technologies

MDPI

Article

A Lightweight Messaging Protocol for Internet of Things Devices

Justice Owusu Agyemang [1,*], Jerry John Kponyo [1], James Dzisi Gadze [1], Henry Nunoo-Mensah [1] and Dantong Yu [2]

[1] Faculty of Electrical/Computer Engineering, Kwame Nkrumah University of Science and Technology, Kumasi AK-509-4752, Ghana; jjkponyo@ieee.org (J.J.K.); jdgadze@gmail.com (J.D.G.); hnunoo-mensah@knust.edu.gh (H.N.-M.)
[2] New Jersey Institute of Technology, Newark, NJ 07102, USA; dtyu@njit.edu
* Correspondence: justiceowusuagyemang@gmail.com

Abstract: The move towards intelligent systems has led to the evolution of IoT. This technological leap has over the past few years introduced significant improvements to various aspects of the human environment, such as health, commerce, transport, etc. IoT is data-centric; hence, it is required that the underlying protocols are scalable and sufficient to support the vast D2D communication. Several application layer protocols are being used for M2M communication protocols such as CoAP, MQTT, etc. Even though these messaging protocols have been designed for M2M communication, they are still not optimal for communications where message size and overhead are of much concern. This research paper presents a Lightweight Messaging Protocol (LiMP), which is a minified version of CoAP. We present a detailed protocol stack of the proposed messaging protocol and also perform a benchmark analysis of the protocol on some IoT devices. The proposed minified protocol achieves minimal overhead (a header size of 2 bytes) and has faster point-to-point communication from the benchmark analysis; for communication over LAN, the LiMP-TCP outperformed the CoAP-TCP by an average of 21% whereas that of LiMP-UDP was over 37%. For a device to remote server communication, LiMP outperformed CoAP by an average of 15%.

Keywords: messaging protocol; Internet of Things; CoAP

Citation: Agyemang, J.O.; Kponyo, J.J.; Gadze, J.D.; Nunoo-Mensah, H.; Yu, D. A Lightweight Messaging Protocol for Internet of Things Devices. *Technologies* **2022**, *10*, 21. https://doi.org/10.3390/technologies10010021

Academic Editors: Yury A. Skorik and Vijayakumar Varadarajan

Received: 16 December 2021
Accepted: 27 January 2022
Published: 29 January 2022

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Copyright: © 2022 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

There has been a significant advancement towards 'smart' and 'smarter' systems due to the integration of smart objects into the existing and new infrastructure of today's data-intensive applications [1,2]. This move has led to the evolution of IoT which is currently driving sectors such as agriculture, manufacturing, smart healthcare, etc. [3–6]. IoT devices range from simple wearable to large machines, each containing sensor chips or microcontrollers [7]. These devices can occur in physical or virtual space. In physical space, consider humans, vehicles, residences, computers, switches, routers, smart devices, road networks, office buildings, etc. In virtual space, consider software, data streams, virtual machines, virtual networks, etc. [8].

The building blocks of IoT include: Sensor, Aggregator, Communication channel, eUtility, and Decision Trigger. Sensors measure physical properties by employing mechanical, electrical, chemical, optical, or other effects at an interface to a controlled process or open environment. An aggregator is a software implementation based on mathematical function(s) that transforms groups of *raw* data into *intermediate*, *aggregated* data. Aggregators help in managing 'big' data. A communication channel is a medium by which data is transmitted (e.g., wireless, wired, etc.). An eUtility is an abstraction of unforeseen future services that can be incorporated in future types of IoTs yet to be defined. It may include databases, mobile devices, software or hardware systems, etc. A decision trigger creates the final result(s) needed to satisfy the purpose, specification, and requirements of a specific IoT device.

Most IoT devices follow a layered architecture comprising of three or four layers [9–16]; as shown in Figure 1.

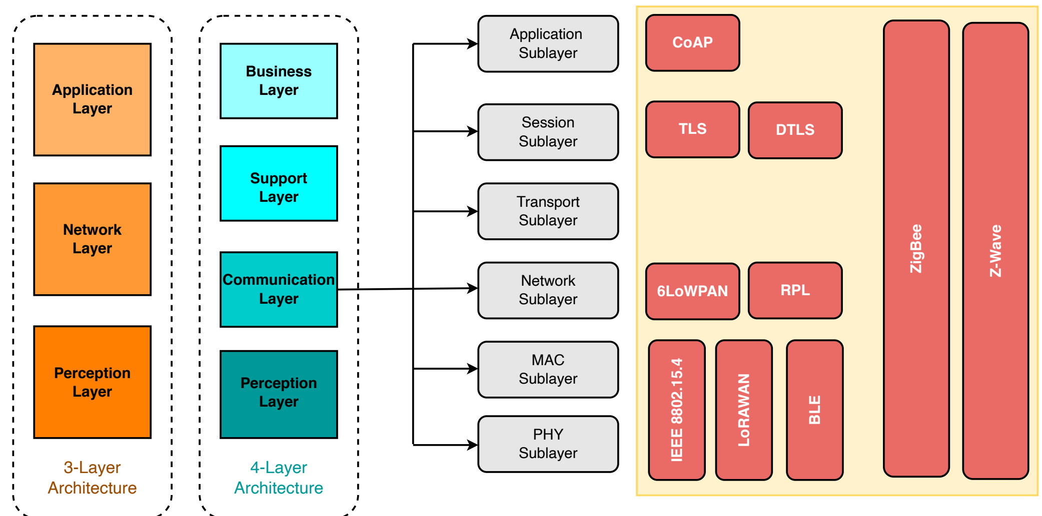

Figure 1. Most common IoT architectures [10,17].

The perception layer comprises the physical level of objects and how they interact with the surrounding environment by collecting and processing information [17]. This level includes objects that can interact with the external world and are also equipped with computing capability. The network layer, also known as the communication layer, transports the data provided by the perception layer to the application layer. This layer can be broken down into six (6) sublayers, viz., PHY, MAC, transport, network, session, and application. It includes all the technologies and protocols that make this connection possible. There are several protocols used by IoT devices [1,2]. The choice of the communication protocol is dependent on the type of technology been used; Zigbee, BLE, NFC, Z-Wave, LPWAN etc. [18]. Some network layer protocols have been developed for low computational devices of which IoT devices form part. For instance, to meet the requirements of WSN, the 6LoWPAN protocol and other routing protocols such as RPL were created. Also, to enhance the security of traversing data, TLS (for TCP) and DTLS (for UDP) were developed. Application layer messaging protocols such as CoAP, MQTT, AMQP, etc have also been developed and tailored specifically for M2M type of communication. The support layer enhances the operation of the other layers, providing storage and computing services. The application layer includes all the software necessary to offer a specific service. The data from the previous levels are stored, aggregated, filtered, processed, and later used in making informed decisions or used in the provision of real-time IoT applications.

Information sharing among IoT devices is made possible by the use of application layer messaging protocols such as CoAP, WebSocket, DDS, XMPP and AMQP. These protocols are not optimal and efficient in instances where header and message size complexities, together with resource constraints, are of great concern. Besides, with the myriad growth of IoT devices, it is expected that these messaging protocols are optimal and efficient in supporting D2D communication. This research paper proposes a lightweight messaging protocol to reduce message header complexity without compromising the security of the protocol. The remaining sections of the research paper are organized as follows: Section 2 reviews related literature. The research motivation is presented in Section 3. Section 4 describes the research approach used in the study. Section 5 discusses the results. Section 6 concludes the research paper.

2. Related Works

The most common application layer messaging protocols used by IoT devices include: CoAP, WebSocket, DDS, XMPP and AMQP.

CoAP is a specialized web transfer protocol for use with constrained nodes and constrained networks. The work on CoRE aims at realizing the REST architecture in a suitable form for constrained nodes. Constrained nodes such as 6LoWPAN support the fragmentation of IPv6 into small link-layer frames; however, this causes a significant reduction in packet delivery probability. CoAP has been designed to keep the message overhead small, thus limiting the need for fragmentation [19]. It supports both UDP [20] and TCP transport protocols with the default being UDP. It has optional reliability supporting both unicast and multicast requests. It supports asynchronous message exchanges and also has simple proxy and caching capabilities.

The WebSocket protocol was developed to address a high overhead due to HTTP polling. Bidirectional communication between a client and a server has required an abuse of HTTP to poll the server for updates while sending upstream notifications as distinct HTTP calls [21]. This results in problems such as the server being forced to use several different underlying TCP connections for each client, one for sending information to the client and a new one for each incoming message. Besides, the wire protocol has a high overhead, with each client-to-server messaging having an HTTP header. Furthermore, the client-side script is forced to maintain a mapping from the outgoing connections to the incoming connections to track replies. The WebSocket protocol addresses these problems by using a single TCP connection for traffic in both directions. Combined with WebSocket API, it provides an alternative to HTTP polling for two-way communication between a client, and a server [22].

Some IoT device communication is based on a publish/subscribe protocol such as MQTT. MQTT is a client-server publish/subscribe messaging transport protocol applicable in constrained environments for communication in M2M and IoT contexts. The protocol runs over TCP/IP, or over other network protocols that provide ordered, lossless, bi-directional connections. The use of the publish/subscribe message pattern provides one-to-many message distribution and decoupling of applications [23].

XMPP is also used for the near-real-time exchange of information. It is an application profile of the XML that enables the exchange of structured yet extensible data (called "XML stanzas") between any two or more network entities [24,25]; based on TCP. It is typically implemented using a distributed client-server architecture, wherein a client needs to connect to a server in order to gain access to the network and thus be allowed to exchange XML stanzas with other entities (which can be associated with other servers). Within XMPP, one server can optionally connect to another server to enable inter-domain or inter-server communication after the communicating servers negotiate a connection between themselves.

DDS is a middleware protocol and API for data-centric connectivity. It integrates the components of a system, provides low-latency data connectivity, extreme reliability, and scalable architecture that business and mission-critical IoT applications need. The middleware is a software layer that lies between the operating system and applications, as shown in Figure 2. It enables the various components of the system to communicate and share data more easily. It abstracts the application for the details of the operating system, network transport, and low-level data formats. Low-level details like data wire format, discovery, connections, reliability, protocols, transport selection, QoS, security, etc., are managed by the middleware.

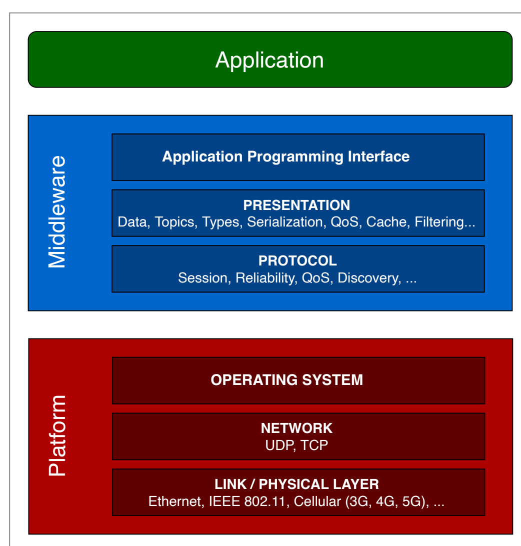

Figure 2. The DDS Middleware [26].

AMQP is a binary wire-level protocol that allows the reliable exchange of messages between two entities. It is a corporate messaging protocol designed for reliability, security, provisioning, and interoperability and supports both request/response and publish/subscribe architectures. The protocol also offers a wide range of features related to messaging, such as reliable queueing, topic-based publish-and-subscribe messaging, flexible routing, and transactions. AMQP communication system requires that either the publisher or consumer creates an "exchange" with a given name and then broadcasts that name. Publishers and consumers use the name of this exchange to discover each other. Messages are exchanged in various ways: directly, by topic, or based on headers [27].

A summary of the mostly used application layer messaging protocols is shown in Table 1.

Table 1. Comparative Analysis of Application Layer Messaging Protocols for IoT Devices: CoAP, WebSocket, MQTT, DDS, XMPP, AMQP.

Criteria	CoAP	WebSocket	MQTT	DDS	XMPP	AMQP
Architecture	Client/Server or Client Broker	Client/Server or Client/Broker	Client/Broker	Client/Server	Client/Server	Client/Broker or Client/Server
Header Size	4 Bytes	4 Bytes	2 Bytes	124 Bytes	Undefined	8 Bytes
Transport Protocol	UDP (default), TCP, SCTP	TCP	TCP, UDP (Some Implementations)	UDP, TCP, SCTP	TCP	TCP, SCTP
Encoding Format	Binary	Text (UTF-8)	Binary	Binary	Binary	XML

Performance of IoT devices and applications are significantly influenced by choice of messaging protocols. The application layer messaging protocols are pervasive and different from each other. For example, CoAP has the smallest message size and overhead as compared to the other messaging protocols. However, MQTT is lightweight and has the most diminutive header size of 2-bytes per message, but its requirement of TCP connection increases the overall overhead, and thus the whole message size. AMQP is also a lightweight binary protocol; however, its support for security, reliability, provisioning, and interoperability increases the overhead and message size. WebSocket requires the highest power resource than any other protocols, and then it decreases for the other protocols with CoAP requiring the lowest power and resource [27].

3. Research Motivation

The magnitude and rate at which IoT devices generate data are rapidly increasing. Given that IoT systems primarily depend on IoT devices for data aggregation and message

exchanges for the overall functioning of the system, the choice of which communication or messaging protocol to use for device interconnectivity is very significant [28].

Current research works on lightweight messaging protocols have explored compressing some transport protocols [29], as well as adopting some middleware approaches for multi-protocol translation [30,31]. These studies identify that CoAP generates much less traffic overhead compared to MQTT when message sizes are small, and the loss rate is equal to or less than twenty-five percent. A comparative study on MQTT and CoAP revealed that both protocols achieve 100% data transfer with minimal packet loss [32]. Furthermore, CoAP's data loss rate is low when handling smaller data sizes. This research work is aimed at proposing a minified version of CoAP with minimal header and message complexity. The research contributions of this paper are:

1. The proposal of a lightweight messaging protocol based on CoAP for D2D communication, and
2. The proposal of a mechanism to detect message spoofing with reduced header parsing complexity.

4. Methodology

This section presents LiMP, a lightweight messaging protocol, based on CoAP. We discuss CoAP in much details and also present how a lighter version was developed. CoAP deals with interchanges asynchronously over transport protocols such as UDP (default) and TCP. This is done logically using a layer of messages that supports optional reliability with exponential back-off. It defined four (4) types of messages: Confirmable, Non-confirmable, Acknowledgment, Reset.

Logically, CoAP can be considered as using a two-layer approach; a messaging layer and the asynchronous nature of the interactions (the request/response interactions using Method and Response Codes) as shown in Figure 3. The CoAP messaging model is based on the exchanges of messages over UDP/TCP between endpoints and may also be used over DTLS and TLS.

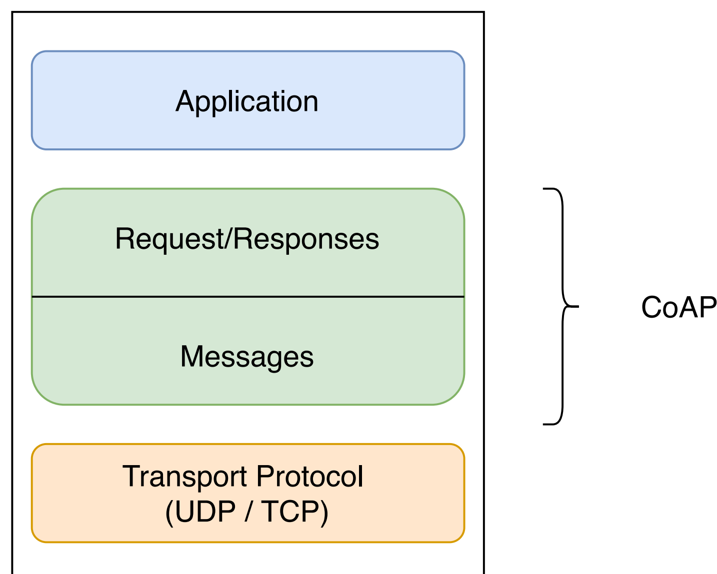

Figure 3. Abstract Layering of CoAP.

4.1. CoAP Stack

CoAP messages are transport over UDP by default (i.e., each CoAP message occupies the data section of one UDP datagram). CoAP messages are encoded in a simple binary format. The message format is shown in Figure 4.

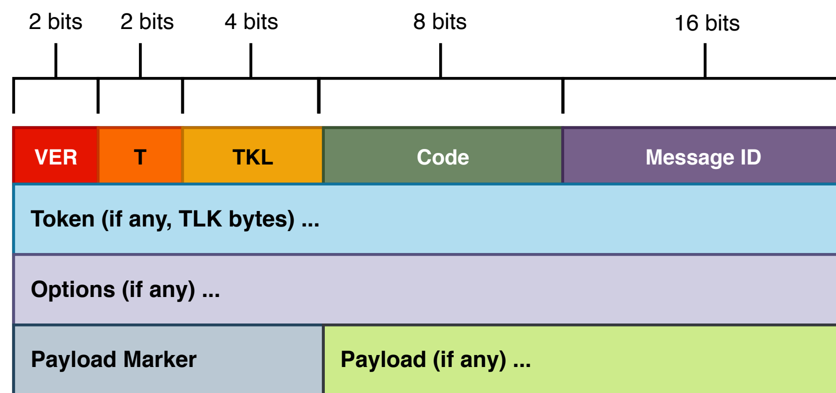

Figure 4. CoAP Message Format.

A detail representation of the various sections in shown in Figure 5.

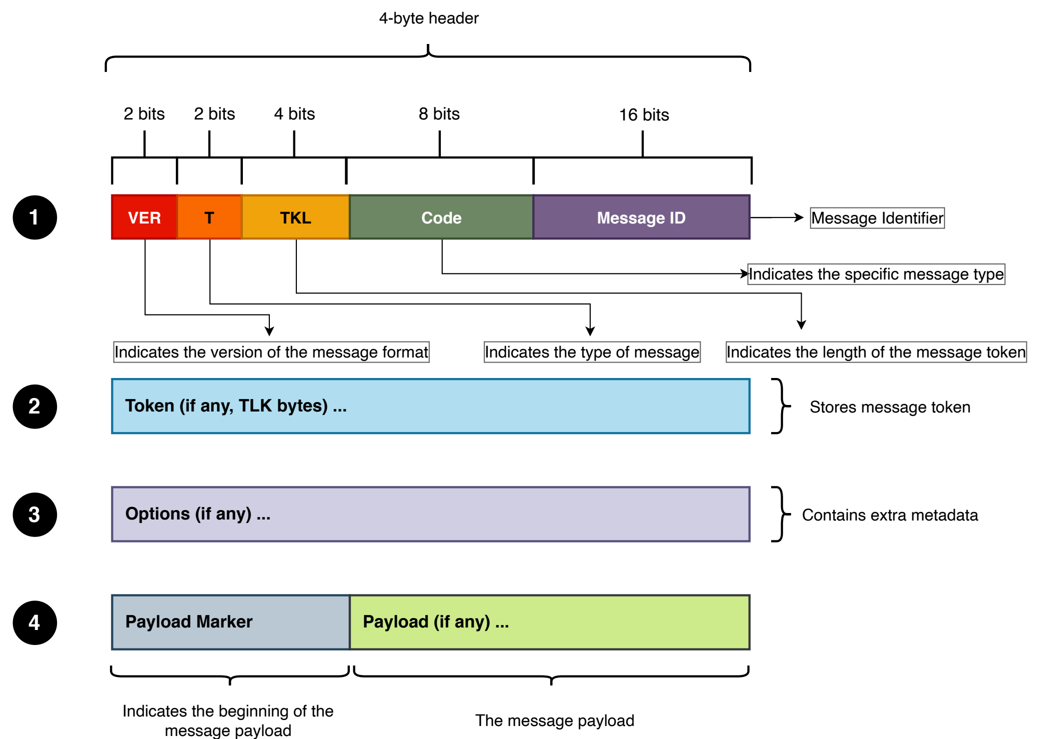

Figure 5. A Detailed Representation of the CoAP Message Format.

The message format starts with a fixed 4-byte header (indicated as (1) in Figure 5. The header consists of a the following fields: Version (**VER**), Type (**T**), Token Length (**TKL**), **Code** and a message identifier (**Message ID**). The 'VER' field indicates the CoAP version. The 'Type (T)' field indicates whether a particular message is confirmable, non-confirmable, is an acknowledgement or a reset (a retransmission). The 'Code' is designated for request/response codes. It is similar to HTTP request/response codes which are used to indicate client/server response formats. For instance, a successful request to a server will return a status code. This status code can be found in the 'Code' field of the response message. The 'Message ID' is an identifier for requests/response. It is used to match messages of type Acknowledgement/Reset to messages of type Confirmable/Non-confirmable. A summary of CoAP's header fields is shown in Table 2.

Table 2. A Summary of CoAP's Header Fields.

Field	Number of Bits	Details
Version (VER)	2-bit unsigned integer	Indicates the CoAP version number. The default value is set to 1 (01 in binary).
Type (T)	2-bit unsigned integer	Indicates if the message is of type Confirmable (0), Non-confirmable (1), Acknowledgement (2), or Reset (3).
Token Length (TKL)	4-bit unsigned integer	Indicates the length of the variable-length Token field (0-8) bytes. Lengths 9-15 are reserved and processed as a message format error.
Code	8-bit unsigned integer	Divided into 3-bit class (most significant bits) and 5-bit detail (least significant bits), which are used to indicate requests and responses format.
Message-ID	16-bit unsigned integer	It is used to detect message duplication and further match messages of type Acknowledgement/Reset to messages of Confirmable/Non-confirmable.

The header field is followed by a 'Token' field (marked as (2) in Figure 5) which is used to match responses to requests independently from the underlying messages. After the 'Token' field comes the 'Options' field (marked as (3) in Figure 5); which is made up of some metadata such as the message format, ETag, etc. A payload marker comes after the 'Options' field. This marks the beginning of the actual message payload; marked in Figure 5 as (4). The presence of a marker followed by a zero-length payload is processed as a message format error.

4.2. Analysis of the CoAP Stack

The current CoAP message format indicates the ability to add a **Token** (whose length is specified by the TKL) for each request and response. Also, the **Message-ID** is used to match each request to a response. Figure 6 shows a sample packet analysis output of a point-to-point communication between two IoT devices. It can be observed that a sample request can be sent without a Token which sets the TKL to 0 in the message header.

```
> Frame 1: 57 bytes on wire (456 bits), 57 bytes captured (456 bits)
> Ethernet II, Src: Apple_9e:3c:df (0c:4d:e9:9e:3c:df), Dst: Nvidia_8d:75:f3 (00:04:4b:8d:75:f3)
> Internet Protocol Version 4, Src: 10.10.0.3, Dst: 10.10.0.11
> User Datagram Protocol, Src Port: 55999, Dst Port: 5683
v Constrained Application Protocol, Confirmable, GET, MID:255
    01.. .... = Version: 1
    ..00 .... = Type: Confirmable (0)
    .... 0000 = Token Length: 0
    Code: GET (1)
    Message ID: 255
  v Opt Name: #1: Etag: 77 65 65 74 61 67
      Opt Desc: Type 4, Elective, Safe
      0100 .... = Opt Delta: 4
      .... 0110 = Opt Length: 6
      Etag: 776565746167
  v Opt Name: #2: Uri-Path: a
      Opt Desc: Type 11, Critical, Unsafe
      0111 .... = Opt Delta: 7
      .... 0001 = Opt Length: 1
      Uri-Path: a
  v Opt Name: #3: Max-age: 2
      Opt Desc: Type 14, Elective, Unsafe
      0011 .... = Opt Delta: 3
      .... 0001 = Opt Length: 1
      Max-age: 2
    [Uri-Path: /a]
```

Figure 6. A Sample CoAP Message Format.

Both the Token and Message-ID are redundant since both seem to perform the same role. Besides, the length of the Token payload is encoded as 4 bits in the header (which makes the maximum Token payload 15 bytes). Even though a maximum Token payload of 16 bytes can be encoded, the default specification of CoAP processes the TKL length from 9–15 bytes as a message format error.

Furthermore, Four (4) request and twenty-one (21) response Method Codes are supported in the CoAP message format; *GET*, *POST*, *PUT* and *DELETE*. All other Method Codes are unassigned. These response codes were borrowed from the HTTP request and response code formats. Not all these requests and response codes are used in the normal communication between network entities; hence can be simplified to make the protocol further lightweight. The 'Option' field in the CoAP's message is delta encoded and defines several options that be included in a message. Fourteen (14) default 'Option' formats are supported. A sample of these fields can be seen after the Message-ID in Figure 6. These 'Option' formats describe the structure of the message sent specifying fields such as Content-Format, ETag, Max-Age, etc. Most of these Option Formats are unused, which makes them redundant.

4.3. LiMP Stack

The LiMP message supports both TCP and UDP transport protocols. It is a simplistic protocol where messages are encoded in a simple binary format. Its simplicity is achieved by removing the redundant and unused fields in the standard CoAP implementation. The message format is shown in Figure 7.

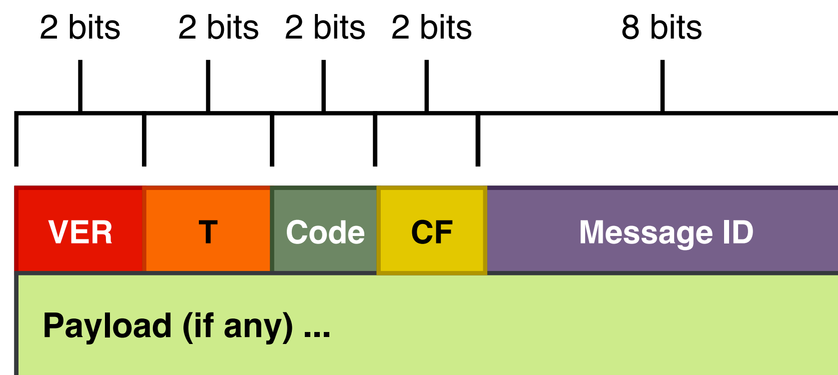

Figure 7. LiMP Message Format.

Figure 8 gives a detailed information of the message format. The message format starts with a fixed-size 2-byte header (marked as (1)). The message header comprises of the following fields: Version (**VER**), Type (**T**), **Code**, Content-Format (**CF**) and a message identifier (**Message ID**).

The header fields are described as follows:

1. **Version (VER):** VER is a 2-bit unsigned integer. Indicates the LiMP version number. The default value is set to 1 (01 in binary) to indicate its initial release.
2. **Type (T):** The T field is a 2-bit unsigned integer. Indicates of the message is of the type *Confirmable* (0), *Non-confirmable* (1), *Acknowledgement* (2), or *Reset* (2).
3. **Code:** This field stores a 2-bit unsigned integer. Indicates the request and response Method Codes. *GET* (0), *POST* (1), *BadRequest* (2), *ServiceUnavailable* (3). Detailed responses to requests can be included in the message payload.
4. **Content-Format (CF):** The CF is a 2-bit unsigned integer. It indicates the type of encoding of the payload contents. It was narrowed down to two(2) most common content-formats; *AppXML (XML format)*—0 and *AppJSON (JSON format)*—1.
5. **Message-ID:** The Message-ID portion is an 8-bit unsigned integer in network byte order. It is used to match requests to responses. This is 8-bits lesser than CoAP's implementation. To deal with request/response replays by an attacker or a malicious device, an ETag is included in the message payload to detect such attacks since 8-bit or 16-bit (in the case of CoAP) message-IDs are computationally easy for an attacker to compute. The ETag is a hexadecimal value resulting from a bitwise operation of the Message-ID and a current timestamp embedded in the message payload. Further details are provided in Section 5.1.

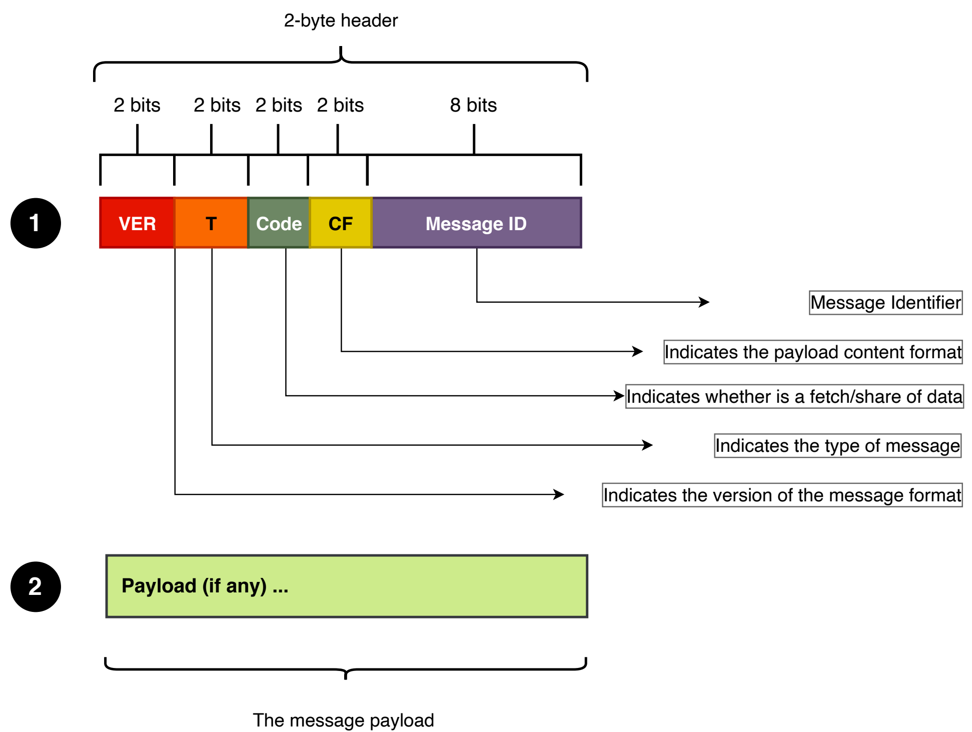

Figure 8. A Detailed Representation of the LiMP Message Format.

The header is followed by the message payload whose content format is indicated by the 'CF' field.

4.4. Benchmark Test and Analysis

A benchmark analysis of CoAP and LiMP was conducted on some embedded devices to evaluate the flexibility and efficiency of the proposed message protocol. A TCP and UDP versions of CoAP and LiMP were implemented and used in the benchmark test. The ARM devices used were: Nvidia Jetson Nano (**N**), Nvidia Tegra (**T**), Raspberry Pi (**P**) and and a Phone (**I**); as shown in Figure 9. ARM-based devices were selected because most IoT devices are of such architecture.

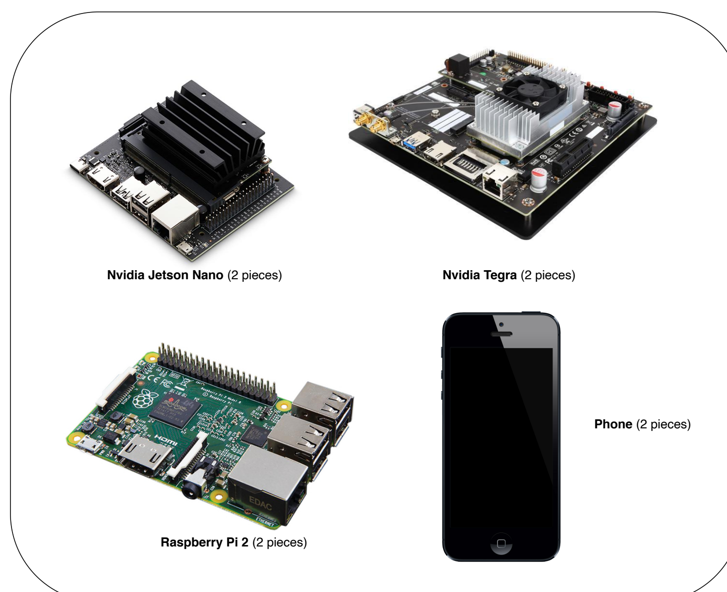

Figure 9. Benchmark Devices.

The Nvidia Jetson Nano and the Tegra have an ARM Cortex-A57 MPCore processor, the Raspberry Pi is an ARM Cortex-A53 processor, and the phone is an Apple A10 Fusion chipset. All the IoT devices except the phone were running an ARM-based Debian operating system, whereas the phone was running iOS. Two categories of the benchmark test was considered, (as shown in Figure 10) viz., Intra-Devices communication, and Out-of-domain communication. The intra-communication test was done on the same LAN and the out-of-domain communication test was done between the devices and a remote server (16 hops from the local network). The key performance indicators include:

1. The PDU sizes and
2. The RTT of requests/responses of the application layer messaging protocols

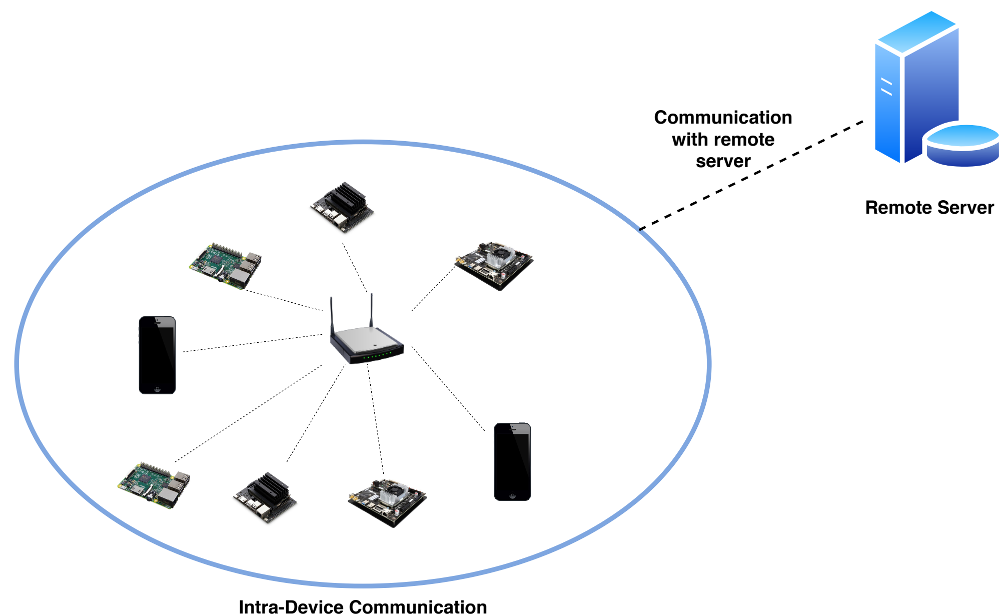

Figure 10. Setup for benchmark analysis.

The RTT is the measured time taken for an IoT device to send a request and receive a response. A total of 14 (*Point-to-Point Communication*) *x* 4 (*Protocols*) scenarios were used in the evaluation. A base implementation of CoAP and LiMP (written in *Golang*) was used in the evaluation analysis (Source code is available at: https://github.com/jayluxferro/levis accessed on 6 January 2022).

5. Results and Discussion

This section presents an analysis of the benchmark test and discusses the findings. Table 3 shows the default PDU sizes (in bytes) of LiMP and CoAP. The PDU size of LiMP-TCP is 16% smaller than the CoAP-TCP. Furthermore, the PDU size of LiMP-UDP is 23% smaller than the CoAP-UDP.

Table 3. Default PDU Size (in bytes): CoAP-TCP, LiMP-TCP, CoAP-UDP, LiMP-UDP.

	CoAP-TCP	LiMP-TCP	CoAP-UDP	LiMP-UDP
PDU (bytes)	81	68	57	44

Figure 11 shows the RTT for a point-to-point communication between two Jetson Nano devices. LiMP-TCP outperformed the CoAP-TCP. The LiMP-UDP outperformed all the other protocols; averaging 18 milliseconds (ms). This amounts to an efficiency of 17.758% more than CoAP.

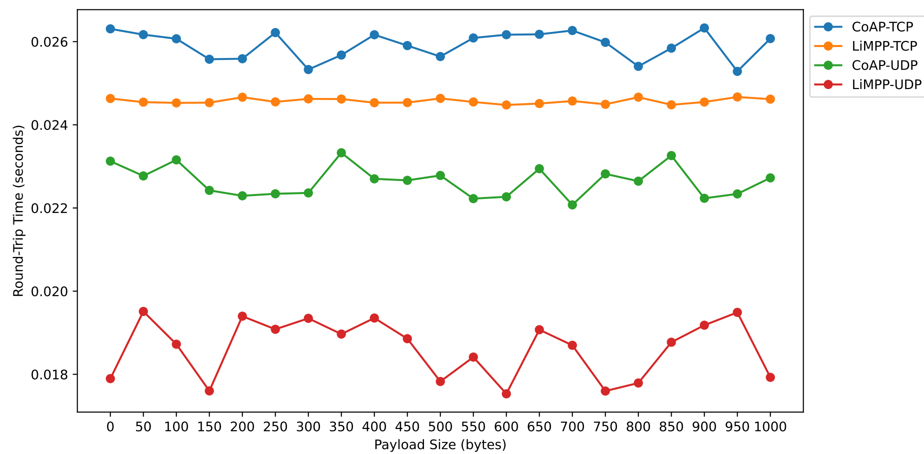

Figure 11. Point-to-Point Communication RTT between two Jetson Nano devices.

In Figure 12, the LiMP-UDP outperformed all the rest; averaging 5.7 ms. Similar observations were made in Figures 13–16. The efficiency of LiMP-TCP is 10.28% more than CoAP-TCP and that of LiMP-UDP is 18.04% more than CoAP-UDP.

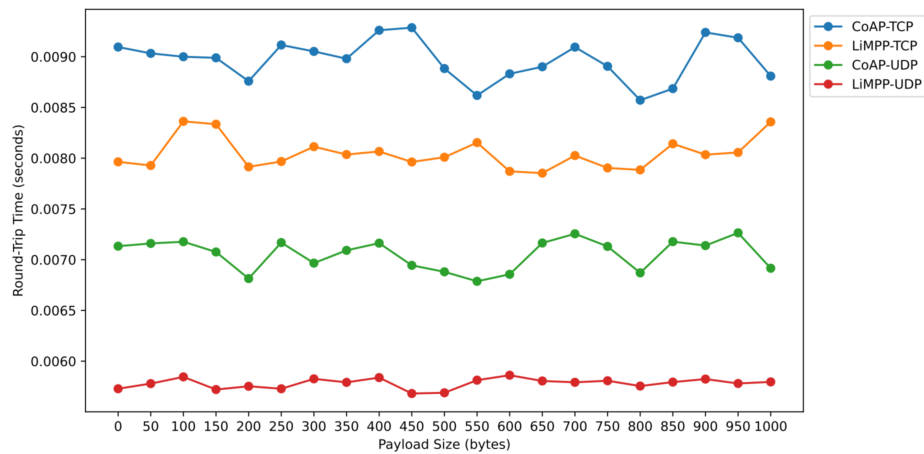

Figure 12. Point-to-Point Communication RTT between a Jetson Nano and a RaspberryPi.

In Figure 13, the efficiency of LiMP-TCP is 13.54% more than CoAP-TCP and that of LiMP-UDP is 20.85% more than CoAP-UDP.

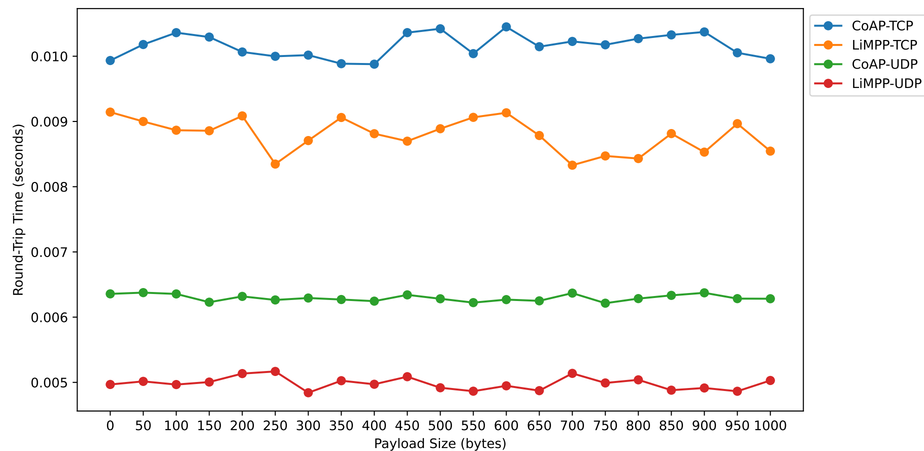

Figure 13. Point-to-Point Communication RTT between a Jetson Nano and a Jetson Tegra.

In Figure 14, the efficiency of LiMP-TCP is 11.22% more than CoAP-TCP and that of LiMP-UDP is 34.58% more than CoAP-UDP.

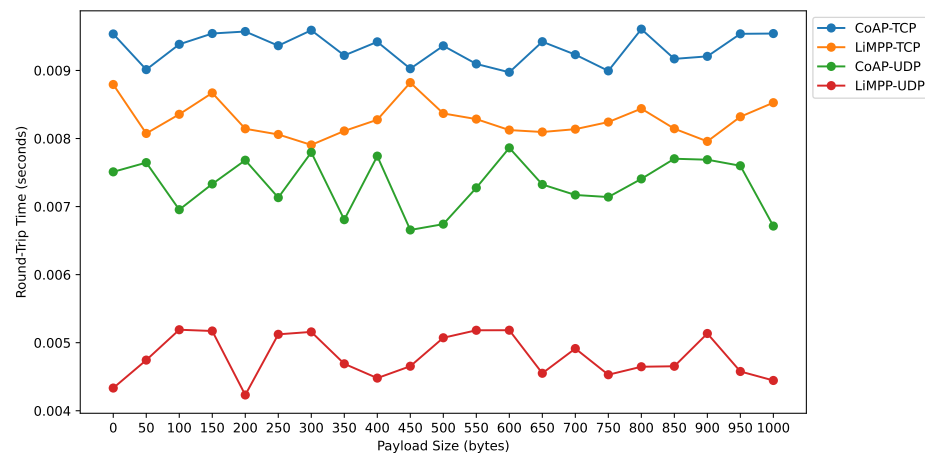

Figure 14. Point-to-Point Communication RTT between two RaspberryPi devices.

In Figure 15, the efficiency of LiMP-TCP is 14.30% more than CoAP-TCP and that of LiMP-UDP is 20.54% more than CoAP-UDP.

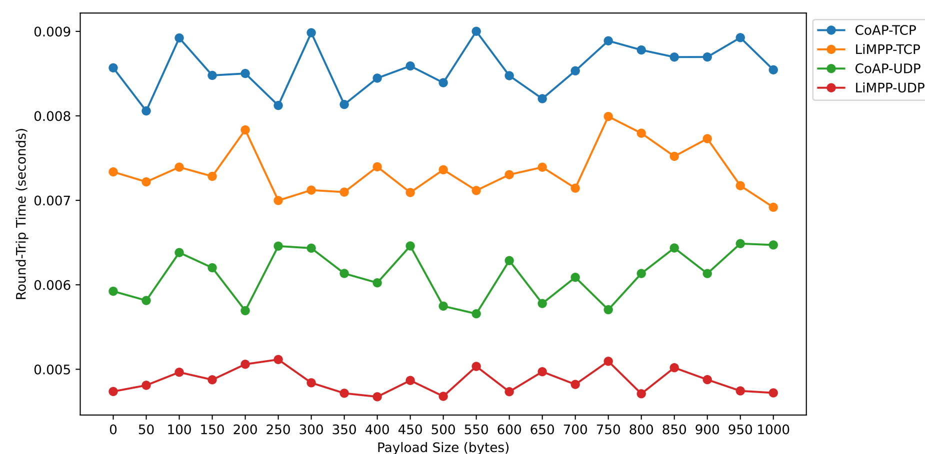

Figure 15. Point-to-Point Communication RTT between a Raspberry Pi and a Jetson Tegra.

In Figure 16, the efficiency of LiMP-TCP is 13.76% more than CoAP-TCP and that of LiMP-UDP is 44.39% more than CoAP-UDP.

The RTTs for communication between the iPhone and the other embedded devices showed a lower RTT (very good) for LiMP-UDP as compared to the other protocols; as shown in Figures 17–19. For instance, in Figure 19, LiMP-UDP was three-times lesser than that of CoAP-UDP with an average RTT of 4.55 ms. In Figure 18, the efficiency of LiMP-TCP is 39.90% more than CoAP-TCP and that of LiMP-UDP is 56.22% more than CoAP-UDP.

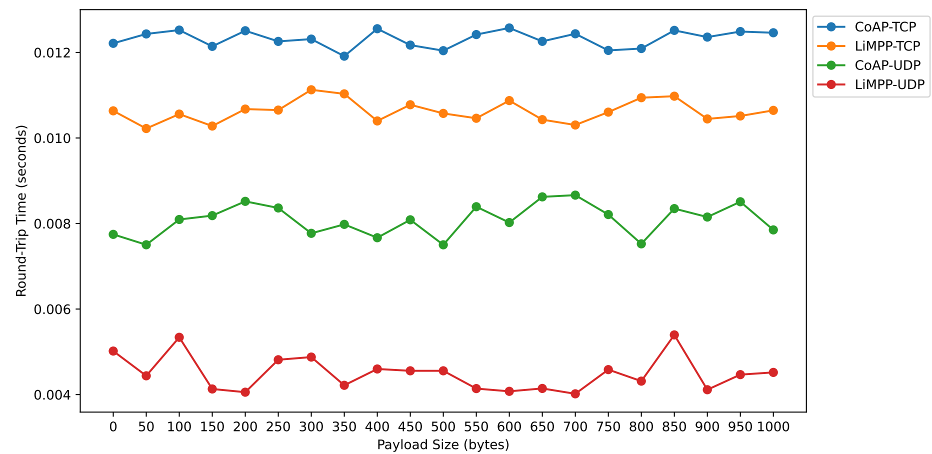

Figure 16. Point-to-Point Communication RTT between two Jetson Tegra devices.

In Figure 17, the efficiency of LiMP-TCP is 31.60% more than CoAP-TCP and that of LiMP-UDP is 72.11% more than CoAP-UDP.

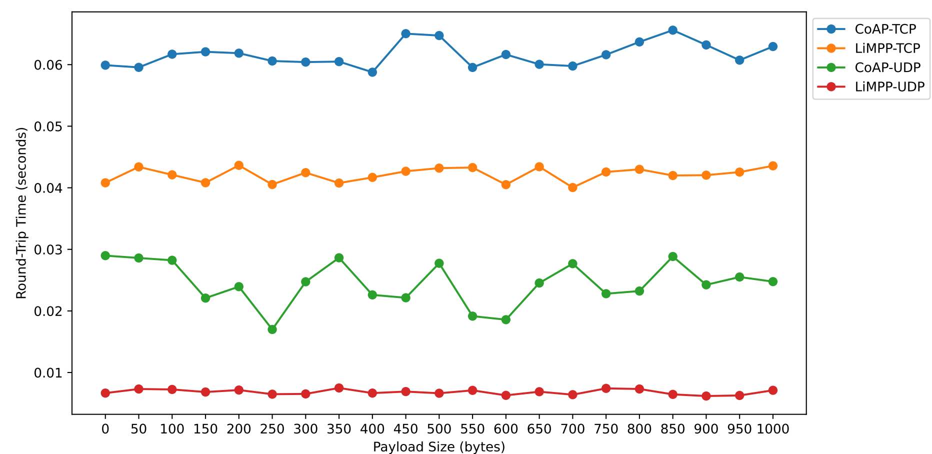

Figure 17. Point-to-Point Communication RTT between a Phone and a Jetson Nano.

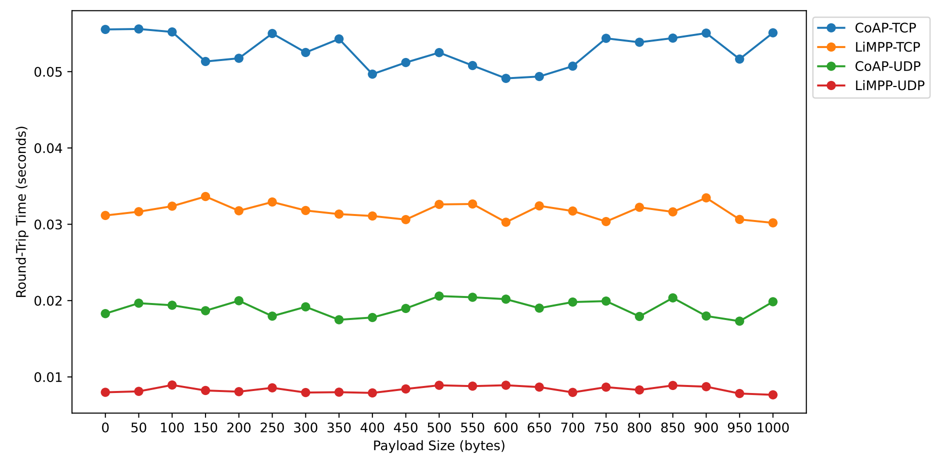

Figure 18. Point-to-Point Communication RTT between a Phone and a RaspberryPi.

In Figure 19, the efficiency of LiMP-TCP is 39.47% more than CoAP-TCP and that of LiMP-UDP is 67.10% more than CoAP-UDP.

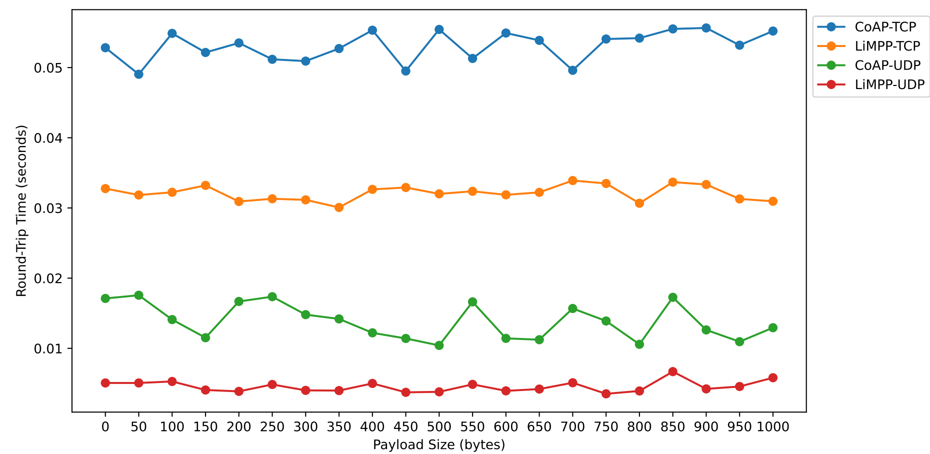

Figure 19. Point-to-Point Communication RTT between a Phone and a Jetson Tegra.

For the communication between the two Phones, the marginal difference between each protocol averages 0.2 ms. In Figure 20, the efficiency of LiMP-TCP is 17.65% more than CoAP-TCP and that of LiMP-UDP is 21.32% more than CoAP-UDP.

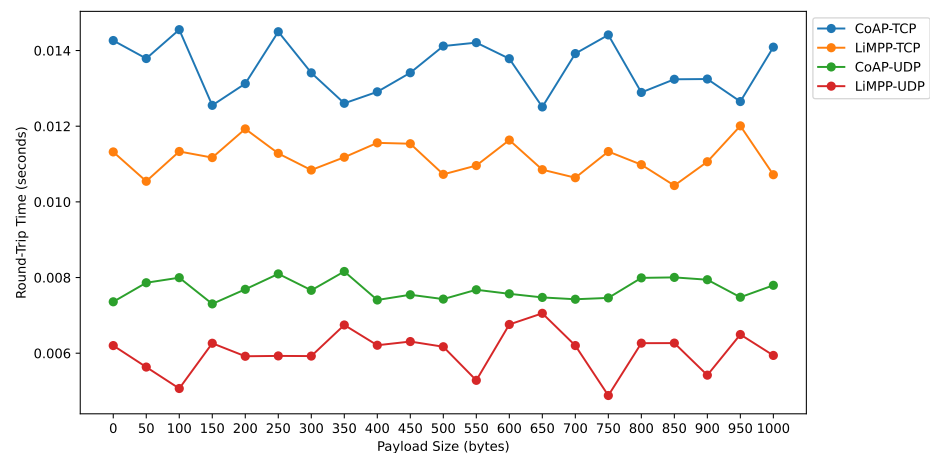

Figure 20. Point-to-Point Communication RTT between two Phones.

For the communication between the devices and the remote server, the marginal difference between LiMP-UDP and CoAP-UDP averages 5 ms. Also the LiMP-TCP outperformed that of CoAP-TCP with a marginal difference of 0.7 ms. These deductions are observed in Figures 21–24. In Figure 21, the efficiency of LiMP-TCP is 15% more than CoAP-TCP and that of LiMP-UDP is 15.90% more than CoAP-UDP.

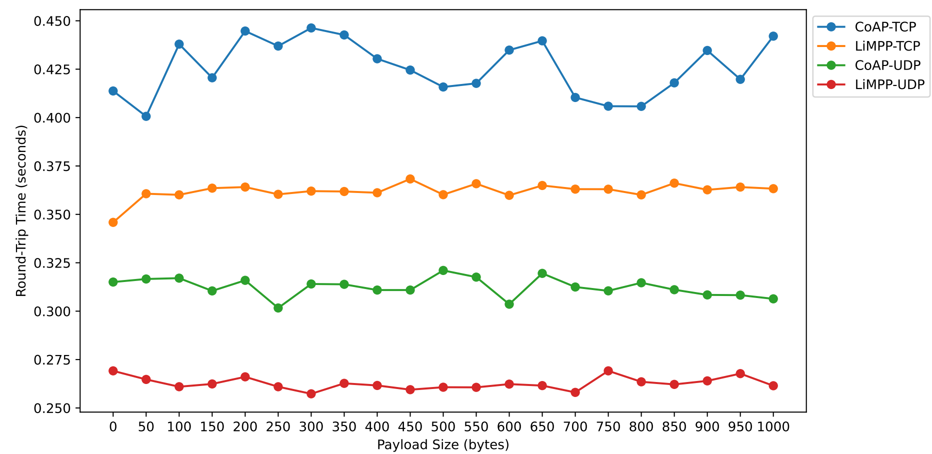

Figure 21. Point-to-Point Communication RTT between a Jetson Nano and a remote Server.

In Figure 22, the efficiency of LiMP-TCP is 15.04% more than CoAP-TCP and that of LiMP-UDP is 15.85% more than CoAP-UDP.

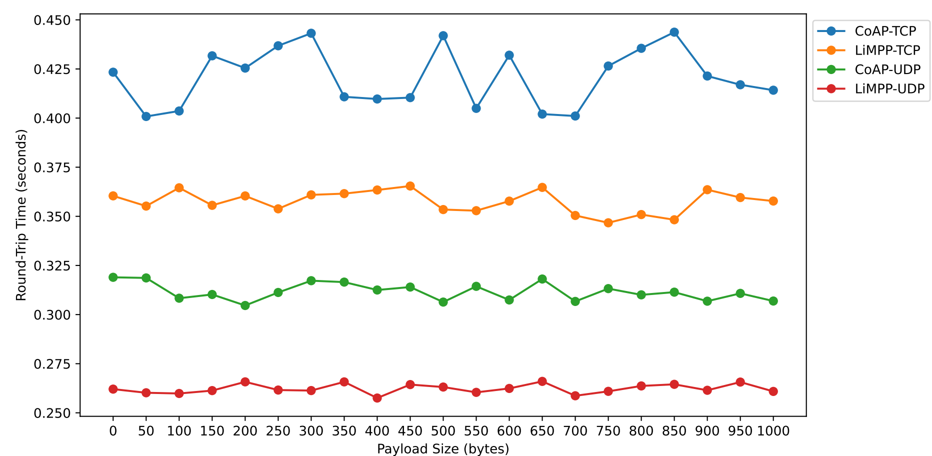

Figure 22. Point-to-Point Communication RTT between a RaspberryPi and a remote Server.

In Figure 23, the efficiency of LiMP-TCP is 14.86% more than CoAP-TCP and that of LiMP-UDP is 16.15% more than CoAP-UDP.

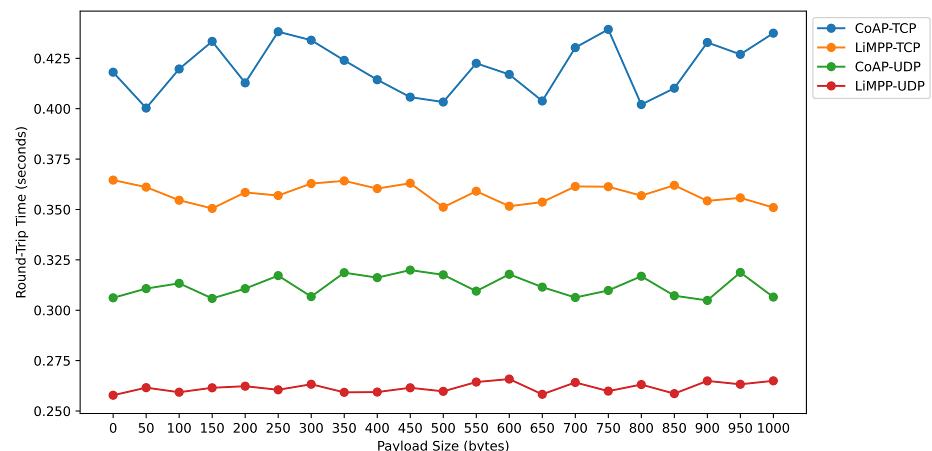

Figure 23. Point-to-Point Communication RTT between a Jetson Tegra and a remote Server.

In Figure 24, the efficiency of LiMP-TCP is 15.45% more than CoAP-TCP and that of LiMP-UDP is 16.37% more than CoAP-UDP.

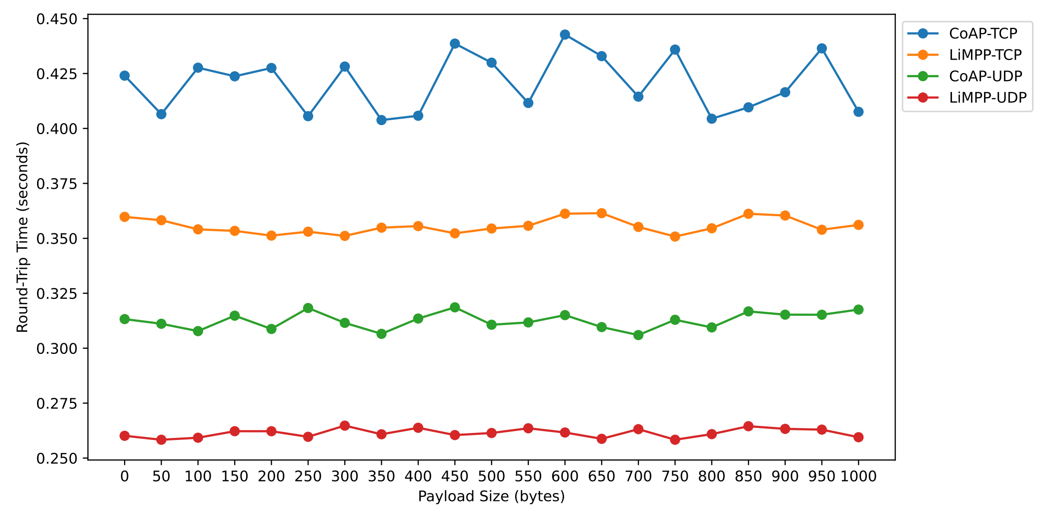

Figure 24. Point-to-Point Communication RTT between a Phone and a remote Server.

In summary, for communication over LAN, the LiMP-TCP outperformed the CoAP-TCP by an average of 21% whereas that of LiMP-UDP was over 37%. For a device to remote server communication, LiMP outperformed CoAP by an average of 15%. The LiMP-UDP achieved the fastest RTT. The LiMP-TCP is better in comparison to the CoAP-TCP in instances where the choice of the transport protocol has to be TCP.

5.1. Security Analysis of LiMP

This section provides a security analysis of LiMP in comparison to CoAP. The CoAP header size (as shown in Figure 4) consists of a 16-bit message-ID used to identify request/response messages; a total of 65536 possible identifiers. One can argue that reducing the Message-ID to 8-bit (as shown in Figure 7), in the case of LiMP, makes the protocol vulnerable to message spoofing. In this section, we demonstrate the following:

1. the ease of spoofing both 16 and 8-bit message-ID, and
2. how LiMP uses a simple ETag generation mechanism to prevent message spoofing.

Table 4 and Figure 25 show how easy it is to compute both 16 and 8-bit Message-IDs. This analysis was made on the same devices used in the benchmark analysis.

Table 4. The Time Taken to Compute both 16 and 8-bit possible message-IDs.

Device	16-bit (Time/ms)	8-bit (Time/ms)
Nvidia Jetson Nano	8.037	0.022
Nvidia Tegra	12.989	0.032
Raspberry Pi	20.946	0.063
iPhone 7	1.891	0.005

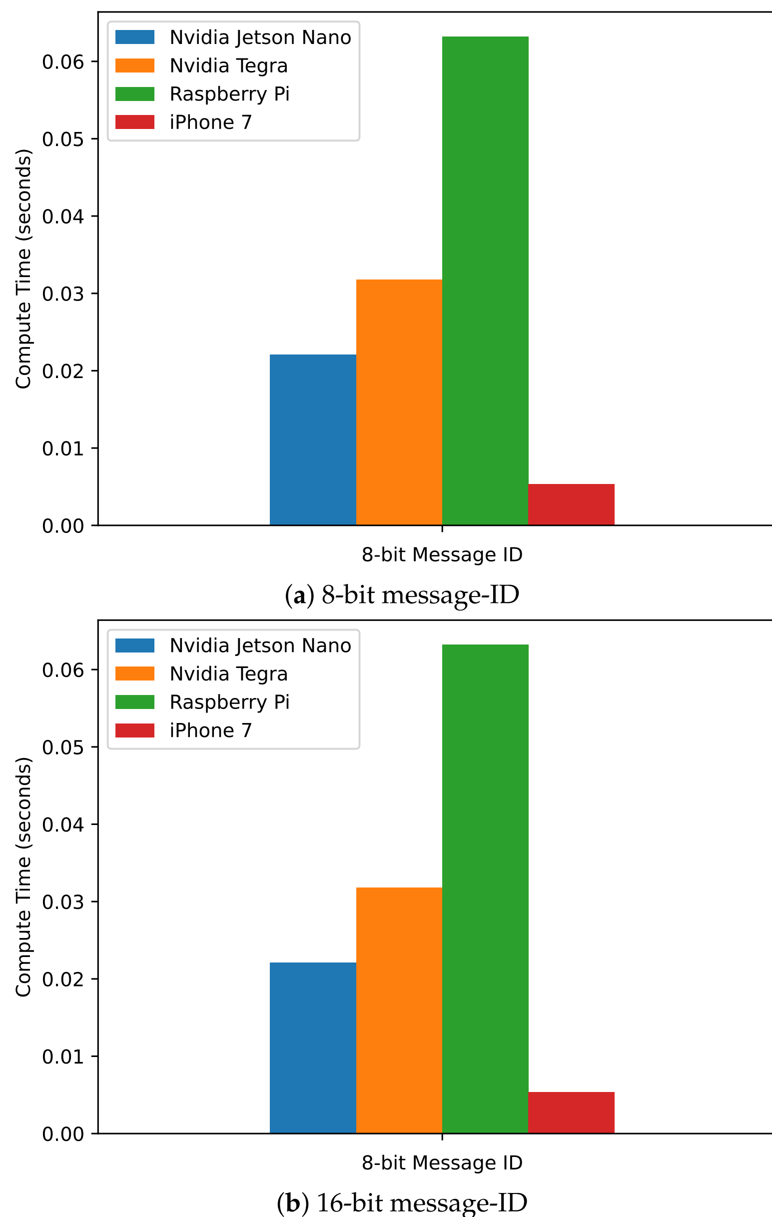

Figure 25. Comparison of the Time Taken to Compute All Possible 8 and 16-bit message-IDs.

It can be observed that both 8 and 16-bit Message-IDs are computationally feasible and easy for a compromised node to spoof Message-IDs. The fastest compute time was 1.890 ms in the case of the 16-bit Message-ID and 0.005 ms for that of the 8-bit Message-ID.

LiMP uses the 8-bit Message-ID as a seed to generate an ETag to prevent message spoofing. The ETag generation technique is based on binary-shift operations; since binary-shift operations are easy to compute as compared to other algebraic operations. The ETag generation mechanism (Algorithm 1) uses the seed together with a timestamp value (which is also contained in the message payload) to produce a unique hexadecimal string (the ETag). It is assumed that each IoT device is time-synchronized; hence time becomes a good entropy source. Line 1 of Algorithm 1 is the procedure for the generation of the ETag. As input parameters, it requires the seed value and the timestamp (in milliseconds). The binary equivalent of the seed value is prefixed to the binary value of the current timestamp. The resulting value is then converted to its hexadecimal equivalent, which then becomes the ETag. The ETag is collision-resistant due to its usage of the current timestamp as a source of entropy. The seed value ensures that concurrent requests do not end up having the same ETag value.

Algorithm 1 ETag Generation Algorithm.

```
1: procedure GENERATE(seed, timestamp)
2:     % seed (bounded between 0, 255 inclusive), timestamp (time in milliseconds)
3:     ← bin2hex(bin(seed) + bin(timestamp))
4: end procedure

5: procedure BIN2HEX(binary_data)
6:     ← hex(int(binary_data))
7: end procedure

8: procedure HEX2BIN(hex_data)
9:     ← bin(int(hex_data))
10: end procedure

11: procedure IS_VALID(e_tag, timestamp, seed)
12:     if generate(seed, timestamp) == e_tag then
13:         ← True
14:     else
15:         ← False
16:     end if
17: end procedure
```

6. Conclusions and Recommendation

With the proliferation of IoT devices, it has become essential to simplify the development of network and application layer protocols for M2M communication. Although the most common application layer protocols such as CoAP, MQTT, XMPP, DDS, and XMPP are applicable in IoT networks; they are limited in instances where minimal overhead and message sizes are key requirements. For data-centric IoT, the minimal header and message complexity, as well as efficient delivery of messages, is key. In this research paper, we proposed a lightweight messaging protocol with a minimal header (2 bytes) size and a PDU of 68 and 44 bytes for TCP and UDP respectively. With the reduced header size, it can be argued that it compromises the security of the proposed protocol. Therefore, we proposed a mechanism through which spoofing of messages can be detected by proposing the use of an ETag. We also demonstrated that the proposed messaging protocol has a faster RTT due to reduced complexity; for communication over LAN, the LiMP-TCP outperformed the CoAP-TCP by an average of 21% whereas that of LiMP-UDP was over 37%. For a device to remote server communication, LiMP outperformed CoAP by an average of 15%. In the context of IoT, aside D2D communication, host-discovery and multicast communication are essential. Future works will explore how this minified protocol can be leveraged for host discovery with minimal broadcast overhead.

7. Patents

IoT applicable.

Author Contributions: Conceptualization, J.O.A. and H.N.-M.; methodology, J.O.A. and D.Y.; validation, J.J.K., H.N.-M., J.D.G. and D.Y.; formal analysis, J.J.K.; investigation, J.D.G.; resources, D.Y.; writing—original draft preparation, J.O.A.; writing—review and editing, J.O.A.; supervision, J.J.K., J.D.G. and H.N.-M.; project administration, J.J.K.; funding acquisition, J.J.K. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by KNUST Engineering Education Project, Ghana.

Institutional Review Board Statement: Not Applicable.

Informed Consent Statement: Not Applicable.

Data Availability Statement: The data used in this study is available at (https://github.com/jayluxferro/levis, accessed on 6 January 2022).

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

IoT	Internet of Things
D2D	Device-to-Device
LAN	Local Area Network
M2M	Machine-to-Machine
MQTT	Message Queueing Telemetry Transport
AMQP	Advanced Message Queuing Protocol
CoAP	Constrained Application Protocol
eUtility	External Utility
BLE	Bluetooth Lower Energy
NFC	Near Field Communication
LPWAN	Low Power Wide Area Network
WSN	Wireless Sensor Networks
6LoWPAN	IPv6 over Low-Power Wireless Personal Area Networks
RPL	Routing Protocol for Low-Power and Lossy Networks
TLS	Transport Layer Security
DTLS	Datagram Transport Layer Security
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
HTTP	HyperText Transfer Protocol
LiMP	Lightweight Messaging Protocol
DDS	Data Distribution Service
XMPP	eXtensible Messaging and Presence Protocol
REST	Representational State Transfer
CoRE	Constrained RESTful Environments
IP	Internet Protocol
IPv6	IP version 6
API	Application Programming Interface
QoS	Quality of Service
XML	eXtensible Markup Language
SCTP	Stream Control Transmission Protocol
UTF	Unicode Transformation Format
JSON	JavaScript Object IoTation
ETag	Entity Tag
PDU	Packet Data Unit
RTT	Round-Trip Time
ARM	Advanced RISC Machines

References

1. Laghari, A.A.; Wu, K.; Laghari, R.A.; Ali, M.; Khan, A.A. A Review and State of Art of Internet of Things (IoT). In *Archives of Computational Methods in Engineering*; Springer: Berlin/Heidelberg, Germany, 2021; doi:10.1007/s11831-021-09622-6. [CrossRef]
2. Malhotra, P.; Singh, Y.; Anand, P.; Bangotra, D.K.; Singh, P.K.; Hong, W.C. Internet of Things: Evolution, Concerns and Security Challenges. *Sensors* **2021**, *21*, 1809. [CrossRef] [PubMed]
3. Wang, J.; Lim, M.K.; Wang, C.; Tseng, M.L. The evolution of the Internet of Things (IoT) over the past 20 years. *Comput. Ind. Eng.* **2021**, *155*, 107174. [CrossRef]
4. Hanes, D.; Salgueiro, G.; Grossetete, P.; Barton, R.; Henry, J. *IoT Fundamentals: Networking Technologies, Protocols, and Use Cases for the Internet of Things*; Cisco Press: Indianapolis, IN, USA, 2017.
5. Sicari, S.; Rizzardi, A.; Grieco, L.; Coen-Porisini, A. Security, privacy and trust in Internet of Things: The road ahead. *Comput. Netw.* **2015**, *76*, 146–164. [CrossRef]
6. Mosenia, A.; Jha, N.K. A Comprehensive Study of Security of Internet-of-Things. *IEEE Trans. Emerg. Top. Comput.* **2017**, *5*, 586–602. [CrossRef]

7. Khan, M.A.; Salah, K. IoT security: Review, blockchain solutions, and open challenges. *Future Gener. Comput. Syst.* **2018**, *82*, 395–411. [CrossRef]
8. Voas, J. Networks of 'Things'. *NIST Spec. Publ.* 2016, *800*, 183–800.
9. Di Martino, B.; Rak, M.; Ficco, M.; Esposito, A.; Maisto, S.; Nacchia, S. Internet of things reference architectures, security and interoperability: A survey. *Internet Things* **2018**, *1–2*, 99–112. [CrossRef]
10. Radoglou Grammatikis, P.I.; Sarigiannidis, P.G.; Moscholios, I.D. Securing the Internet of Things: Challenges, threats and solutions. *Internet Things* **2019**, *5*, 41–70. [CrossRef]
11. binti Mohamad Noor, M.; Hassan, W.H. Current research on Internet of Things (IoT) security: A survey. *Comput. Netw.* **2019**, *148*, 283–294. [CrossRef]
12. Yousuf, O.; Mir, R.N. A survey on the Internet of Things security: State-of-art, architecture, issues and countermeasures. *Inf. Comput. Secur.* **2019**, *27*, 292–323. [CrossRef]
13. Aly, M.; Khomh, F.; Haoues, M.; Quintero, A.; Yacout, S. Enforcing security in Internet of Things frameworks: A Systematic Literature Review. *Internet Things* **2019**, *6*, 100050. [CrossRef]
14. Bhabad, M.A.; Bagade, S.T. Article: Internet of Things: Architecture, Security Issues and Countermeasures. *Int. J. Comput. Appl.* **2015**, *125*, 1–4.
15. HaddadPajouh, H.; Dehghantanha, A.; Parizi, R.M.; Aledhari, M.; Karimipour, H. A survey on Internet of Things security: Requirements, challenges, and solutions. *Internet Things* **2021**, *14*, 100129. [CrossRef]
16. Ogonji, M.M.; Okeyo, G.; Wafula, J.M. A survey on privacy and security of Internet of Things. *Comput. Sci. Rev.* **2020**, *38*, 100312. [CrossRef]
17. Lombardi, M.; Pascale, F.; Santaniello, D. Internet of Things: A General Overview between Architectures, Protocols and Applications. *Information* **2021**, *12*, 87. [CrossRef]
18. Lin, J.; Yu, W.; Zhang, N.; Yang, X.; Zhang, H.; Zhao, W. A Survey on Internet of Things: Architecture, Enabling Technologies, Security and Privacy, and Applications. *IEEE Internet Things J.* **2017**, *4*, 1125–1142. [CrossRef]
19. Shelby, Z.; Hartke, K.; Bormann, C. The Constrained Application Protocol (CoAP). RFC 7252, 2014. Available online: https://iottestware.readthedocs.io/en/master/coap_rfc.html (accessed on 10 December 2021).
20. Postel, J. User Datagram Protocol STD 6, RFC 768, The Request for Comments (RFC) Series, 1980. Available online: https://www.hjp.at/doc/rfc/rfc768.html (accessed on 10 December 2021).
21. Saint-Andre, P.; Loreto, S.; Salsano, S.; Wilkins, G. *Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP*; RFC 6202: 2011; doi:10.17487/RFC6202. Available online: https://www.hjp.at/doc/rfc/rfc6202.html (accessed on 10 December 2021). [CrossRef]
22. Melnikov, A.; Fette, I. *The WebSocket Protocol*; RFC 6455, The Request for Comments (RFC) Series, 2011; doi:10.17487/RFC6455. Available online: https://www.hjp.at/doc/rfc/rfc6455.html (accessed on 10 December 2021). [CrossRef]
23. *Message Queueing Telemetry Transport Protocol*; v3.1.1; 2014. OASIS Standard. Available online: https://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html (accessed on 10 December 2021).
24. Saint-Andre, P. Extensible Messaging and Presence Protocol (XMPP): Core; RFC 6120, The Request for Comments (RFC) Series, 2011; doi:10.17487/RFC6120. Available online: https://www.hjp.at/doc/rfc/rfc6120.html (accessed on 10 December 2021). [CrossRef]
25. Saint-Andre, P. Extensible Messaging and Presence Protocol (XMPP): Address Format; RFC 6122, The Request for Comments (RFC) Series, 2011; doi:10.17487/RFC6122. Available online: https://www.hjp.at/doc/rfc/rfc7622.html (accessed on 10 December 2021). [CrossRef]
26. *Data Distribution Service*; v1.4; 2015. Object Management Group (OMG). Available Online: https://www.omg.org/spec/DDS/1.4/About-DDS (accessed on 10 December 2021).
27. Naik, N. Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP. In Proceedings of the 2017 IEEE International Systems Engineering Symposium (ISSE), Vienna, Austria, 11–13 October 2017; pp. 1–7.
28. Al-Masri, E.; Kalyanam, K.R.; Batts, J.; Kim, J.; Singh, S.; Vo, T.; Yan, C. Investigating Messaging Protocols for the Internet of Things (IoT). *IEEE Access* **2020**, *8*, 94880–94911. [CrossRef]
29. Huh, J.H. Reliable User Datagram Protocol as a Solution to Latencies in Network Games. *Electronics* **2018**, *7*, 295. [CrossRef]
30. Thangavel, D.; Ma, X.; Valera, A.; Tan, H.X.; Tan, C.K.Y. Performance evaluation of MQTT and CoAP via a common middleware. In Proceedings of the 2014 IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), Singapore, 21–24 April 2014; pp. 1–6.
31. Tan, E.K.; Chong, Y.W.; Setyawan, R.A.; Niswar, M.; Mya, K.T. Lightweight messaging protocol for precision agriculture. In Proceedings of the 2021 International Conference on Information Networking (ICOIN), Jeju Island, Korea, 13–16 January 2021; pp. 403–407.
32. Thota, P.; Kim, Y. Implementation and Comparison of M2M Protocols for Internet of Things. In Proceedings of the 2016 4th Intl Conf on Applied Computing and Information Technology/3rd Intl Conf on Computational Science/Intelligence and Applied Informatics/1st Intl Conf on Big Data, Cloud Computing, Data Science Engineering (ACIT-CSII-BCD), Las Vegas, NV, USA, 12–14 December 2016; pp. 43–48.