady rate of 100 messages/second, that speed is maintained in the final output, but with big drops for larger payloads.

Payload Size	Speed
1 byte	98.474 messages/second
10 bytes	98.550 messages/second
100 bytes	98.741 messages/second
1000 bytes	100.267 messages/second
10000 bytes	98.031 messages/second
100000 bytes	90.756 messages/second
1000000 bytes	28.173 messages/second
10000000 bytes	2.580 messages/second

Table 15. MQTT 100 messages/second test per payload size on wifi

The plots of observed speed for the duration of the transfer are similar to before, with a higher speed at the beginning but with less variation during the transfer. The results on WiFi are generally not too different to the results of the 100 messages/second test on a single machine, showing that the unreliable connection may not be as apparent in slower speeds and acts more as a bottleneck for the observed transfer speed.

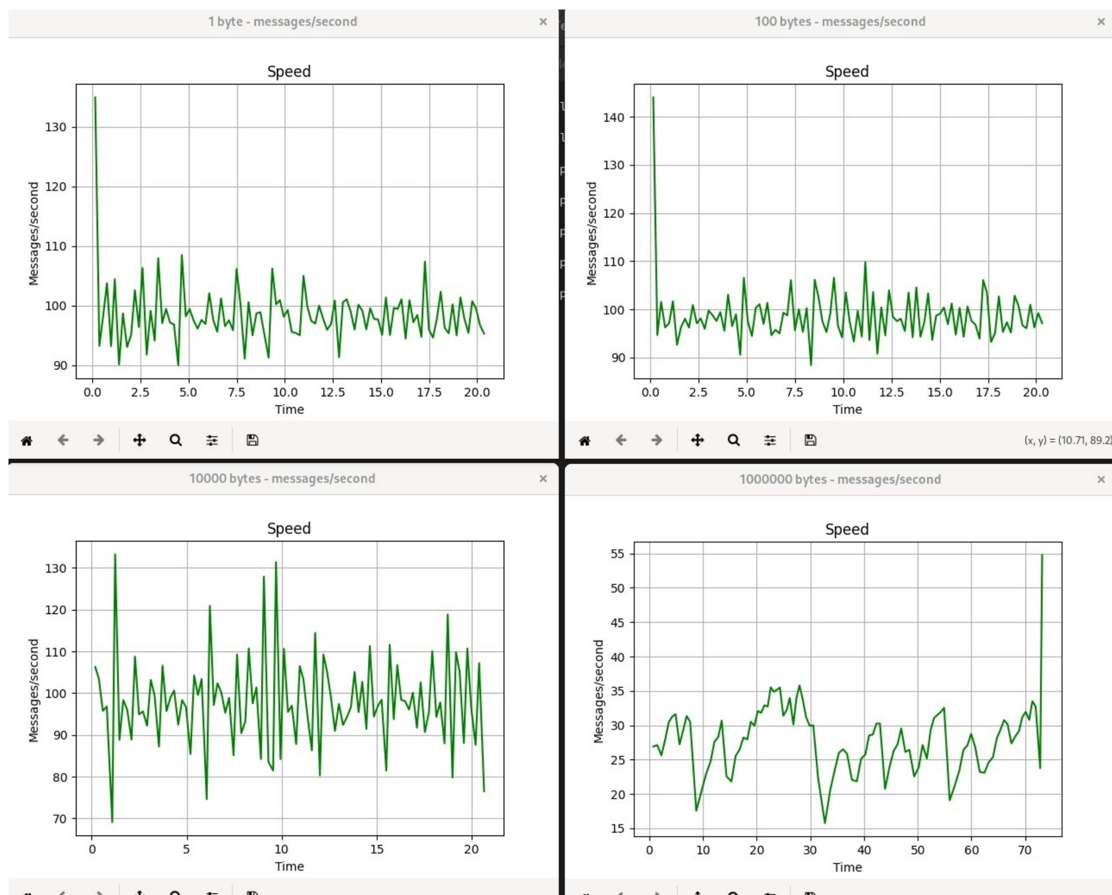

Figure 22. MQTT speeds in time for 100 messages/second per payload size on wifi

4.3.2 CoAP

Similar to MQTT, a server script was used that transmits multiple messages of a specified payload size at intervals when a client subscribes to a resource. The client then validates the sizes of received payloads and calculates the transfer speed. The maximum transfer speed, with no interval, was also tested.

When assessing availability, CoAP offers two types of messages, confirmable (CON) and non-confirmable (NON). To minimize data loss, confirmable messages were used in tests, which verify that a message was received by sending back a confirmation for each message and, if not received, retransmits it.

Again the security configuration of the previous section was used in all tests, with the default DTLS configuration provided by the library, using PSKs for securing the connection.

During testing with different delays between messages, no data loss was observed. Speeds of 1, 100 and 1000 messages/second were tested. Likewise, an increase in payload size did not impact the measured transfer speed, with payloads of 1, 10, and 1000 bytes. However the DTLS implementation of the library seems to have issues with larger payloads and transmissions with a size over 1300 bytes are rejected and retransmitted without reaching the client. Without using DTLS, a similar problem is observed for payload sizes over 4087 bytes. At a higher payload size of 65498 bytes, a socket error is triggered on the server side for the message being too long. This could be related to the implementation of Block-Wise Transfers (RFC 7959 [22]).

When sending messages at a speed of 100 messages/second, the requested speed cannot be maintained but the final recorded speed does not seem correlated to the payload size.

Payload Size	Speed
1 byte	85.952 messages/second
10 bytes	85.664 messages/second
100 bytes	85.924 messages/second
1000 bytes	87.440 messages/second

Table 16. CoAP 100 messages/second test per payload size

In all cases, the transfer starts fast and then the speed decreases but doesn't fluctuate too much, possibly due to the lower rate at which messages are sent, which doesn't reach any limits of the protocol. As CoAP is stateless, there is probably a connection delay on each message that is responsible for the final ~85 messages/second transfer speed.

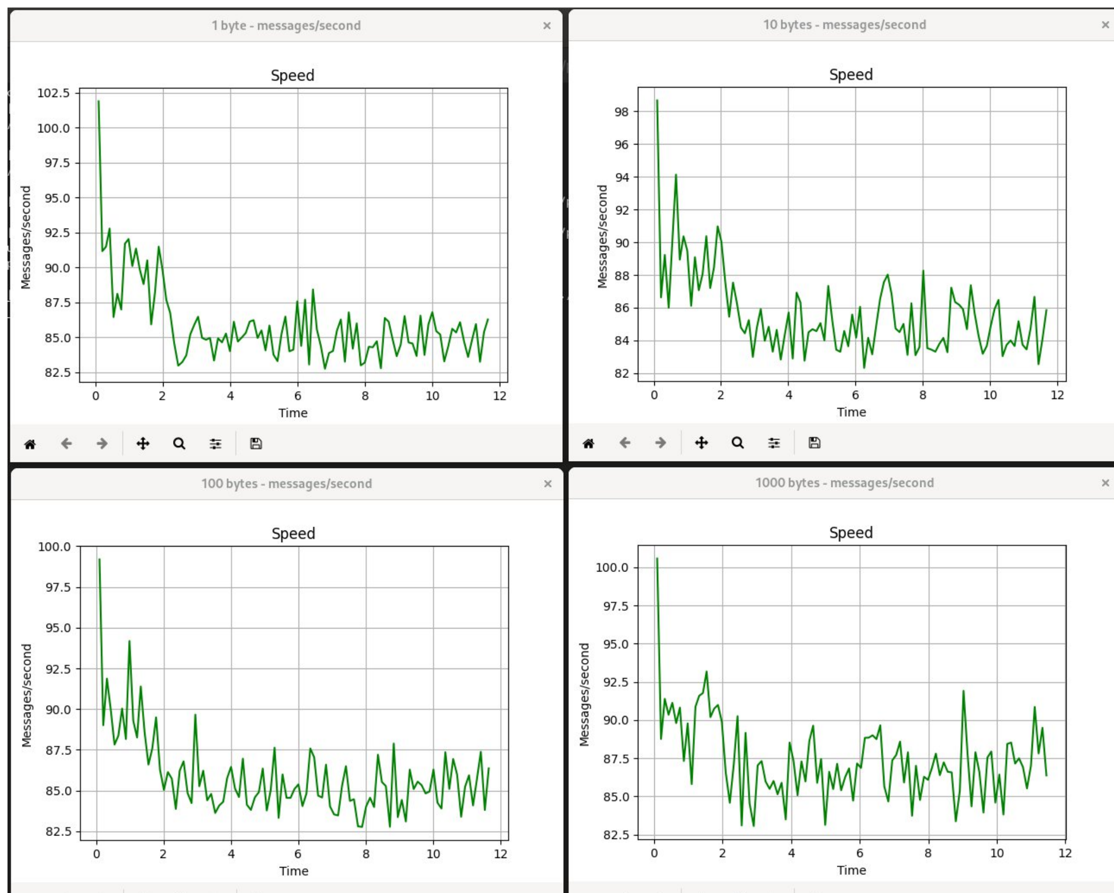

Figure 23. CoAP speeds in time for 100 messages/second per payload size

When not specifying a transfer speed, the server will try to send messages as fast as possible. This defines the maximum transfer speed and is affected by the payload size. Specifically, for the tested sizes, while using DTLS, the below transfer speeds are observed.

Payload Size	Speed
1 byte	1151.144 messages/second
10 bytes	1163.623 messages/second
100 bytes	1156.274 messages/second
1000 bytes	1109.002 messages/second

Table 17. CoAP max transfer speeds per payload size

Again the speed is not affected much by the payload size. However, now it starts slower and increases with time, varying more for the duration of the transfer, as it is not limited by the rate it is pushed but reaches the limits of the protocol.

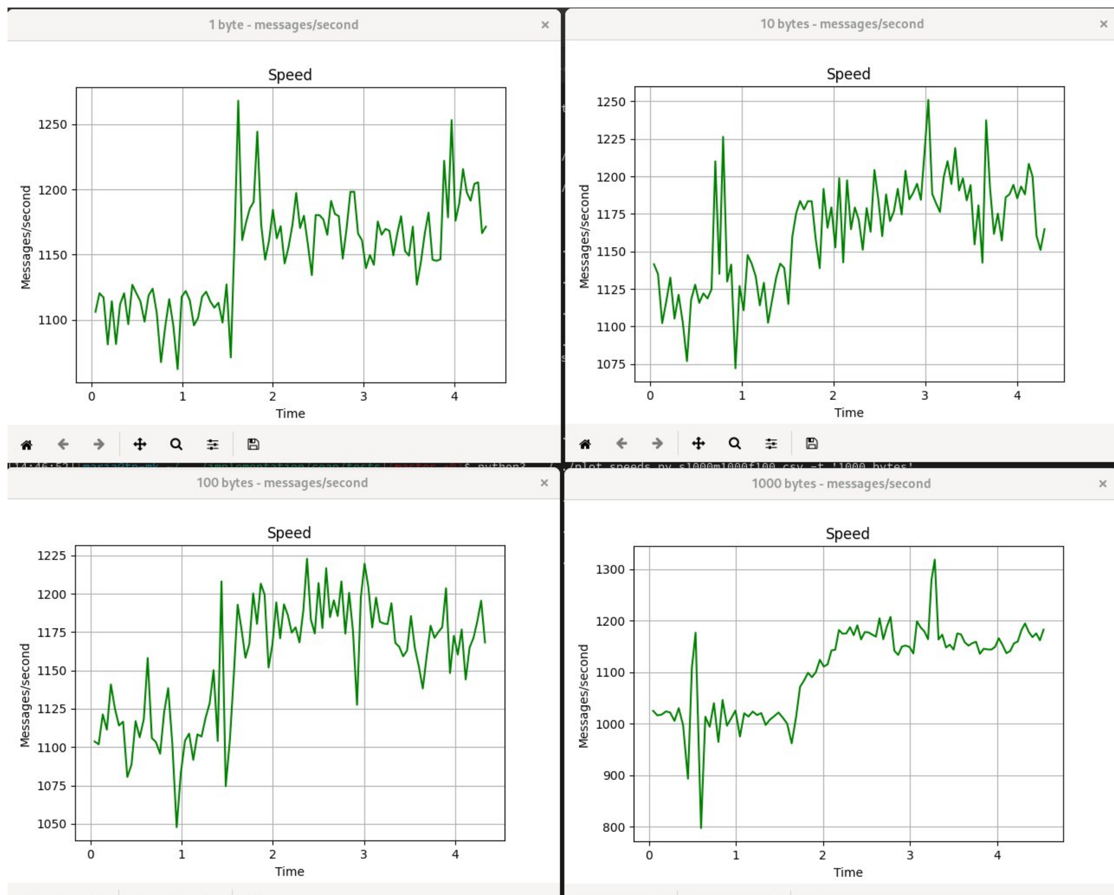

Figure 24. CoAP max speeds in time per payload size

When multiple clients connect to the same server, updates to resources that multiple clients observe are transmitted to all of them. Resources on the server are shared between them and the maximum speed decreases as the number of clients increases. Tests with multiple clients on the same and separate machines show a similar decrease in speed, showing the main limiting factor being the server.

Number of clients	Speed
1	1192.428 messages/second
2	857.847 messages/second
5	509.915 messages/second
10	316.340 messages/second

Table 18. CoAP max transfer speeds per number of clients (10 bytes payload)

Examining the changes in transfer speed during time in one of the clients, there are large increases in speed at specific points in time. This is possibly due to the transfer finishing for other clients and more resources getting allocated for the current client.

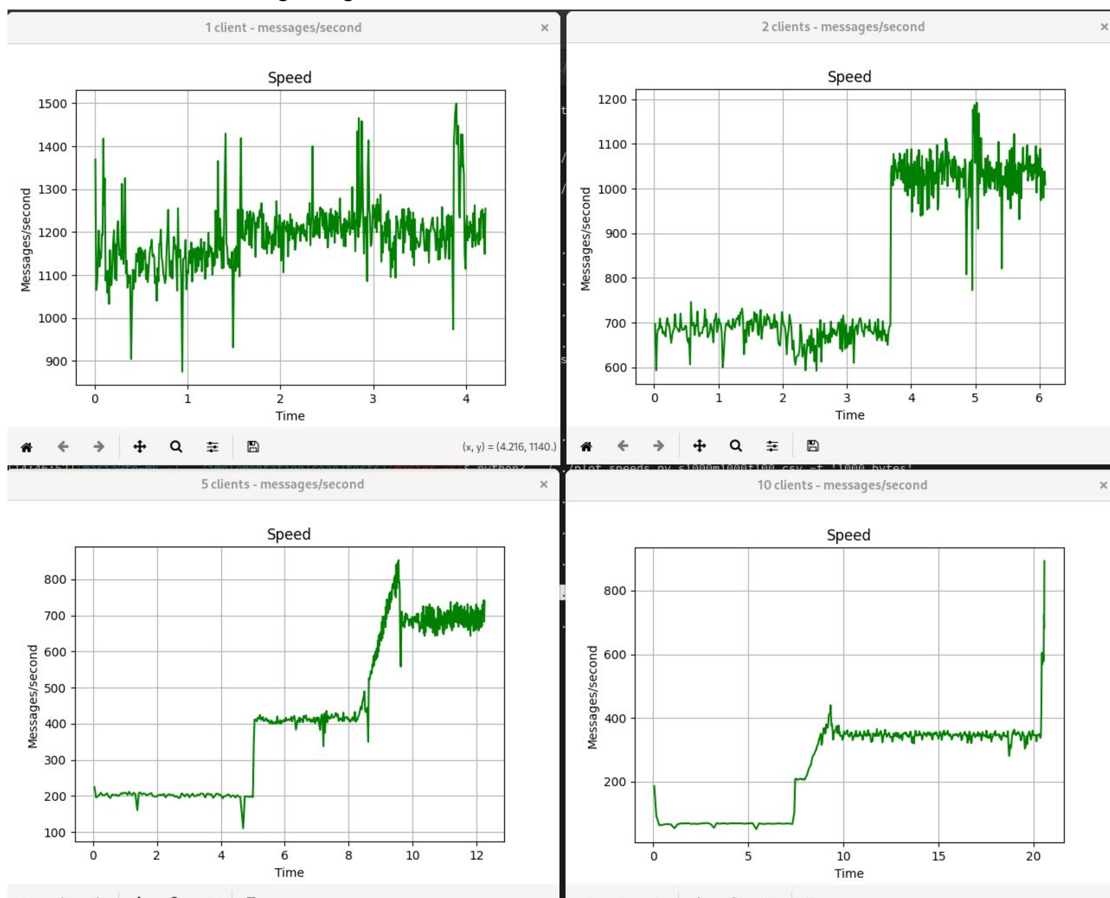

Figure 25. CoAP max speeds in time per number of clients (10 bytes payload)

When server and client are running on different machines connected through a Wi-Fi network, the max transfer speed is decreased significantly but increasing the payload has no effect.

Payload Size	Speed
1 byte	93.781 messages/second
10 bytes	83.221 messages/second
100 bytes	85.309 messages/second
1000 bytes	82.720 messages/second

Table 19. CoAP max transfer speeds per payload size on wifi

Transfer speed starts high and then decreases, with more fluctuation than before, which makes sense given the unreliable nature of the connection.

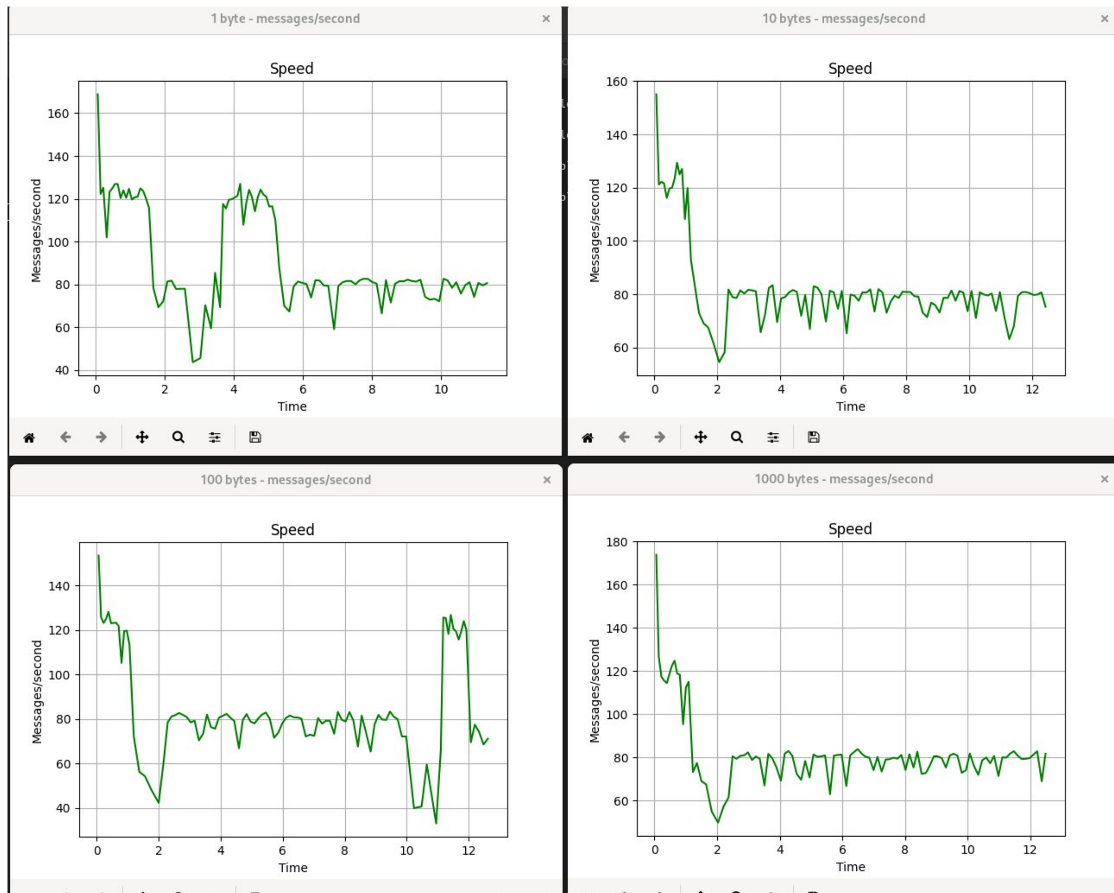

Figure 26. CoAP max speeds in time per payload size on wifi

While sending messages at a rate of 100 messages/second, however, a steady pace can be kept regardless of payload size, showing that a WiFi connection may offer a more reliable speed for connections with limited sending rates.

Payload Size	Speed
1 byte	93.524 messages/second
10 bytes	93.587 messages/second
100 bytes	93.352 messages/second
1000 bytes	93.542 messages/second

Table 20. CoAP 100 messages/second test per payload size on wifi

Here too, the transfer starts at a relatively higher speed and then varies a lot throughout the transfer, similarly to the previous non-limited case.

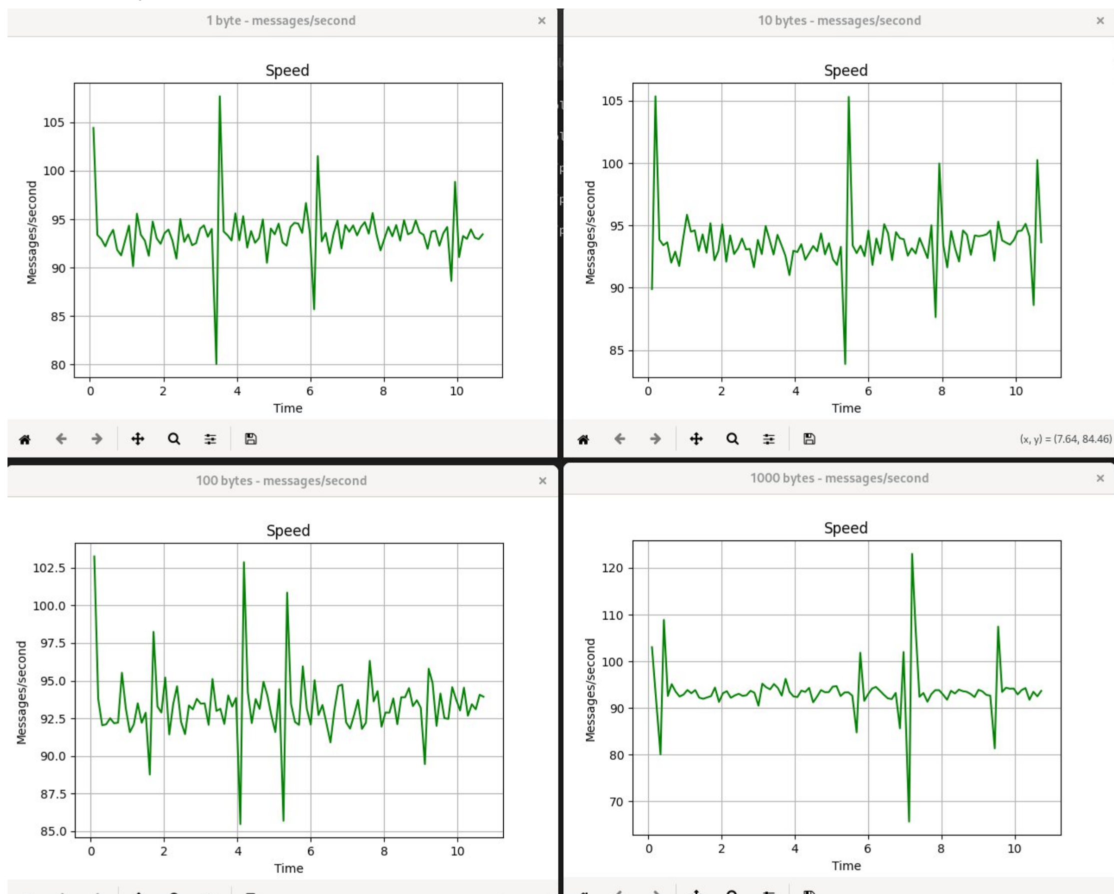

Figure 27. CoAP speeds in time for 100 messages/second per payload size on wifi

4.3.3 HTTP

Since HTTP operates on a request – response model, tests were conducted in a different way to the previous protocols. A client script was created which sends a number of requests to a server that then replies with a payload of a specified size. The transfer speed is calculated on the client based on the rate of received responses and the size of received data is also validated.

The security configuration used for all tests is the same as that of the previous section. HTTP/2 uses TLS 1.3 with cipher TLS_AES_256_GCM_SHA384 and Basic Authentication for the clients. HTTP/3 uses the same encryption algorithm through QUIC with Basic Authentication again for the clients.

Two methods can be used when sending requests. The HTTP client can send the requests sequentially, waiting for a response before sending the next request, or send all requests in parallel. Both approaches were tested to calculate the maximum transfer speeds.

HTTP/2 and HTTP/3 were tested separately and presented differences in calculated times. Both are slower than the other protocols when executing requests sequentially, with HTTP/3 being considerably faster, due to using UDP which skips the overhead of the TCP handshake. No data loss was observed even with payloads of 50000000 bytes but the speed is greatly decreased to less than 1 message/second.

Payload Size	HTTP/2	HTTP/3
1 byte	18.810 messages/second	429.113 messages/second
10 bytes	18.806 messages/second	443.050 messages/second
100 bytes	19.017 messages/second	412.914 messages/second
1000 bytes	18.735 messages/second	441.086 messages/second
10000 bytes	18.789 messages/second	262.795 messages/second
100000 bytes	19.879 messages/second	57.465 messages/second
1000000 bytes	15.854 messages/second	5.796 messages/second
10000000 bytes	4.083 messages/second	0.348 messages/second

Table 21. HTTP max transfer speeds per payload size

Throughout the transfer HTTP/2 maintains a relatively stable speed while HTTP/3 shows multiple spikes, possibly due to its use of UDP as an underlying protocol. HTTP/3 generally remains faster than HTTP/2 but its speed decreases a lot more as the payload size increases until it becomes slower than HTTP/2 at around 500000 bytes of payload.

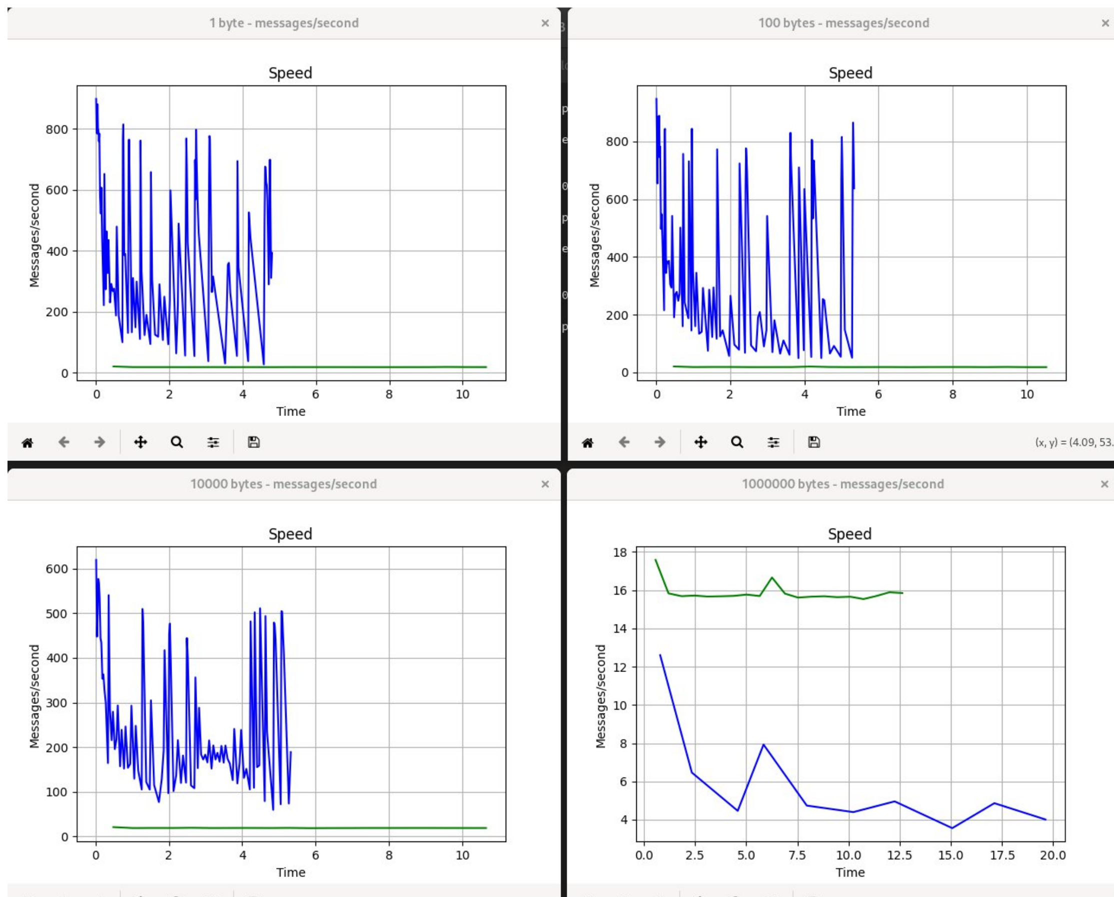

Figure 28. HTTP max speeds in time per payload size (HTTP/2 green, HTTP/3 blue)

Since the transfer speed is already low, no tests were conducted at a frequency of 100 messages/second for HTTP/2 as it cannot be reached. HTTP/3 can maintain a speed of around 81 messages/second for smaller payloads which then decreases a lot as the payload increases.

Payload Size	HTTP/3
1 byte	81.052 messages/second
10 bytes	81.921 messages/second
100 bytes	81.687 messages/second
1000 bytes	81.891 messages/second
10000 bytes	45.982 messages/second
100000 bytes	45.736 messages/second
1000000 bytes	5.011 messages/second
10000000 bytes	0.314 messages/second

Table 22. HTTP/3 100 messages/second test per payload size

Again there are big drops in speed, especially towards the end of the transfer, proving HTTP/3 as not particularly stable when it comes to transfer speed.

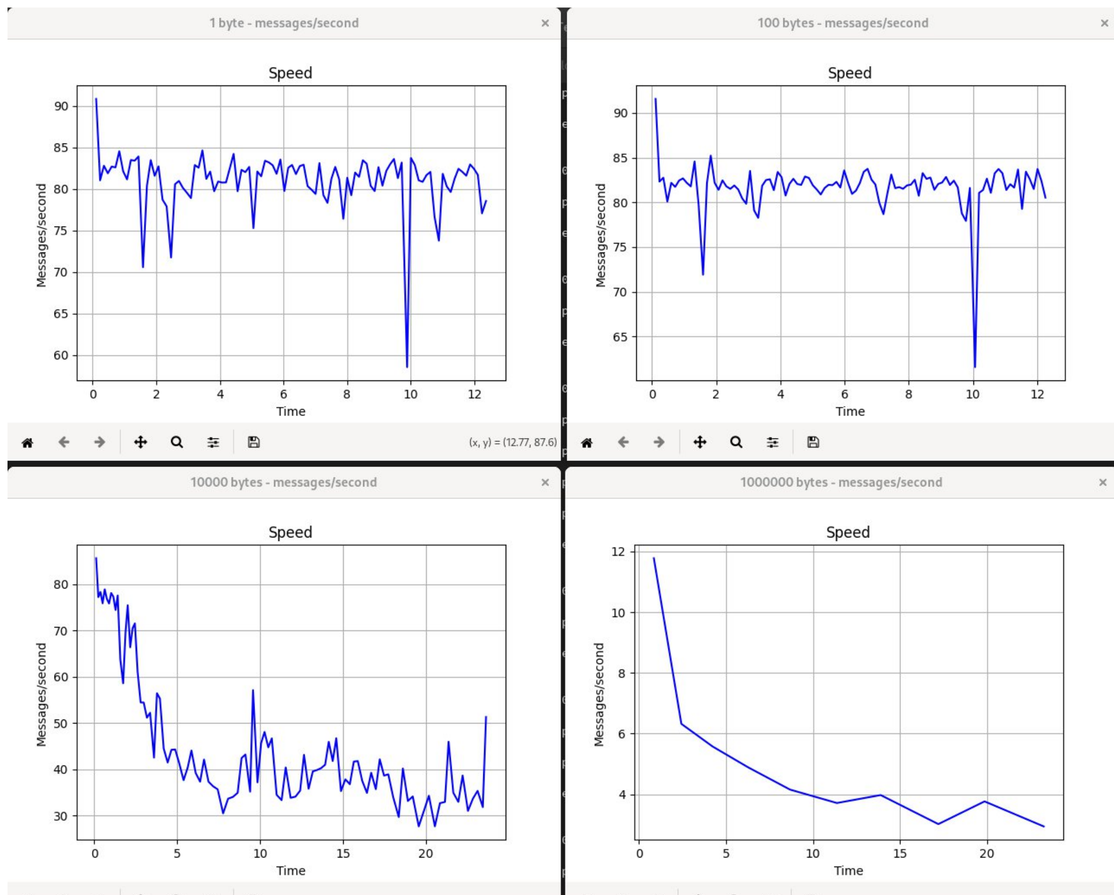

Figure 29. HTTP/3 speeds in time for 100 messages/second per payload size

When executing requests in parallel, the transfer speed is much higher for both HTTP/2 and HTTP/3 as all messages are sent at the same time, opening different connections with the server. However, the size of data that can be processed in parallel is limited so increases in payload size decrease the speed. HTTP/2 in this case is faster, possibly due to the server holding less information for each connection. The amount of messages that can be sent in parallel is also limited in HTTP/2 and, depending on the payload size, connections may be dropped after 100-1000 messages. HTTP/3 does not experience errors or drops but connections are much slower.

Payload Size	HTTP/2	HTTP/3
1 byte	6022.219 messages/second	2993.113 messages/second
10 bytes	4514.652 messages/second	3609.967 messages/second
100 bytes	5119.543 messages/second	2820.474 messages/second
1000 bytes	5458.627 messages/second	1344.232 messages/second
10000 bytes	5262.394 messages/second	133.014 messages/second
100000 bytes	3658.901 messages/second	37.544 messages/second
1000000 bytes	703.013 messages/second	5.284 messages/second
10000000 bytes	-	0.236 messages/second

Table 23. HTTP max transfer speeds per payload size with parallel requests

When sending requests in parallel, batches of requests return at the same time, causing spikes to appear in the measurements. HTTP/2 and HTTP/3 work similarly but HTTP/3 maintains a much slower stable speed at higher payloads.

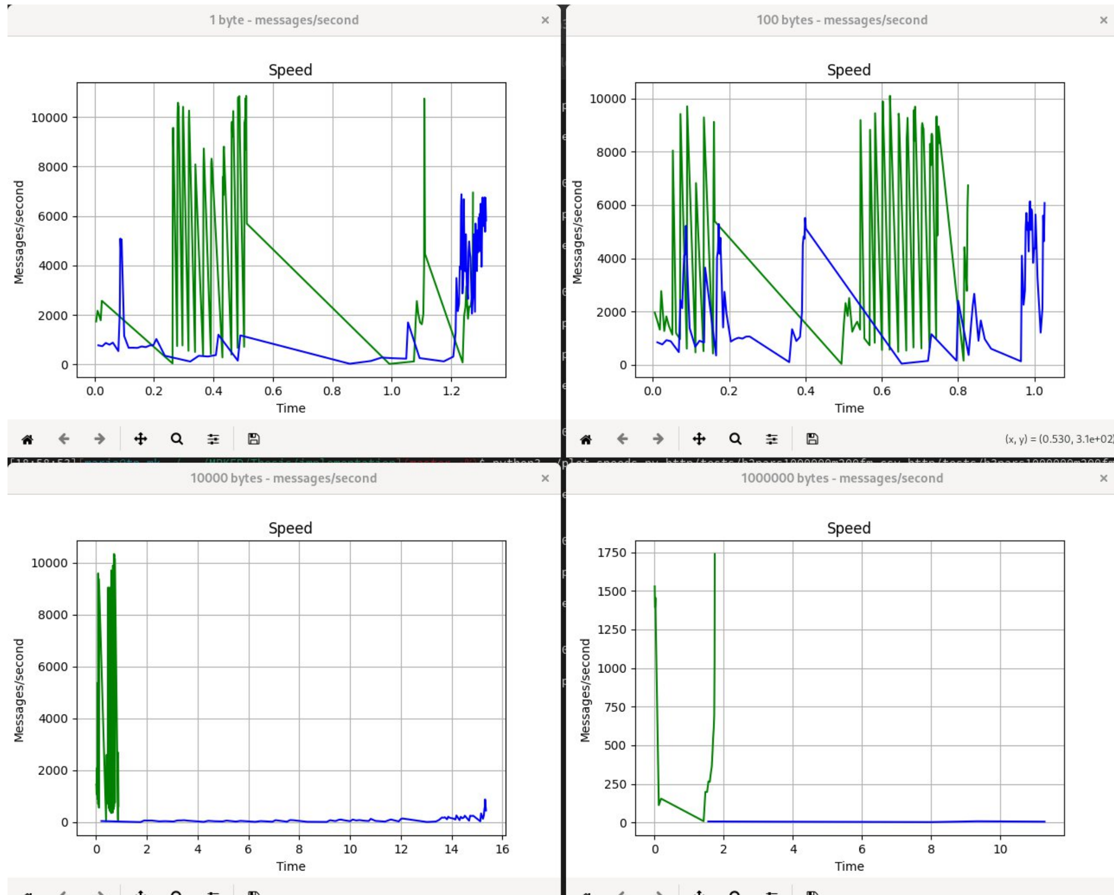

Figure 30. HTTP max speeds in time per payload size with parallel requests (HTTP/2 green, HTTP/3 blue)

Usually multiple clients will be communicating with a single server. Testing the connection with multiple clients sending requests at the same time does not show a decrease in speed in HTTP/2, probably because the server is designed to serve multiple clients with higher loads. In the higher speeds of HTTP/3 the difference is visible as the bandwidth seems to be distributed to the clients.

Number of clients	HTTP/2	HTTP/3
1	18.870 messages/second	444.894 messages/second
2	18.422 messages/second	286.334 messages/second
5	18.517 messages/second	94.667 messages/second
10	16.401 messages/second	43.123 messages/second

Table 24. HTTP max transfer speeds per number of clients (10 bytes)

Speeds here are similar to the other cases with HTTP/3 having variable speeds during the transfer and HTTP/2 being slow but stable, meaning that the increase in clients does not greatly affect the behavior of the server, only the average transfer speed.

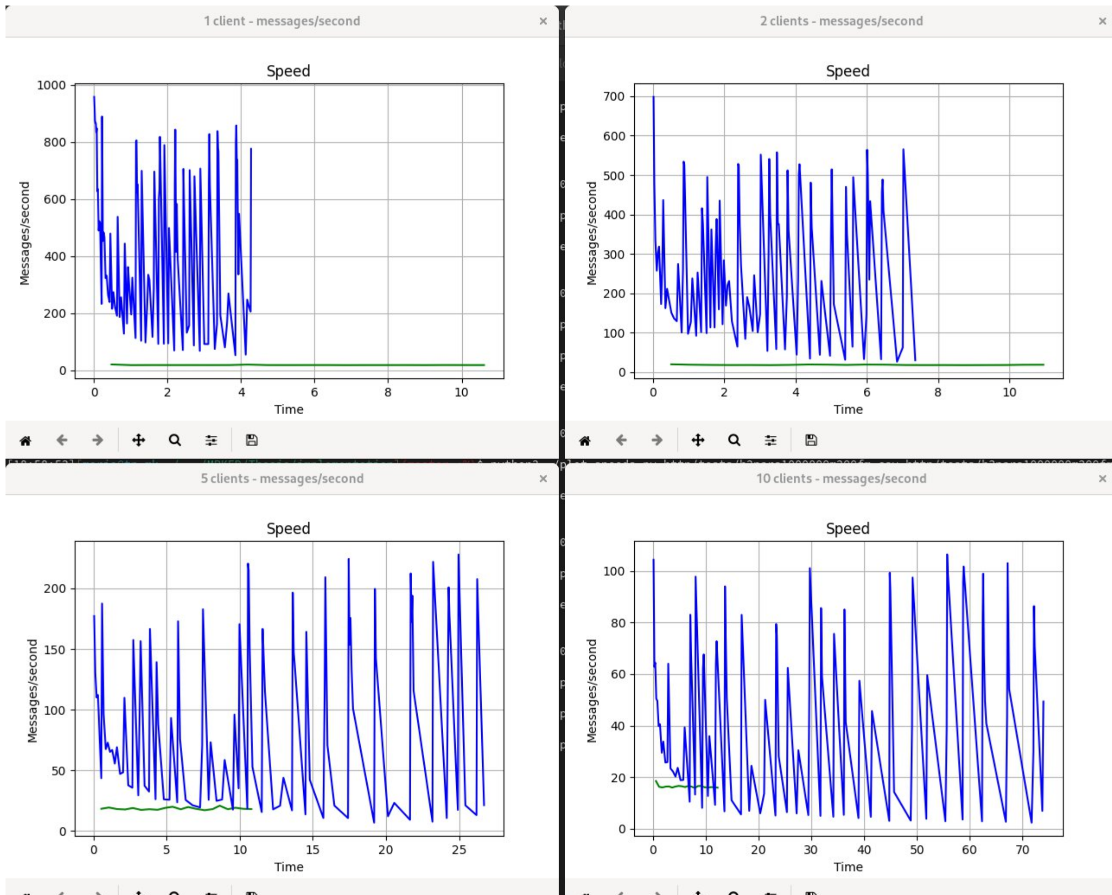

Figure 31. HTTP max speeds in time per number of clients (10 bytes payload) (HTTP/2 green, HTTP/3 blue)

When server and client are running on different machines connected through a Wi-Fi network, the speed is decreased even more but stays relatively stable at smaller payloads. HTTP/3 sees a bigger drop in speed and does worse than HTTP/2 at bigger payloads.

Payload Size	HTTP/2	HTTP/3
1 byte	10.326 messages/second	68.584 messages/second
10 bytes	11.110 messages/second	68.843 messages/second
100 bytes	11.026 messages/second	74.287 messages/second
1000 bytes	9.221 messages/second	70.526 messages/second
10000 bytes	14.310 messages/second	67.358 messages/second
100000 bytes	6.604 messages/second	26.127 messages/second
1000000 bytes	5.209 messages/second	3.091 messages/second
10000000 bytes	1.767 messages/second	0.206 messages/second

Table 25. HTTP max transfer speeds per payload size on wifi

On Wi-Fi, speeds during the transfer are not as stable for HTTP/2 while HTTP/3 shows drops for larger time periods due to the unstable nature of the network causing delays. The use of QUIC

as an underlying protocol may also affect HTTP/3 in unreliable network conditions as it handles retransmissions of lost packages over UDP.

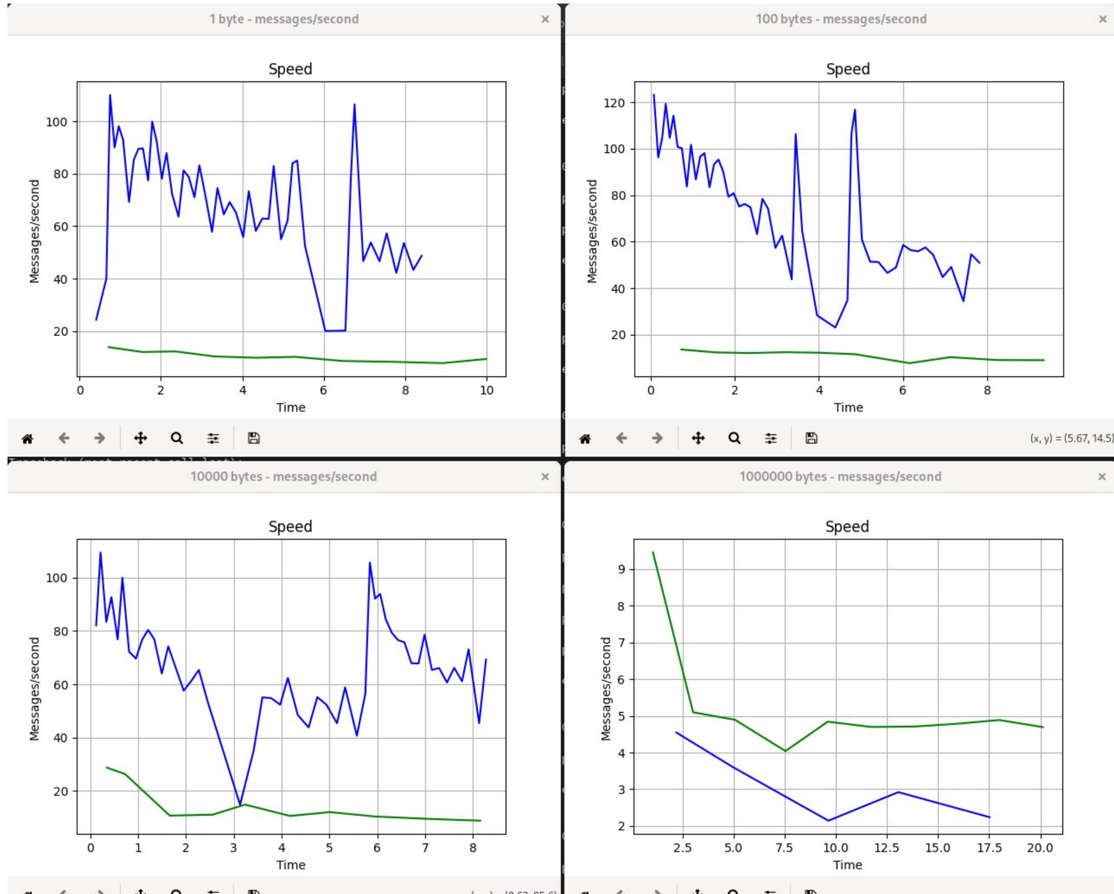

Figure 32. HTTP max speeds in time per payload size on wifi (HTTP/2 green, HTTP/3 blue)

4.3.4 Speed comparisons

After testing the connections under the same conditions, some comparisons can be made on their efficiency where possible.

When sending messages at a specific rate, both MQTT and CoAP maintain a relatively high transfer speed with MQTT being closer to the original rate (here 100 messages/second). MQTT is also able to transfer much larger payloads, however the transfer speed greatly decreases at payloads over 10000 bytes. When testing connections of a single server or publisher and multiple clients or subscribers, depending on the architecture, MQTT and CoAP see a similar decrease in transfer speed as clients increase, while MQTT remains the faster protocol. When traffic has to pass through the unreliable connection of a Wi-Fi network all protocols see a big decrease in speed, with the change being more dramatic for MQTT and HTTP/3. HTTP, when used sequentially, is slower than the other two. HTTP/3 is much faster than HTTP/2 in all scenarios, due to the underlying UDP connection, but has issues with very large payloads. Sending requests in parallel with HTTP can be faster but limited and much heavier on the clients, which may not be able to support it well when running on constrained devices. When not using encryption MQTT and CoAP see a sizable increase in speed, with the change being more dramatic for CoAP.

Payload Size	MQTT	CoAP	HTTP/2	HTTP/3
1 byte	94.683 m/s	85.952 m/s	-	81.052 m/s
10 bytes	94.733 m/s	85.664 m/s	-	81.921 m/s
100 bytes	94.942 m/	85.924 m/s	-	81.687 m/s
1000 bytes	95.215 m/s	87.440 m/s	-	81.891 m/s
10000 bytes	93.579 m/s	-	-	45.982 m/s
100000 bytes	84.330 m/s	-	-	45.736 m/s
1000000 bytes	47.525 m/s	-	-	5.011 m/s
10000000 bytes	8.357 m/s	-	-	0.314 m/s

Table 26. 100 messages/second test per payload size

Payload Size	MQTT	CoAP	HTTP/2	HTTP/3
1 byte	7283.225 m/s	1151.144 m/s	18.810 m/s	429.113 m/s
10 bytes	7592.324 m/s	1163.623 m/s	18.806 m/s	443.050 m/s
100 bytes	7193.931 m/s	1156.274 m/s	19.017 m/s	412.914 m/s
1000 bytes	6878.299 m/s	1109.002 m/s	18.735 m/s	441.086 m/s
10000 bytes	4719.285 m/s	-	18.789 m/s	262.795 m/s
100000 bytes	2354.147 m/s	-	19.879 m/s	57.465 m/s
1000000 bytes	67.170 m/s	-	15.854 m/s	5.796 m/s
10000000 bytes	3.237 m/s	-	4.083 m/s	0.348 m/s

Table 27. max transfer speeds per payload size

Number of clients	MQTT	CoAP	HTTP/2	HTTP/3
1	7621.712 m/s	1192.428 m/s	18.870 m/s	444.894 m/s
2	6796.350 m/s	857.847 m/s	18.422 m/s	286.334 m/s
5	3250.519 m/s	509.915 m/s	18.517 m/s	94.667 m/s
10	1463.447 m/s	316.340 m/s	16.401 m/s	43.123 m/s

Table 28. max transfer speeds per number of clients/subscribers (10 bytes payload)

Payload Size	MQTT	CoAP	HTTP/2	HTTP/3
1 byte	306.546 m/s	93.781 m/s	10.326 m/s	68.584 m/s
10 bytes	307.129 m/s	83.221 m/s	11.110 m/s	68.843 m/s
100 bytes	304.222 m/s	85.309 m/s	11.026 m/s	74.287 m/s
1000 bytes	249.009 m/s	82.720 m/s	9.221 m/s	70.526 m/s
10000 bytes	227.382 m/s	-	14.310 m/s	67.358 m/s
100000 bytes	180.568 m/s	-	6.604 m/s	26.127 m/s
1000000 bytes	27.315 m/s	-	5.209 m/s	3.091 m/s
10000000 bytes	14.185 m/s	-	1.767 m/s	0.206 m/s

Table 29. max transfer speeds per payload size on wifi

4.3.5 Assessment with Security Configurations

As mentioned above, all the tests so far have been conducted with a basic level of security, using the defaults provided by each protocol and strong encryption keys. We can also study how much different security attributes affect the connections and whether changing the key sizes or authentication mechanisms has a visible effect in the observed final speed of the connection.

Like above, separate tests were conducted for MQTT, CoAP, HTTP/2 and HTTP/3. Three different configurations were tested with different modes of security, aiming at a high security level, a lower security level and finally a connection without any security features applied.

For MQTT, the high security configuration includes the use of TLS 1.3 with the TLS_AES_256_GCM_SHA384 cipher, client authentication through certificates and both clients and broker certificates using 4096-bit RSA keys. Access to topics is given to the clients through an access control list configured on the broker, using the identifiers in the client certificates.

For the lower security configuration, TLS 1.2 is requested by the client, with the weaker TLS_RSA_WITH_AES_128_GCM_SHA256 cipher and a shorter 2048-bit RSA key for the broker. The clients use username-password authentication which does not require a certificate but sends some extra data through the appropriate user fields of the requests. Access to topics is given to the clients through an access control list configured on the broker, using the username as an identifier of the client.

Finally the no security configuration does not apply any encryption to the connection, nor authentication for the clients and anyone is allowed access to all topics.

After executing some tests, it appears that the different security configurations do not have a big effect on the transfer speed. The high and low security levels show very similar results, even while using different protocols and key sizes. The no security configuration is a bit faster but not enough to have a visible effect in a real-world application.

Payload Size	High Security	Low Security	No Security
1 byte	9802.732 m/s	9868.822 m/s	10919.696 m/s
10 bytes	9855.493 m/s	10196.775 m/s	10692.769 m/s
100 bytes	9798.191 m/s	9796.615 m/s	11133.855 m/s
1000 bytes	9265.471 m/s	9360.552 m/s	10045.032 m/s
10000 bytes	6188.022 m/s	6019.881 m/s	6103.334 m/s
100000 bytes	2873.458 m/s	2897.456 m/s	1953.625 m/s
1000000 bytes	785.130 m/s	312.094 m/s	331.078 m/s
10000000 bytes	3.235 m/s	3.167 m/s	3.410 m/s

Table 30. MQTT max transfer speeds on different security levels

The changes in speed throughout the transfer are also fairly similar between different modes for the same payload size. The modes using encryption do show some bigger drops at times. These are more frequent for the low security mode, which may be explained by TLS 1.2 generally being considered less efficient than TLS 1.3.

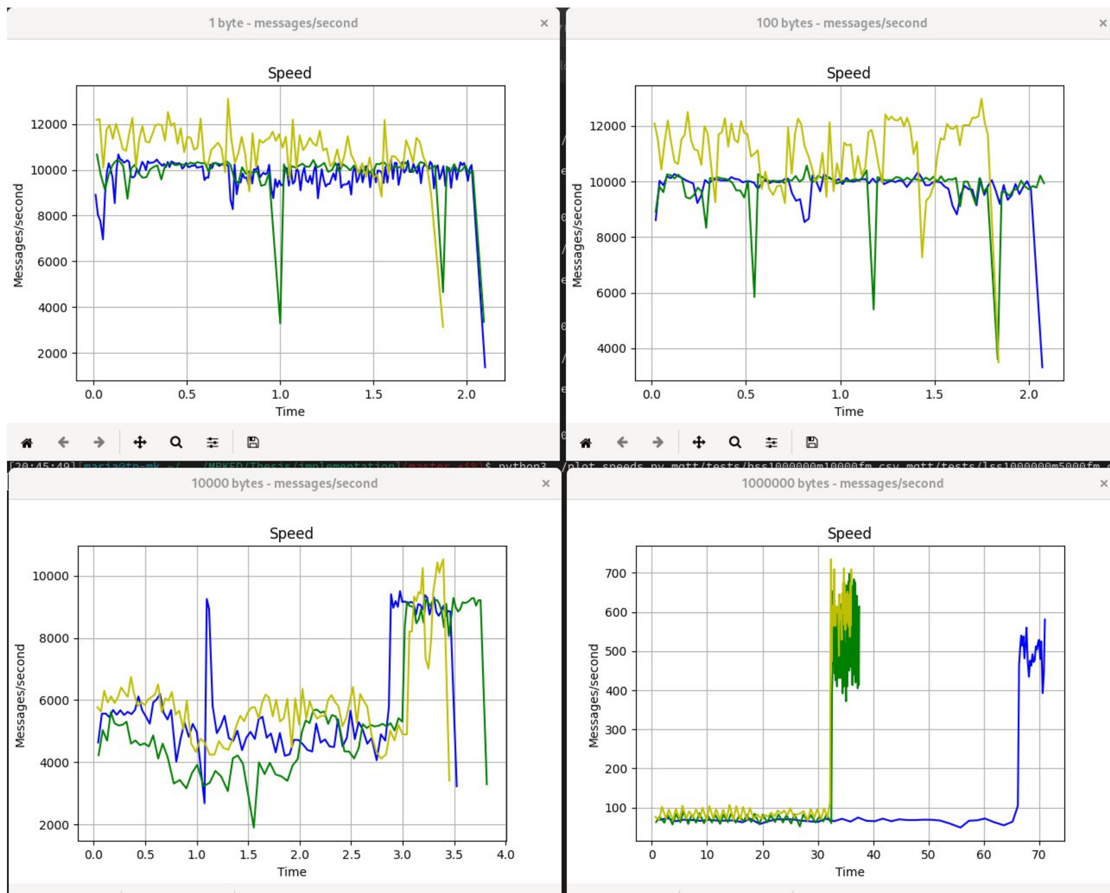

Figure 33. MQTT max speeds in time per payload size (high security blue, low security green, no security yellow)

Overall, different security configurations in MQTT affect the performance of the connection in a small way.

For CoAP, the utilized library doesn't provide much configuration when it comes to security. The protocol itself supports DTLS with pre-shared keys, raw public key certificates or X.509 certificates but the current implementation of DTLS is limited to PSKs. The OSCORE extension is also supported, which provides end-to-end encryption of the message content for use with proxies that may require access to the packet properties for correct routing.

Therefore three modes were selected for this test, the first using the default configuration of DTLS (DTLS 1.2 with a PSK), the second utilizing OSCORE but without DTLS and the third without any encryption. The PSK selected was a 10-character word.

In these results, like before there isn't a great difference in the performance of the connections, however DTLS appears very efficient and even faster than the no security configuration on average, possibly due to maintaining an open connection. OSCORE makes a bigger difference, being the most inefficient.

Payload Size	DTLS	OSCORE	No Security
1 byte	3507.193 m/s	2115.931 m/s	2574.217 m/s
10 bytes	3144.243 m/s	1956.630 m/s	2426.122 m/s
100 bytes	3357.649 m/s	2101.629 m/s	2350.175 m/s
1000 bytes	3279.234 m/s	2177.636 m/s	2592.119 m/s

Table 31. CoAP max transfer speeds on different security levels

Looking at the progression during the transfer, all connections start a bit slower and then pick up speed. This generally happens when the server has finished sending the requests and the client can now receive them faster. The DTLS mode starts faster from the beginning and raises the transfer much faster than the other modes, showing that it's not causing a big overhead to the server, while OSCORE is heavier on the server as it takes longer to finish sending the messages.

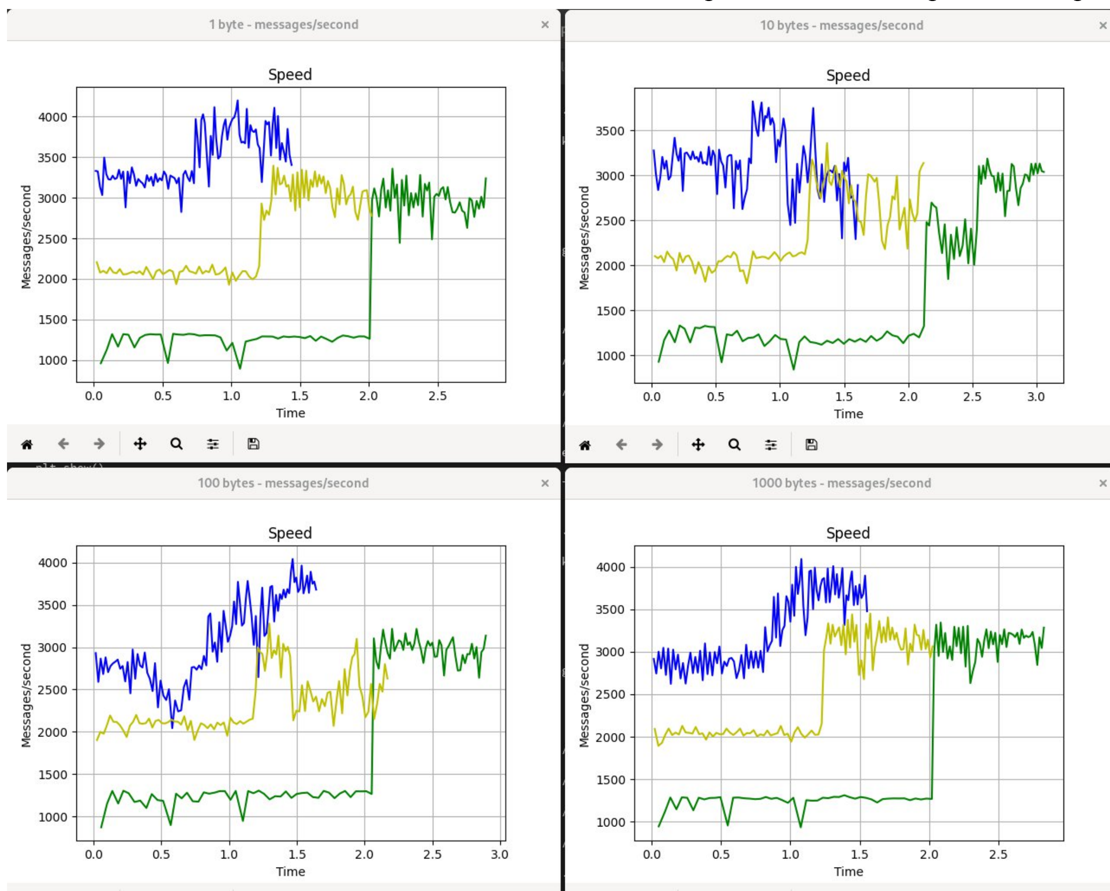

Figure 34. CoAP max speeds in time per payload size (DTLS blue, OSCORE green, no security yellow)

Overall, DTLS is very efficient in CoAP, while OSCORE affects performance in a negative way.

HTTP/2 also provides TLS for encrypted connections and several authentication mechanisms that can be customly applied through code. Here, again, three modes of security will be tested with similar features to the MQTT tests.

The high security mode includes the use of TLS 1.3 with the TLS_AES_256_GCM_SHA384 cipher and a server certificate with a 4096-bit RSA key.

The low security mode uses TLS 1.2 with the TLS_RSA_WITH_AES_128_GCM_SHA256 cipher and a server certificate with a shorter 2048-bit RSA key and no authentication for the clients.

Finally the no security mode does not apply any encryption or authentication to the connection.

After conducting some tests it is clear that the high and low security modes have a very similar performance. The low security mode is marginally faster but in a negligible way. The no security mode actually proved less efficient. It is possible that is due to the way TLS connections are handled by the machine, possibly keeping a persistent connection. Part of the initial HTTP/2 negotiation is also included in the TLS handshake, making up for any delays due to the TLS handshake as HTTP/2 is designed to be used with HTTPS.

Payload Size	High Security	Low Security	No Security
1 byte	21.220 m/s	21.577 m/s	18.867 m/s
10 bytes	21.228 m/s	21.638 m/s	19.083 m/s
100 bytes	21.293 m/s	21.851 m/s	18.912 m/s
1000 bytes	21.291 m/s	21.610 m/s	18.951 m/s
10000 bytes	21.234 m/s	21.521 m/s	18.819 m/s
100000 bytes	19.731 m/s	19.876 m/s	17.092 m/s
1000000 bytes	17.995 m/s	18.057 m/s	16.708 m/s
10000000 bytes	4.530 m/s	4.689 m/s	6.293 m/s

Table 32. HTTP/2 max transfer speeds on different security levels

Looking at the progress throughout the transfer, all three connections seem to follow the same pattern of a fast start that then becomes stable, just at different speeds.

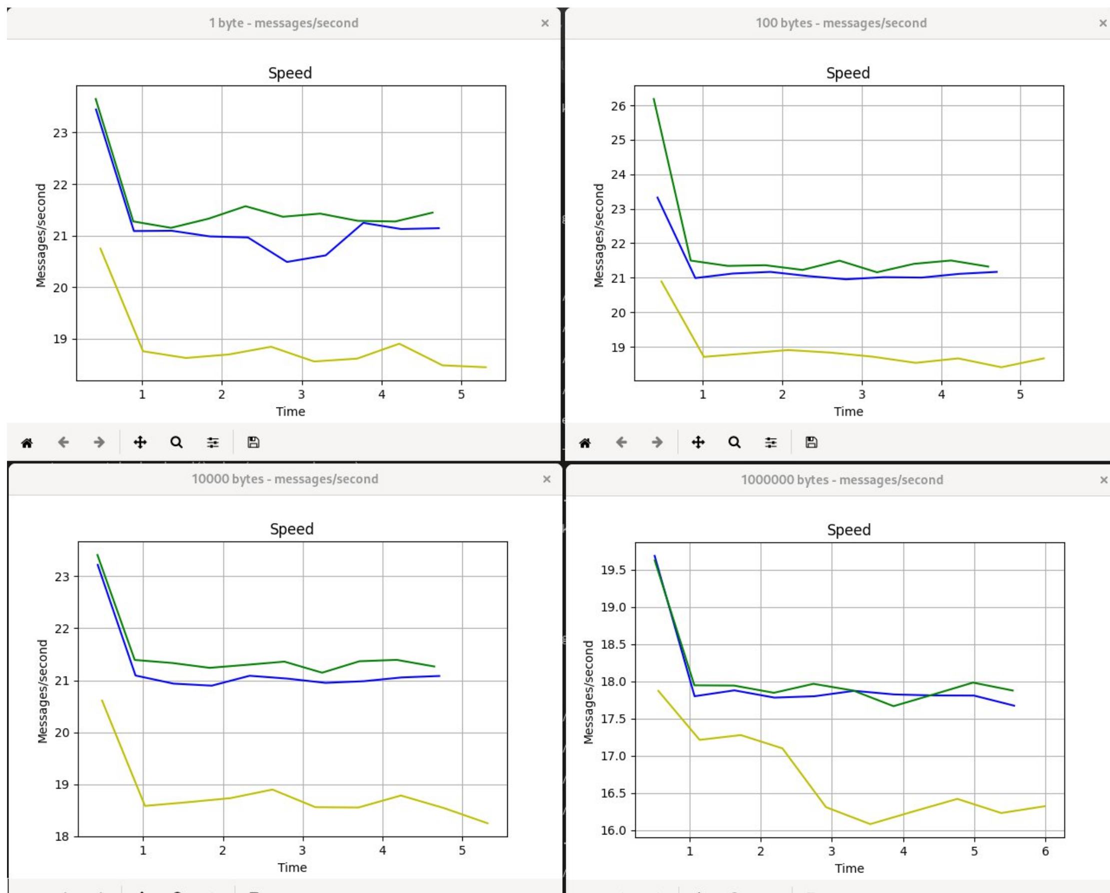

Figure 35. HTTP/2 max speeds in time per payload size (high security blue, low security green, no security yellow)

Finally, for HTTP/3 there is no option to remove encryption as it is required by the protocol. TLS 1.3 is also included as a requirement in QUIC, making it the most secure protocol so far. As HTTP/3 support on the server is experimental there aren't many configurations that can be made to the underlying QUIC connection. In this case two modes were tested.

The high security mode uses the default TLS 1.3 connection of QUIC, with the TLS_AES_256_GCM_SHA384 cipher, a 4096-bit RSA key for the server certificate and Basic Authentication for the clients.

The low security mode uses the same encryption configuration but with a shorter 2048-bit RSA key and no authentication for the clients.

In this case the low security configuration is slightly faster than the high security but mostly in smaller payloads and the difference is again small enough that it would probably not be detectable in a real-life scenario. This shows that the key size does not greatly affect the connection performance while the authentication parameters are only a few bytes in request headers which should not slow down the connection in any visible way.

Payload Size	High Security	Low Security
1 byte	538.403 m/s	629.023 m/s
10 bytes	524.310 m/s	503.902 m/s
100 bytes	546.044 m/s	563.509 m/s
1000 bytes	541.800 m/s	529.499 m/s
10000 bytes	393.524 m/s	342.990 m/s
100000 bytes	96.166 m/s	71.111 m/s
1000000 bytes	8.378 m/s	8.002 m/s
10000000 bytes	0.661 m/s	0.768 m/s

Table 33. HTTP/3 max transfer speeds on different security levels

Looking at the progress throughout the transfer, both types of connections seem very unstable, with the high security mode being slightly more stable. For larger payloads they are slow enough that not much data can be gathered. In any case the two configurations behave in a similar way.

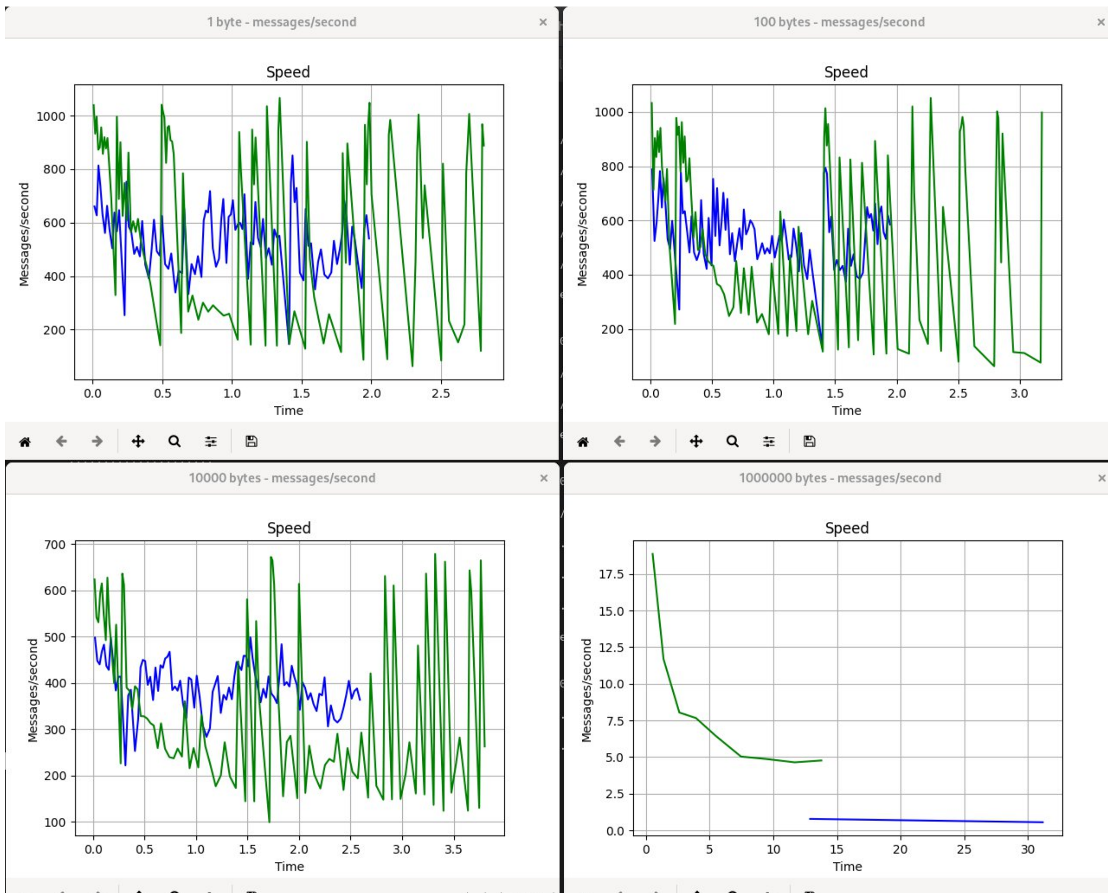

Figure 36. HTTP/3 max speeds in time per payload size (high security blue, low security green)

Overall for HTTP, higher security configurations are worthwhile as they do not negatively affect the performance of the connection.

4.4 Comparison

Based on the implemented setup we can compare the protocols on their features in practice. On the security scope, integrity, confidentiality and availability are considered, through the use of encryption, authentication and authorization, and availability tests conducted. On the functional scope, ease of setup and observed support for each protocol is considered, along with the amount of resources they consume and number of features that are made available to implementations.

As the implementation was made in Python, only Python libraries were studied during the setup process and some of the metrics regarding protocol support may be different for other languages.

4.4.1 Security Scope

On the security scope, all protocols offer encryption, with HTTP/3 enforcing TLS 1.3 with the newest algorithms. While it is optional in the other protocols, TLS 1.3 is also supported, except in the case of CoAP which uses DTLS. Key length and use of different ciphers does not seem to affect much the performance of the connection so opting for the more secure options is probably better. Authentication and authorization may depend on the tools used but both MQTT and HTTP versions offer multiple mechanisms. CoAP is more susceptible to availability issues, while the others have generally good performance on larger payloads and can configure limits and protections on their servers. For MQTT a small overhead may be added by the TLS handshake but it shouldn't be noticeable in real use cases.

In general HTTP versions offer more customization on their security setups while MQTT can be good for IoT setups, especially when looking for availability in constrained environments. CoAP can be useful in simpler setups where only smaller payloads are expected and authorization can be managed through more static keys and certificates.

	MQTT	CoAP	HTTP/2	HTTP/3
Encryption	Optional TLS Supports newest algorithms Good support Small effect on performance	Optional DTLS Extra layer for proxies (OSCORE) Limited support Does not lower performance	Optional TLS Supports newest algorithms Good support Does not lower performance	TLS 1.3 required Full forward secrecy by default Good support Does not lower performance
Authentication & Authorization	Username – password Certificates Other mechanisms depending on broker implementation	Custom mechanisms through DTLS keys or certificates	Basic Authentication JWT Cookies OAuth Other custom mechanisms	Basic Authentication JWT Cookies OAuth Other custom mechanisms
Availability	Faster transfer speeds Can handle large payloads Limits can be configured on the broker side	Slower transfer speeds Only smaller payloads Servers can have fewer resources when running on IoT nodes	Very slow transfer speeds Can handle large payloads Limits can be configured for servers	Slower transfer speeds but faster than HTTP/2 Connections are more robust due to multiple streams Can handle large payloads Limits can be configured for servers Different attack vectors through UDP

Table 34. Protocol comparison on security scope

4.4.2 Functional Scope

On the functional scope, it is relatively easy to set up a connection for all protocols but available resources and support vary, especially on the server side. MQTT and HTTP/2 seem to be the most supported protocols, with multiple tool options and the most feature support at the time of selecting libraries for the implementation. On the consumption of resources, MQTT and CoAP are generally lighter as they are designed for constrained environments, while HTTP protocols are focused on more generic uses and are generally heavier in resource and power consumption [10, 12]. Similarly, HTTP offers a greater set of features with more customization but many IoT-focused features should be available in MQTT and CoAP as well.

In general MQTT and HTTP/2 connections are the easiest to set up with the best support, while the features of MQTT and CoAP may be better suited for constrained IoT environments, with CoAP having the simplest and lightest structure.

	MQTT	CoAP	HTTP/2	HTTP/3
Setup	Easy when using off-the-shelf broker Many available resources	Simpler architecture Easy setup for clients Harder setup on server side due to less support	Easy with more options Can support different architectures More familiarity	More experimental Less available resources Can support different architectures Similar setup to HTTP/2
Support	Well supported Many tools and options	Not all features are supported by different libraries	Well supported Many tools and options	Supported, sometimes experimental support Fewer tool options than HTTP/2
Resources	Light clients Broker consumes more resources Low network bandwidth Works in low resource conditions	Light clients and servers Low network bandwidth Works in low resource conditions	Heavier on resources Depends on architecture	Heavier on resources Heavier than HTTP/2 on the server side Depends on architecture
Features	Good set of features Focused on IoT use cases	Smaller set of features Focused on IoT use cases	Many features Customizable More generic use cases	Many features Customizable More generic use cases

Table 35. Protocol comparison on functional scope

5 Conclusions and Future Work

5.1 Conclusions

In the end, each of the studied protocols can be used in applications in an IoT environment, bringing their unique characteristics. The selection of protocol should depend on the specific use-case of the application and the environment and conditions it's expected to work in.

One of the most important aspects is probably the architecture, where depending on requirements, devices of different capabilities will be set up in an IoT network, implementing different functions. In this aspect, CoAP and HTTP provide the most flexibility, by supporting different possible architectures. If a decentralized architecture is desired XMPP can provide that while MQTT provides the more common publish/subscribe architecture.

Then, the selection will depend on individual features, on the functionality level as well as on the security requirements. From this analysis MQTT checks most of the boxes for a usual IoT application as it provides the main functionality for clients to communicate with each other in a publish/subscribe architecture, plus some other useful features aimed at working with multiple clients. It can work in constrained environments and also supports a good level of security, with TLS encryption and authentication and authorization mechanisms, while being fairly resilient when working with many and big payloads. CoAP can cover simpler cases when transporting smaller payloads through networks of low-powered devices. It can be used in cases where security is not as important, where encryption can be used, with possible end-to-end encryption if proxies are used, but where authentication can be limited to pre-generated certificates instead of a more complex solution. HTTP provides more familiarity at development time and will allow for more custom solutions and integrations with other protocols but can be resource-heavy in an IoT environment and slower than the other protocols. It can also provide the most security features and be more robust, especially in setups where HTTP/3 is supported and used. Finally XMPP, while designed for instant messaging can be adapted to an IoT setup and provide a wide range of features through selected extensions, including end-to-end encryption and authorization mechanisms.

All in all, in use-cases where low resource use is a requirement, the lighter CoAP and MQTT protocols will be more fitting. But if transfer of larger payloads is expected and a more powerful device is available for the role of broker, MQTT may be a better option. MQTT and HTTP will also provide better features when working with more complicated authorization structures like access control lists. If integration with other services and protocols or more customization is required, HTTP can prove easier to work with but may need more configuration to fit the IoT use-case. When selecting HTTP, HTTP/3 will fit the IoT case more, by skipping the slow TCP handshake at the beginning of each transmission.

Based on the above, IoT setups where high performance is a requirement along with strong security, such as health-care applications, will benefit more from MQTT. On the other hand, environments that do not have complicated security requirements, for example because communication is executed in a trusted network without unknown nodes, could also utilize CoAP. An example would be smart-home applications where payloads are usually smaller. Finally if speed is not as important but interoperability and customizations might be needed, such as in industrial applications which also require a good level of security, HTTP/3 is a good candidate as long as it is supported by the relevant devices.

Of course the final thing to take into account is the state of adoption and support at the time of development of the IoT solution. During the experimental implementation not all protocols proved to have the same level of support, with some features not being available at this time. At the time of writing, in Python, MQTT and HTTP/2 are fully supported with multiple tools and libraries to choose from. HTTP/3 is available on an experimental level with a more limited selection of tools and may be lucking support in some setups. CoAP is generally supported but without all features being

implemented. Adoption may differ when using other programming languages so it is important to research before making a final decision.

5.2 Future Work

When developing an IoT solution, the selection of communication protocol is an important step. This selection depends on the specific needs of the application as different protocols operate under different architectures and provide different features. In this work we presented some of the most common options for protocols on the application level, with a detailed analysis of their features and architecture, following by an experimental implementation to get a better idea of how they work in practice. A final comparison was presented, highlighting the conditions and use cases where each protocol will prove most useful, trying to aid engineers in making the right selection for their application. As a next step, more tests can be made under constrained conditions, such as running on low-powered devices or limited networks and deploying the client software to a larger number of devices, communicating with the same central points, simulating a big IoT network that may be constructed as part of a real-life application.

6 Bibliography

1. *Asghari, P., Rahmani, A.M., & Javadi, H.H. (2019). Internet of Things applications: A systematic review. Comput. Networks, 148, 241-261.*
2. *Bayilmis, C., Ebleme, M.A., Çavuşoğlu, Ü., Küçük, K., & Sevin, A. (2022). A survey on communication protocols and performance evaluations for Internet of Things. Digit. Commun. Networks, 8, 1094-1104.*
3. *Al-Fuqaha, A., Guizani, M., Mohammadi, M., Aledhari, M., & Ayyash, M. (2020). Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications. IEEE Communications Surveys & Tutorials, 17(4), 2347–2376.*
4. *Reddy, S.M., & Krishna.A, V. (2017). PERFORMANCE ANALYSIS OF DATA PROTOCOLS OF INTERNET OF THINGS: A QUALITATIVE REVIEW.*
5. *Karagiannis, V., Vázquez-Gallego, F., Chatzimisios, P., & Alonso-Zarate, J. (2015). A Survey on Application Layer Protocols for the Internet of Things.*
6. *Silva, D.M., Carvalho, L., Soares, J.A., & Sofia, R.C. (2021). A Performance Analysis of Internet of Things Networking Protocols: Evaluating MQTT, CoAP, OPC UA. Applied Sciences, 11, 4879.*
7. *Sethi, P., & Sarangi, S.R. (2017). Internet of Things: Architectures, Protocols, and Applications. J. Electr. Comput. Eng., 2017, 9324035:1-9324035:25.*
8. *Burhan, M., Rehman, R.A., Khan, B.M., & Kim, B. (2018). IoT Elements, Layered Architectures and Security Issues: A Comprehensive Survey. Sensors (Basel, Switzerland), 18.*
9. *Xing, L. (2020). Reliability in Internet of Things: Current Status and Future Perspectives. IEEE Internet of Things Journal, 7, 6704-6721.*
10. *Gemirter, C. B., Şenturca, Ç., & Baydere, Ş. (2021, September 1). A Comparative Evaluation of AMQP, MQTT and HTTP Protocols Using Real-Time Public Smart City Data. IEEE Xplore. https://doi.org/10.1109/UBMK52708.2021.9559032*
11. *Koutras, D., Stergiopoulos, G., Dasaklis, T.K., Kotzanikolaou, P., Glynos, D., & Douligeris, C. (2020). Security in IoMT Communications: A Survey. Sensors (Basel, Switzerland), 20.*
12. *Ochoa, H.J., Peña, R., Mezquita, Y.L., Gonzalez, E., & Camacho-Léon, S. (2023). Comparative Analysis of Power Consumption between MQTT and HTTP Protocols in an IoT Platform Designed and Implemented for Remote Real-Time Monitoring of Long-Term Cold Chain Transport Operations. Sensors (Basel, Switzerland), 23.*
13. *Srivastava, N., & Pandey, P. (2022). Internet of things (IoT): Applications, trends, issues and challenges. Materials Today: Proceedings.*
14. *Domínguez-Bolaño, T., Fernández, O.C., Barral, V., Escudero, C.J., & García-Naya, J.A. (2022). An overview of IoT architectures, technologies, and existing open-source projects. Internet Things, 20, 100626.*
15. *Colitti, W., Steenhaut, K., Caro, N.D., Buta, B., & Dobrota, V. (2011). Evaluation of constrained application protocol for wireless sensor networks. 2011 18th IEEE Workshop on Local & Metropolitan Area Networks (LANMAN), 1-6.*
16. *Perna, G., Trevisan, M., Giordano, D., & Drago, I. (2022). A first look at HTTP/3 adoption and performance. Comput. Commun., 187, 115-124.*

17. *MQTT. (2022). MQTT - The Standard for IoT Messaging. Mqtt.org. https://mqtt.org*
18. *MQTT Version 5.0. (n.d.). Docs.oasis-Open.org. https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html*
19. *CoAP — Constrained Application Protocol | Overview. (n.d.). Coap.space. http://coap.space*
20. *RFC 7252 - The Constrained Application Protocol (CoAP). (2014). IETF Datatracker. https://datatracker.ietf.org/doc/html/rfc7252*
21. *RFC 6690 - Constrained RESTful Environments (CoRE) Link Format. (2012) IETF Datatracker. https://datatracker.ietf.org/doc/html/rfc6690*
22. *RFC 7959 - Block-Wise Transfers in the Constrained Application Protocol (CoAP). (2016). IETF Datatracker. https://datatracker.ietf.org/doc/html/rfc7959*
23. *RFC 7641 - Observing Resources in the Constrained Application Protocol (CoAP). (2015). IETF Datatracker. https://datatracker.ietf.org/doc/html/rfc7641*
24. *RFC 8613 - Object Security for Constrained RESTful Environments (OSCORE). (2019). IETF Datatracker. https://datatracker.ietf.org/doc/html/rfc8613*
25. *MDN Contributors. (2019, August 3). HTTP. MDN Web Docs. https://developer.mozilla.org/en-US/docs/Web/HTTP*
26. *"http3" | Can I use... Support tables for HTML5, CSS3, etc. (2024). Caniuse.com. https://caniuse.com/?search=http3*
27. *RFC 9110 - HTTP Semantics. (2022). IETF Datatracker. https://datatracker.ietf.org/doc/html/rfc9110*
28. *XMPP | The universal messaging standard. (n.d.). Xmpp.org. https://xmpp.org*
29. *Saint-André P., Smith, K., & Remko Troncon. (2009). XMPP : the definitive guide. O'reilly.*
30. *RFC 6120 - Extensible Messaging and Presence Protocol (XMPP): Core. (2011). IETF Datatracker. https://datatracker.ietf.org/doc/html/rfc6120*
31. *RFC 6121 - Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence. (2011). IETF Datatracker. https://datatracker.ietf.org/doc/html/rfc6121*
32. *RFC 7622 - Extensible Messaging and Presence Protocol (XMPP): Address Format. (2015). IETF Datatracker. https://datatracker.ietf.org/doc/html/rfc7622*
33. *XEP-0384: OMEMO Encryption. (2022). Xmpp.org. https://xmpp.org/extensions/xep-0384.html*
34. *XEP-0060: Publish-Subscribe. (2021). Xmpp.org. https://xmpp.org/extensions/xep-0060.html*
35. *XEP-0198: Stream Management. (2022). Xmpp.org. https://xmpp.org/extensions/xep-0198.html*
36. *XEP-0322: Efficient XML Interchange (EXI) Format. (2018). Xmpp.org. https://xmpp.org/extensions/xep-0322.html*