IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received April 20, 2020, accepted May 3, 2020, date of publication May 8, 2020, date of current version June 2, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2993363

Investigating Messaging Protocols for the Internet of Things (IoT)

EYHAB AL-MASRI, (Member, IEEE), KARAN RAJ KALYANAM, JOHN BATTS, JONATHAN KIM, SHARANJIT SINGH, TAMMY VO, AND CHARLOTTE YAN

School of Engineering and Technology, University of Washington Tacoma, Tacoma, WA 98402, USA

Corresponding author: Eyhab Al-Masri (ealmasri@uw.edu)

ABSTRACT As the number of Internet of Things (IoT) devices proliferates, the magnitude and velocity of data continues to increase rapidly. IoT systems rely primarily on using messaging protocols for exchanging IoT data and there exists several protocols or frameworks that support distinct types of messaging patterns. Given that IoT devices typically have limited computational resources and processing power, choosing a lightweight, reliable, scalable, interoperable, extensible and secure messaging protocol becomes a very challenging task. As a result, it is not uncommon that IoT systems may employ multiple messaging protocols for supporting device heterogeneity and different message exchange patterns. In addition, basic similarities among existing several messaging protocols or frameworks that exist today for exchanging IoT data within IoT systems suggest the potential of interoperability. Given that IoT systems help facilitate the interconnectivity among distributed, heterogeneous entities, interoperability among existing messaging protocols will play an increasingly important role in simplifying the development and deployment of IoT systems. In this paper, we present a comprehensive review of the existing messaging protocols that can be used in deploying IoT systems. Throughout this paper, we highlight the protocols' distinctive approaches and applicability of using them across various IoT environments. In addition, we highlight challenges, strengths and weaknesses of these messaging protocols in the context of IoT.

INDEX TERMS Internet of Things, IoT, HTTP, MQTT, CoAP, AMQP, XMPP, DDS, data distribution service, constrained application protocol, message queuing telemetry transport, extensible messaging and presence protocol, HyperText transfer protocol, edge computing, fog computing, cloud applications.

I. INTRODUCTION

Advances in networking technology have profoundly contributed to how IoT devices produce, exchange and perceive data. This is becoming more evident as the magnitude and rate at which data is generated by IoT devices is rapidly increasing. This has also contributed to the deployment of a wide array of messaging protocols that enabled IoT devices to exchange messages more efficiently. Application layer protocols are considered as the underlying layers used by applications in defining the structure of message exchanges and how they can be transmitted. In this paper, we primarily focus on the messaging protocols at the application layer of the Open System Interconnection (OSI) model [8].

There are a number of communication or middleware protocols that are commonly used in the deployment of IoT applications including HyperText Transfer Protocol (HTTP), Message Queuing Telemetry Transport (MQTT), Advanced Message Queuing Protocol (AMQP), Constrained Application Protocol (CoAP), Extensible Messaging and Presence Protocol (XMPP), and Data Distribution Service (DDS), among others. Such protocols support to some extent similar features in terms of connectivity, however, they vary in the degree to which these features are offered. Given that IoT systems primarily depend on IoT devices for data collection and message exchanges for the overall functioning of the system, the choice of which communication or messaging protocol to use for device interconnectivity becomes a very time consuming and challenging task. Furthermore, when choosing a suitable messaging protocol, it is imperative to consider the hardware characteristics of IoT devices and type of data link layer protocols employed.

In addition, IoT devices can vary significantly in terms of the bandwidth they support. That is, IoT devices do not use a universal radio technology and therefore the physical data rate they support varies considerably depending on the

The associate editor coordinating the review of this manuscript and approving it for publication was Noor Zaman.

size and hardware components used to build these devices. For example, low-rate wireless personal area networks (LR-WPANs) leverage the IEEE 802.15.4 physical layer that supports data rates up to 250 Kbps with packet length of approximately 127 bytes. Other variations also exist at other layers of the OSI model such as HTTP.

A common request header size in HTTP, which includes basic attributes such as user-agent, is approximately 700-800 bytes. However, an uncompressed request header may vary in size ranging from 200 bytes to more than 2KB. Moreover, using application layer protocols that are capable of capturing data much faster than actual physical data rates often leads to high latency. Therefore, it would be desirable when developing IoT applications to consider messaging protocols that can accommodate or support physical data rates at the data link layer.

IoT devices producing data at a high velocity often require lightweight communication protocols. For example, a small battery-supported IoT temperature device programmed to send localized temperature readings at a fixed frequency (e.g. every 10 seconds) to an IoT hub (e.g. Microsoft Azure IoT Hub) may not have sufficient power by the time it transmits a day-long temperature readings using HTTP. It would be desirable in this case to alternatively use a lightweight messaging protocol such as Message Queuing Telemetry Transport (MQTT) in which the smallest packet size is 2 bytes (compared to 200 bytes in HTTP in the uncompressed header size scenario). MQTT is a Machine-to-Machine (M2M)/ Internet of Things (IoT) connectivity protocol designed as an asynchronous lightweight publish/subscribe messaging transport protocol [1].

Performance of IoT devices and applications can be significantly influenced by the choice of messaging protocol used. By employing a suitable messaging protocol helps reduce network traffic and latency and thus increases an IoT application's reliability [8]. However, there exists no universal protocol that can be used across heterogeneous IoT environments. To this extent, choosing an appropriate messaging protocol depends on a number of factors including IoT application's business requirements, software capabilities, device or hardware capabilities, and average size of data exchanges, among many others. Furthermore, in deploying IoT devices and developing an IoT application, it is imperative to consider the main characteristics and distinctive features offered by existing protocols. Hence, throughout this study we focus primarily on present messaging or middleware protocols that are commonly used in deploying IoT systems.

There are a number of survey research studies that have examined protocols used for IoT connectivity at the data link layer [120]–[129]. However, the diversity which exists at this layer in terms of protocols' goals, architecture and capabilities makes it progressively difficult to relate these protocols with those that interface at the application level. That is, understanding protocols at the data link layer is not sufficient to build IoT applications. It is essential to also consider protocols that exist at the application level complementing those that exist at the data link layer.

Understanding data link protocols that indirectly interact with those at application layer through the OSI model is essential. Because data link layer protocols (e.g. WirelessHART, Sigfox, LoRa, among others) have different goals and target different types of IoT devices, carefully choosing a protocol closer to the application layer (e.g. HTTP, CoAP, MQTT, among others) while also considering crucial system requirements such as Quality of Service (QoS), bandwidth, interoperability and security becomes inevitable. Although there exists a number of studies that have examined protocols at the data link layer [120]–[129] and the application layer [2]–[7], [130]–[140], there exists a gap on how these studies connect protocols across a number of layers of the OSI model. That is, complementing the knowledge area across multiple layers of the protocol stack for supporting an application's lifecycle is therefore needed. This paper attempts to reduce this gap by providing technical details and thorough analysis of protocols that can be used for developing IoT applications.

This research paper is organized as follows. Section II discusses some of the related work. Section III discusses existing protocols and the different layers of the OSI model. Section IV presents a detailed analysis of existing messaging or middleware protocols including HTTP, CoAP, MQTT, AMQP, XMPP and DDS. Section V provides a comprehensive comparison of the features of existing messaging protocols. Section VI discusses how messaging protocols address the challenges associated with building IoT applications. Section VII highlights the strengths and weaknesses of each protocol in the context of IoT. Finally, Section VIII provides the conclusion and future work suggestions.

II. RELATED WORK

Research and investigation in the IoT domain has focused primarily on data accumulation, data analysis and transformation and collaborative processes implemented in the form of RESTful services. To organize IoT data flows, a multilevel model of IoT or an IoT reference model is employed consisting of a number of layers including: (a) physical device and controllers, (b) connectivity, (c) data accumulation and abstraction (e.g. big data) and (d) application. In this section, we identify key research studies at the application layer protocol.

In [2], the authors conduct a comparison study into the usefulness of utilizing CoAP and MQTT in smart healthcare IoT applications. The paper identifies weaknesses of each of these protocols and security implications in the context of healthcare. Furthermore, the study focuses on the security aspects and identifying weaknesses in protecting sensitive and private patient data. The authors attempt to classify the types of threats as Privacy and Confidentiality, Availability and Integrity [2]. In [3], Naik provides a study surveying some of the existing messaging protocols and identifying their pros and cons [3]. In [7], Dizdarević *et al.* provide

another survey study of existing application layer protocols while distinguishing their operability for possible integration into fog- and cloud-based systems [7].

In [4], Thangavel *et al.* examine the end-to-end performance of MQTT and CoAP using a common middleware. The study focuses primarily on the usefulness of such protocols in wireless sensor networks (WSNs). The authors propose a middleware for extending existing and future protocols. Results from this study show that MQTT messages have lower delay compared to those of CoAP when considering lower packet loss rates. However, MQTT's performance degrades compared to that of CoAP when considering messages at higher data loss rates. The study also identifies that CoAP generates much less traffic overhead compared to MQTT when message sizes are small and loss rate is equal to or less than twenty-five percent [4].

Tandale *et al.* performed a similar study in which they considered CoAP, MQTT and HTTP REST [5]. The authors used an arm-based device (Raspberry Pi 3) acting as a gateway to examine the reliability and network traffic of these protocols while considering various network conditions. That is, the study included two types of networks: (a) cellular 4G and (b) high-speed broadband connection. The authors conclude that CoAP performs more efficiently for small payloads and its performance deteriorates as the size of messages increases [5].

In [6], the authors focus in their study on the implementation and comparison of Machine-to-Machine (M2M) protocols for IoT. The study was centered on the deployment of an IoT system that collects temperature data while using both the MQTT and CoAP protocols. The study determines that both of these protocols achieve 100% data transfer with minimal packet loss induced and have relatively efficient support for re-transmitting data packets. Furthermore, the study finds that CoAP protocol's data loss rate is low when handling smaller data sizes. The study concludes that the performance of each protocol is fairly dependent on various network conditions (e.g. in the case of data retransmission as data size increases) [6].

Apart from research efforts that investigated the use of messaging protocols at the application layer, there have been a number of research studies that focused on the network layer protocols. Network layer protocols enable technologies at that layer to communicate in a unidirectional or bidirectional capability [119]. For example, network protocols have different identification techniques for locating devices over different types of networks (i.e. small or large). Such identification can be achieved, for example, through network addresses using IPv4, IPv6 or 6LoWPAN [120]. Because IoT devices vary in hardware capabilities (e.g. power consumption, connectivity medium, transmission coverage), identifying a suitable communication medium that can be used within the boundaries of IoT devices is challenging [121]–[123]. A number of survey studies examined the various differences amongst existing network layer protocols in the context of IoT.

In [124], the authors discuss key identification and tracking technologies for IoT devices. For example, the Radio Frequency Identification (RFID) is a suitable technology that is generally used for tracking and identification. RFID devices are commonly used in applications such as transportation, logistics, manufacturing, and equipment tracking, among others [119], [122]. However, IoT systems require smart devices that go beyond the traditional identification capabilities [124].

In addition, there exists multiple different types of networks that can be employed in IoT systems such as wireless sensor networks (WSNs). WSNs support longer transmission ranges and, unlike RFID sensor networks, offer a peer-to-peer communication mode. Because of the need for enhancing the network identification of IoT devices, improving data transmission rates and supporting mobility, a number of activities or alliances have evolved in recent years including, but not limited to, Electronic Product Code (EPC) led by EPCGlobal, Machine-to-Machine (M2M), 6LoWPAN, ZigBee, WirelessHART, NFC, ANT+, Thread, MiWi, and Weightless, among many others [119]. Many of these activities vary in terms of communication strategies they support or enable.

The authors in [125] provide a survey of the communication strategies that can be applied for building or deploying smart IoT applications. In this study, the authors identify four main communication strategies: (a) device-to-device (D2D), (b) device-to-cloud (D2C), (c) device-to-gateway (D2G) and (d) device-to-application (D2A). The authors examined the current state-of-the-art for identifying a technical taxonomy that can potentially cover all possible IoT communication types [124]. However, the study focused primarily on a broad taxonomy for covering these communication strategies. It would be desirable to include, in addition to considering this high-level overview of the communication strategies, more granular technical details that can help identify the scope to which existing messaging protocols map or support these communication strategies. This research study attempts to reduce this gap by identifying the level of support provided by existing messaging protocols for each of these communication strategies.

Furthermore, other studies have limited their scope to focus mainly on a particular applied area of IoT. For example, the authors in [120] reviewed communication technologies primarily for smart home systems. The survey study provides a feature comparison of many elements including architecture, software implementations, privacy and security and communication. The study also discusses the advantages and disadvantages of applying wired and wireless communication protocols in terms of frequency, data rates, and network topology, among others. However, the primary focus of the study in [120] is on short range technologies at the data link layer protocols without connecting them to those that exist at the application layer.

Although many of the existing research studies have focused on identifying the advantages and disadvantages of

a number of communication protocols, the need for having a research study that provides detailed comparison with relevant IoT use cases of the existing messaging protocols at the application layer while considering differentiating factors such as interoperability, scalability and performance adaptability and extensibility, security and reliability becomes inevitable. In this paper, we provide a comprehensive study that investigates the use of existing messaging protocols for IoT application and identify the challenges associated with their usage across various IoT deployment strategies (e.g. edge- or fog- versus cloud-based). The following section describes a communication-centric IoT reference model, IoT application requirements and various IoT-related specifications, standards and alliances that exist today.

III. IoT MESSAGING PROTOCOLS

In a traditional IoT cloud architecture, the plurality of data generated by IoT devices relocate to the cloud for data accumulation (i.e. storage) and abstraction (i.e. aggregation and access). In contrast, an IoT edge-based architecture reduces the amount of data in which only partial data relocates to the cloud for further storage and processing.

In the edge-based model, network and system architectures are generally responsible for not only collecting data at the edge of the network but also performing advanced functions such as data analysis and processing. That is, the data accumulation and abstraction partially shift from the cloud to the edge of the network which significantly contributes to reduction in network traffic as the number of IoT devices increases. Therefore, it is imperative when choosing a deployment architecture to identify a suitable messaging protocol. To this extent, in the following section, we discuss a communication-centric IoT reference model for the purpose of distinguishing the various protocols and specifications that exist across the various IoT layers.

A. THE IoT REFERENCE MODEL

Despite the differences between IoT deployment approaches, producing data which occurs at various rates is achieved by the physical layer to which IoT devices connect to a network (i.e. wired or wireless). Given that IoT devices at this layer may have limited power, the rate at which data generates requires special consideration to the amount of power consumed by these devices. That is, reducing the velocity or the rate at which data generates on the edge of the network translates into longer battery life of IoT devices while reducing bandwidth.

A plethora of standards, specifications and technologies at this layer exist including Ethernet 802.3, Wi-Fi 802.11 a/b/g/n, Low Power Wide Area Network (LWPAN), Long Range Radio (e.g. LoRa, Sigfox, Narrowband IoT – NB-IoT), Zigbee, Cellular (2G, 3G, 4G, LTE, 5G), 802.15.4, among many others. Figure 1 presents a Communication-centric Internet of Things (CIoT) reference model.

In the following sections, we describe the main functionality supported by the layers presented in Figure 1 and briefly discuss the relevant protocols, specifications or standards that exist at each layer.

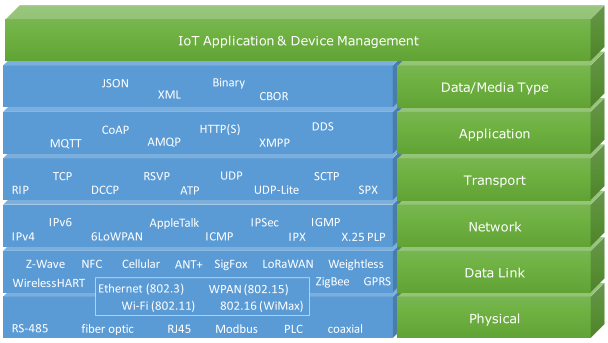

FIGURE 1. A communication-centric IoT reference model (CIoT).

B. PHYSICAL AND DATA LINK LAYERS

At the lowest level of the IoT reference model presented in Figure 1 is the physical layer which contains the electronic circuitry for transmissions. The data link layer is responsible for data transfers between network entities. There are a large number of technologies that exist at the data link layer each of which has been designed to target specific applications or connectivity strategy. For example, Z-Wave enables IoT devices to be controlled via the Internet and is very common among smart home systems.

On the other hand, the Weightless technology uses a set of Low-Power Wide-Area Network (LPWAN) standards for data exchanges primarily between a base station and thousands of IoT nodes. There are three types of Weightless protocols including: (a) Weightless-N, Weightless-W and Weightless-P. Each of these Weightless protocols is designed for specific use cases. Weightless-W, for example, operates in the TV White Space (TVWS) spectrum and takes advantage of the ultra-high frequency (UHF). Weightless-N uses an ultra-narrowband which is ideal for sensor-based networks and metering systems, among others. Weightless-P offers bi-directional support and operates in both licensed and unlicensed ISM radio bands. Common applications using Weightless-P include industrial machine monitoring and health equipment monitoring, among others.

Sigfox is another data link layer technology which offers subscription-based connectivity services over a dedicated LPWAN networks and can be used in smart alarm systems and metering systems, among others. LoRaWAN provides a low-cost and secure bi-directional long-range transmissions supporting very large networks that consist of hundreds, thousands or millions of nodes. Ideal applications of LoRaWAN include smart cities, smart street lighting and smart waste management, among many others.

ZigBee, a technology developed by the ZigBee Alliance, is based on the IEEE 802.15.4 standard and operates in unlicensed radio bands while it supports a number of topologies including the star, cluster tree and mesh topologies. ZigBee can support up to 65,000 nodes over a network and has low data rates. Table 1 provides a feature comparison for a subset

TABLE 1. Some of the data link layer protocols comparison.

Technology	Throughput *(Approximate)* *(Kbps)*	Range *(Approximate)* *(m)*	Mobility Support
NFC	424	0.1	Yes
ANT+	1,000	50	Yes
ZigBee	250	100	Yes
Z-Wave[1]	40	100	Yes
Bluetooth[3]	1,000	100	Yes
WiFi	54,000	150	Yes
WirelessHART	250	150	Yes
Weightless-W	10,000	5,000	Yes
LTE-M	1,000	11,000	Yes
LoRaWAN	0.3[4]	14,000	Yes
Sigfox[2]	0.1	17,000	Yes
NB-IoT	200	20,000	No[5]

[1] outdoor or open air; indoor is approximately 50m
[2] data rate may vary depending on the deployed region (up to 600 bps)
[3] Bluetooth 5 can support a range of approximately 150m (outdoor) with up to 8x broadcasting capacity
[4] range up to 50kbps if using Frequency-Shift Keying (FSK) instead of LoRa
[5] minimal, no full support for mobility as in LTE (possibly during cell reselection - idle state)

of the existing data link layer technologies sorted by the transmission range in an ascending order.

C. NETWORK AND TRANSPORT LAYERS

Above the data link layer as can be seen in Figure 1 is the network layer. This layer is responsible for the logical addressing and delivery of packets between source and destination, which generally requires routing. Hence, a lightweight routing process is essential for IoT devices particularly constrained devices while maintaining a high level of scalability. Because some IoT devices may operate using low-power radio communication, IPv6 Low Power Wireless Personal Area Network (6LoWPAN) provides an optimal method for transmitting IPv6 packets for low-power or constrained devices.

The transport layer is mainly responsible for the end-to-end communication via a network and provides many services such as data-stream support, reliability, multiplexing and security, among others. For connection-oriented transmissions, Transmission Control Protocol (TCP) is used for messaging transmissions. Although TCP is considered one of the most widely used protocols over the Internet, TCP may not be ideal for all types of IoT systems. An IoT system that uses a large number of constrained devices disseminated across multiple geographical areas may benefit significantly from using a connectionless service protocol such as User Datagram Protocol (UDP) for messaging transmissions versus TCP.

D. APPLICATION LAYER

The application layer is an abstraction layer that identifies a variety of protocols and interfacing methods [8]. There exists several application layer protocols that address a wide range of application requirements. Each of these protocols provide various features that vary in terms of reliability, quality of service, performance, functionality and scalability, among other factors.

Some of the protocols that exist at the application layer include: (a) HyperText Transfer Protocol (HTTP), (b) Secure HTTP (HTTPS), (c) Message Queueing Telemetry Transport (MQTT), (d) Advanced Message Queuing Protocol (AMQP), (e) Extensible Messaging and Presence Protocol (XMPP) and (f) Constrained Application Protocol (CoAP), among many others. In addition, a number of industry-specific protocols used primarily in IoT environments (mainly industrial) also exist such as Modbus, Distributed Network Protocol (DNP3), OLE for Process Control (OPC), Manufacturing Message Specification (MMS), among many others. A software developer, system designer or network administrator needs to consider not only an application protocol but also relevant protocols employed at other layers in the OSI model by an IoT system. In Section IV, we discuss in details each of the IoT messaging protocols that exist at the application layer.

TABLE 2. IoT application range requirements [120]–[123], [130]–[140].

Application	~ Range	Technology
Industry Automation	10m - 50m	LoRa, ZigBee, WirelessHART
Smart Metering	15km - 40km	LoRa, Weightless-N
Smart Buildings	10m - 250m	LoRa, Sigfox
Asset Tracking	50m - 500m	LoRa, Sigfox, Weightless
Smart Energy	100m - 15km	LoRa
Environmental Monitoring	100m - 1.5km	LoRa, Sigfox
Health Monitoring	10m - 25m	BLE, LoRa, ZigBee, ANT+
Wearable & Fitness	30m-50m	ANT+, BLE
Consumer Electronics	10m-25m	ZigBee, Z-Wave, BLE

E. IoT APPLICATION RANGE REQUIREMENTS

Because of the diversity of IoT devices, there exists no single communication technology that is capable of supporting heterogeneous environments. To have a more thorough understanding of the transmission requirement ranges for the various IoT application domains, we present in Table 2 a list of some of the common IoT application domains with corresponding ranges based on data we collected from existing research work [2]–[7], [120]–[123], [130]–[140].

F. SPECIFICATIONS, STANDARDS AND ALLIANCES

Over the past few years, there have been a significant increase in the number of standards' governing bodies and alliances that have been formed for enhancing communication technologies for the IoT landscape. Identifying the scope and goals of these initiatives is beyond the scope of this paper. However, we would like to provide the reader an overview of some of these initiatives that we believe would be relevant to this study. In Figure 2, we present a subset of the

FIGURE 2. Standards governing bodies and alliances in the IoT Landscape.

organizations and alliances that have been playing an increasingly important role in solving challenges that exist in the IoT paradigm in recent years.

The initiatives, specifications and standards presented by the various organizations and alliances shown in Figure 2 have different focuses and target specific stakeholders or markets. For example, some of these initiatives offer solutions that aim to solve challenges for Business to Consumer (B2C) or Business to Business (B2B) applications. Other initiatives were developed to accommodate specific vertical or horizontal domains in the IoT landscape. Consider, for example, organizations or alliances such as the IEEE, ZigBee Alliance, ISO, CEN and ULE. All of these organizations or alliances have proposed standards or specifications for a vertical domain which primarily focus on solving a very specific area such as home and building automation.

Other organizations such as the IEC, ISO, oneM2M, OPC and OpenIndustry 4.0 Alliance provide specifications or recommendations that are domain-specific or solve problems within the manufacturing and industrial automation vertical domain. Furthermore, organizations such as the W3C, ITU, OASIS, OMG, IETF and HyperCat [163] provide standards, specifications and recommendations that provide broader support for a number of IoT applications while encompassing many different domains (e.g. industrial, home automation, healthcare, energy, oil and gas, among many others.

G. CHOOSING MESSAGING PROTOCOLS

Although there exists a significant number of initiatives or standards that attempt to solve IoT challenges across different domains, they vary considerably in terms of the following factors: (a) architecture, (b) communication, (c) security and privacy, (d) interoperability, (e) integration, (f) device types and sensor technology, (g) deployment models (h) services' provisioning and (i) application and device management. Some of these features are also supported by a number of protocols that exist across the link and application layers. Examining similarities and differences among existing data link layer's specifications, standards or initiatives is beyond the scope of this paper. However, due to the fact that exchanging data through messages is fundamental for the development and deployment of IoT applications, we focus primarily in this paper on investigating messaging protocols that support IoT data exchange.

As a first step into identifying the protocols to be part of this research paper, we conducted a review of the existing research studies that have examined protocols for IoT applications [7], [118], [121]–[138]. As part of our selection strategy, we also considered the following factors:

- What are the system requirements and challenges that may influence choosing an application protocol for IoT development?
- What is the extent of the coverage of these challenges in existing literature?
- Which communication types are covered by existing application layer protocols?
- What factors were used or applied in conducting prior research studies?
- What is the depth of the examined literature in terms of coverage, comprehensibility and technical knowledge?
- What is the adoption rate of the existing protocols used for IoT applications?

For considering the communication scope, we reviewed the current state-of-art literature to identify the IoT communication types based on environments running IoT applications. The authors in [124] identified four classical communication types for IoT environments. We summarize these communication types below.

- Device-to-Device (D2D) where communication is provided between two nodes or devices directly.
- Device-to-Application (D2A) where communication is performed between devices and an IoT application.
- Device-to-Gateway (D2G) where communication is provided through a gateway that resides in close proximity to the edge of the network while interacting with IoT devices.
- Device-to-Cloud (D2C) where communication is achieved directly between IoT devices and cloud service providers.

We use these communication types to identify existing application layer protocols that provide support for D2D, D2A, D2G and D2C. Not all protocols support all of the identified communication types. For example, MQTT functions in the communication type D2C whereas CoAP only supports D2D. In addition, supporting the communication scope depends on a number of factors such as device capabilities. For an IoT device to send data streams to an IoT cloud platform directly (e.g. Azure IoT Hub), the device must be equipped with networking technology (e.g. WiFi or Ethernet) to send data to the cloud. However, if the device does not support direct connectivity to the cloud (e.g. an RFID tag), performing a D2C communication is not possible. Hence, a gateway may be used to collect data streams from the IoT device and then forward the stream to the cloud. As part of our comprehensive analysis and review of IoT application protocols, we take into consideration these variations and compare these protocols based on the scope of the communication type they support.

In addition, we considered published literature for identifying the adoption rate of the plurality of messaging

protocols that exist today. To this extent, we reviewed existing surveys conducted across the IoT developers' community. We make use of the results from surveys conducted by the Eclipse Foundation between 2015 and 2019 that received significant feedback from the IoT developers' community [155]. We collected statistical data about the adoption rate published by Eclipse Foundation through these surveys and used it to identify protocols that we should consider as part of our study.

Combining the data we collected from our literature review and the results of the surveys published by the Eclipse Foundation, we identify the following messaging protocols as part of our comprehensive review: (a) HTTP(S), (b) CoAP, (c) MQTT, (d) AMQP, (e) XMPP and (f) DDS. We believe that by thoroughly investigating these protocols while having general understanding of the existing data link layer protocols, it is then possible to determine how each of these protocols can address system requirements or challenges that exist across the IoT communication landscape while also identifying suitable IoT cloud service providers for deploying crucial IoT services.

H. SUPPORT FOR IoT MESSAGING PROTOCOLS ON THE CLOUD

Apart from examining adoption rates for IoT messaging protocols, it would be desirable to identify any patterns or trends for protocols that are supported by existing IoT cloud platforms. To this extent, we conducted a literature review examining support levels of messaging protocols across the plurality of existing IoT cloud platforms [166]–[175]. Table 3 presents a detailed list of supported protocols that we gathered for ten of the major players or the plurality of IoT cloud platforms that exist today.

As can be seen in Table 3, all of the IoT cloud platforms support HTTP(S) and MQTT across all ten cloud platforms, followed by AMQP supported by six platforms, then CoAP supported by four IoT platforms. These results align with those of the Eclipse Foundation survey noting that HTTP and MQTT constituted the top two messaging protocols used by developers according to the survey results [155]. We discuss the results of the survey in Section VI.I. Unfortunately, none of the existing IoT cloud platforms provide support for DDS although it is being used by IoT developers based on the results gathered from the survey conducted by the Eclipse Foundation (Section VI.I).

We summarize the distribution of the support by cloud platform providers in Figure 3. Figure 4 presents a comparison of the messaging protocol support across cloud platforms examined in Table 3. As can be seen in Figure 4, Oracle IoT and Siemens MindSphere provide support for all of the five protocols, followed by Eclipse Hono.

In addition to supporting IoT messaging protocols, some IoT service providers also provide support for proprietary protocols or protocols that are specific at the IoT device level. For example, Siemens' MindSphere supports Modbus, Lightweight M2M (LwM2M), LoRaWAN and OPC UA technologies which are commonly used in industrial automation, manufacturing or home automation. Although Alibaba IoT provides support for two of the protocols that we are investigating, it also supports IoT device mobility through dedicated

TABLE 3. List of protocols/technologies supported by existing IoT platform providers for data transmission.

IoT Platform, *Year of First General Availability (GA)*	Protocol
Azure IoT Hub [166], *2014*	HTTP(S), MQTT, MQTT over WebSocket, AMQP, AMQP over WebSocket *custom protocols transmit via a gateway*
Google IoT Core [167], *2018*	HTTP(S), MQTT *custom protocols transmit via a gateway*
IBM Watson IoT [168] *2014*	HTTP(S), MQTT
AWS IoT Core [169] *2015*	HTTP(S), MQTT, MQTT over WebSocket, WebSocket
Alibaba IoT [170] 2015	HTTP(S), CoAP, MQTT, MQTT over WebSocket, WebSocket, *support network types: 3G, 4G, NB-IoT & LoRa*
Oracle IoT [171] *2016*	HTTP(S), CoAP, MQTT, AMQP, XMPP, WebSocket
Siemens MindSphere [172] *2016*	HTTP(S), CoAP, MQTT, AMQP, XMPP, *supports wide range of device protocols via field gateways (e.g. MindConnect) such as OPC UA, LoRaWAN, Modbus, 6LoWPAN, LwM2M*
Bosch IoT Hub [173] *2017*	HTTP(S), MQTT, AMQP, LoRaWAN
Cisco Kinetic [174] *2017*	HTTP(S), MQTT, AMQP, WebSocket *custom protocols transmit via a gateway (e.g. Cisco IoT Gateway)*
Eclipse Hono [175] *2018*	HTTP(S), CoAP, MQTT, AMQP *uses AMQP 1.0 as primary messaging protocol custom protocols transmit via a gateway*

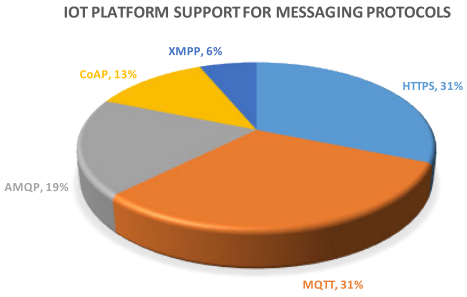

FIGURE 3. IoT platform support distribution.

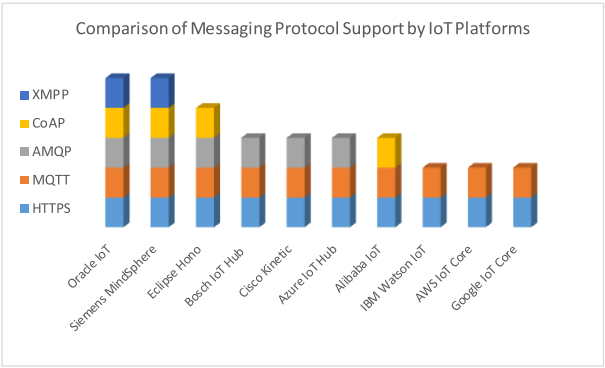

FIGURE 4. Comparison of messaging protocol support by IoT platforms.

transmission channels on its IoT platform providing support for a number of cellular network types including 2G, 3G, 4G and NB-IoT.

Apart from the complexities in choosing an appropriate messaging protocol for IoT applications, results from Figures 3 and 4 show that considering the IoT deployment platform is also important. In cases where an IoT application has high levels of heterogeneity, choosing an IoT platform that supports diverse IoT messaging protocols would be a good choice. However, there are also other factors that may influence the choice of IoT cloud platforms such as cost, bandwidth, reliability, security, service provisioning, interoperability, resiliency and composability, among others.

In addition, the degree of the services offered by existing IoT cloud service providers varies depending on the technologies used or applied. For example, IBM offers hosting services for its SoftLayer data centers [141] which support a large magnitude of data streams (i.e. in billions) from connected IoT devices. Deploying an IoT application on GCP, for example, can take advantage of Google's fiber optics network [142] which can significantly support low-latency IoT application deployments. Microsoft Azure offers a number of services through the Azure Suite running applications such as Salesforce, SAP, Oracle database and Microsoft Dynamics. AWS enables messages to be routed to AWS endpoints through the Rules Engine to AWS's Lambda, Kinesis, machine learning and Elasticsearch service, among many others. In the following sections, we describe the various messaging protocols that exist at the application layer that can be used for developing and deploying IoT applications.

IV. MESSAGING PROTOCOLS FOR THE INTERNET OF THINGS

In this section, we primarily focus on the application layer protocols that support the data link layer protocols discussed in Section III. We applied a selection strategy based on the adoption rates of the application layer messaging protocols from a survey conducted by the Eclipse Foundation [155] as discussed in Section III.G. In addition, we considered the support level of messaging protocols by existing IoT cloud platforms as discussed in Section III.H.

A. HYPERTEXT TRANSFER PROTOCOL (HTTP)

HTTP is an application-level, generic, stateless protocol [9] that is used generally for data communication over the World Wide Web. One of the key features of HTTP is content negotiation of data representation. This enables different heterogeneous devices built independently of the data to be shared [8]. HTTP is a request-response protocol in which a client (e.g. a browser) sends a request message and a host (e.g. a server) generates a response message. HTTP version 3.0 or H3 is the latest (draft) version of HTTP introduced in 2018 [10]. However, the common HTTP version used today is HTTP 1.1. Figure 5.0 presents variations among HTTP versions 1.0, 1.1 and 2.0. In this paper, we focus primarily on the common HTTP version used in IoT applications or HTTP 1.1.

B. CONSTRAINED APPLICATION PROTOCOL (CoAP)

Constrained Application Protocol (CoAP) is a web transfer protocol that is intended for devices running on constrained (e.g., low-power, lossy) networks [11]. Constrained devices generally have 8-bit microcontrollers with small amounts of memory. The CoAP standard was designed for Machine-to-Machine (M2M) applications (e.g. factory automation, smart energy). Similar to HTTP, CoAP is a request-response interaction model and uses major concepts from the web such as Uniform Resource Identifiers (URIs) and Internet media types [11]. CoAP aims to bridge HTTP and RESTful services through simple interfacing. This protocol is used over the UDP transport protocol using the *coap* scheme and over DTLS using the *coaps* scheme [11]. An abstract layer of a DTLS-Secured CoAP is shown in Figure 6.

A request URI can be split into multiple parts: (a) Uri-Host, (b) Uri-Port, (c) Uri-Path and Uri-Query Options. These features provide easy and intuitive way to communicate using this protocol with RESTful services.

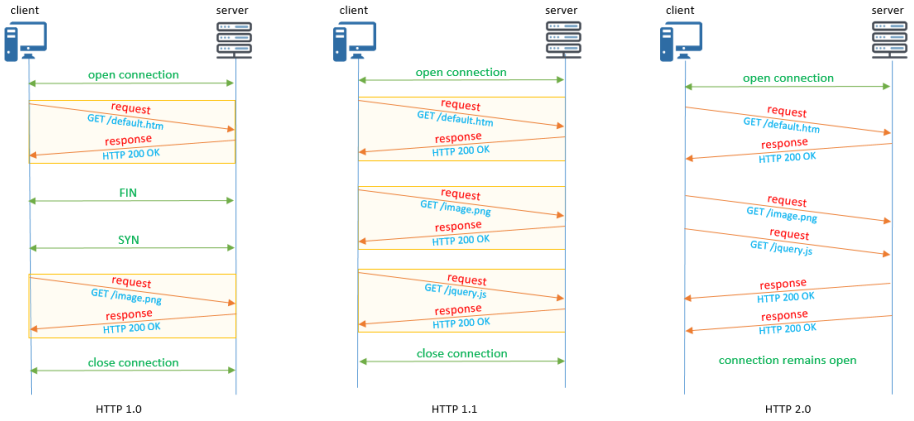

FIGURE 5. HTTP message control flow (left: HTTP 1.0, middle: HTTP 1.1, right: HTTP 2.0.

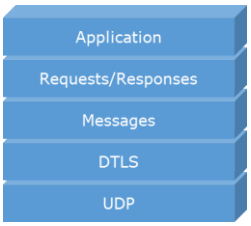

FIGURE 6. Layers of DTLS-Secured CoAP [11].

TABLE 4. CoAP methods.

Method	Safe	Idempotent	Description
GET	✓	✓	retrieves a representation of a valid resource
POST	×	✓	processes a representation of a given request
PUT	×	✓	update/create a resource identified by a request URI
DELETE	×	✓	deletes a resource identified by request URI

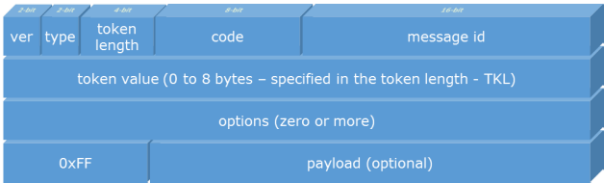

FIGURE 7. CoAP message structure.

CoAP supports CRUD operations through HTTP methods as shown in Table 4 and provides status codes that are very similar to those in HTTP.

CoAP also provides support for discovering services and resources. In addition, when a message is published in a URI, notifications are sent to clients who can access the resource. An example of a CoAP message format is shown in Figure 7. The version specifies the CoAP version number and the type indicates the type of message. CoAP uses a simple binary-base header format and the smallest size is 4 bytes.

Table 5 outlines the four different message types that CoAP support. The message types provide a certain guarantee to the delivery of messages and hence increases the level of quality overall. For example, when a server processes a response identified by a given code (in the code field) matching the request, an 8-bit response code identifies the class of the response: (2) success, (4) client error and (5) server error.

A confirmable message (CON) represents reliable message delivery since this type of message is retransmitted one or more times until the server ultimately receives the message. The ACK message will contain the same message id of the confirmable message (CON). An example of a confirmable message with an acknowledge response message is shown in Figure 8.

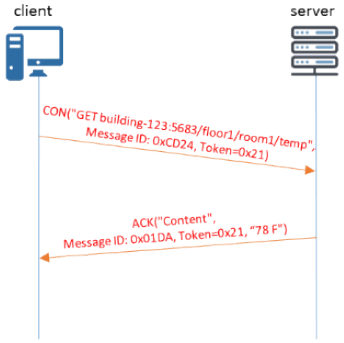

FIGURE 8. CoAP message control flow.

Through the message types and HTTP-like status codes, CoAP becomes an easy to use protocol for integrating into the existing web. IoTivity, an open source software framework that enables device-to-device communication, uses CoAP as its application layer [18]. In addition, CoAP offers extensions such as the Observe Option which helps in observing the changes in the state of resources giving it a RESTful architecture support.

C. MESSAGE QUEUING TELEMETRY TRANSPORT (MQTT)

Although HTTP and CoAP can be used as a request/response protocols used by IoT systems or devices, there is a need for a lightweight protocol that is designed to handle situations in cases of unreliable networks or intermittent connectivity. MQTT, an OASIS standard, is a publish-subscribe lightweight messaging protocol designed for constrained devices [12] and is well-suited for these types of scenarios while enabling the exchange of data with the cloud in a real-time manner.

As outlined in Table 3, nearly all of the existing IoT cloud platform providers including IBM Watson IoT Platform [13], Microsoft Azure IoT Hub [14], Google Cloud

TABLE 5. CoAP message types.

Message Type	2-bit	Description
Confirmable	00	an acknowledgement is required (enhances reliability of message delivery over UDP)
Non-Confirmable	01	processes a representation of a given request
Acknowledgment	10	update/create a resource identified by a request URI
Reset	11	deletes a resource identified by request URI

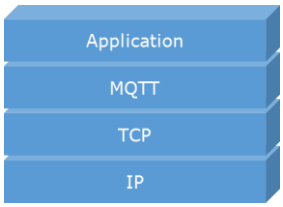

FIGURE 9. MQTT protocol based on TCP/IP stack.

IoT Core [15], Bosch IoT Hub [16] and AWS IoT [17], among many others provide support for an IoT application to send data using the MQTT protocol. MQTT, an application layer protocol designed for Machine-to-Machine (M2M) communication, uses a publish-subscribe model and runs on top of the Transmission Control Protocol/Internet Protocol (TCP/IP) as shown in Figure 9.

MQTT is considered lighter than HTTP 1.1 and supports a near real-time message exchange using the publish-subscribe model. In addition to being a protocol of choice for many IoT and M2M applications, MQTT is also used in various applications where data exchange is necessary such as asset management, automotive telematics, traffic monitoring, home automation and supervisory control and data acquisition (SCADA), among others.

This publish-subscribe model is composed of a broker (i.e. a server) and clients establish a connection with the broker at any time. In this model, clients send messages through the broker which is known as the publisher. Then, the broker filters these incoming messages and distributes them to clients who are interested in receiving these messages. To this extent, a client that registers with the broker to receive these messages must first subscribe to specific topics. Clients receive the payload (or message data) when a new device publishes a message. A subscriber can receive a published message later. That is, the subscribers can receive published messages at different times. Figure 10 presents a high-level overview of the MQTT brokering model that shows all of the entities involved in this architecture including: (a) centralized broker, (b) publishers and (c) subscribers.

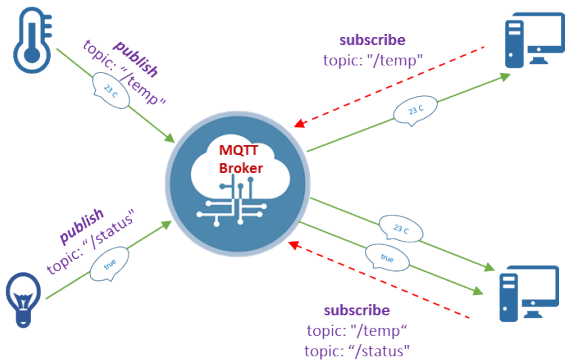

FIGURE 10. MQTT centralized broker model.

In this publish-subscribe model, a publisher can send messages to a number of subscribers with one single publish

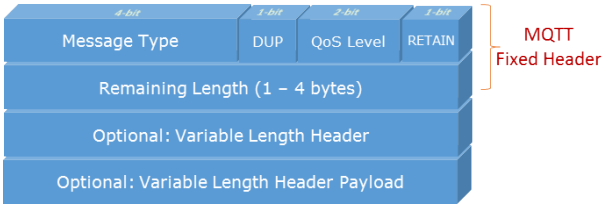

FIGURE 11. MQTT message structure.

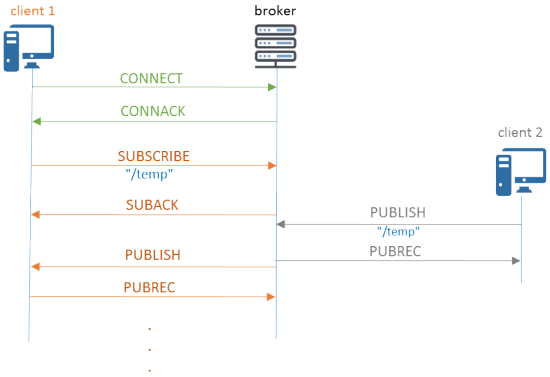

FIGURE 12. An example of a MQTT message control flow.

operation to the broker. The broker handles the "broadcasting" or sending messages to all subscribers subscribed to topic of the message as shown in Figure 10. There are a number of MQTT model implementations including Mosquitto, eMQTT, HiveMQ, Moquette, among many others. Figure 11 presents the MQTT protocol message format. Figure 12 depicts an example of an instance of a message flow in the MQTT protocol.

In the message control flow shown in Figure 12, client 1 wishes to connect to the broker (CONNECT) where the broker accepts the request (CONNACK). Client 1 then subscribes to topic "temp" (SUBSCRIBE) and the broker responds with an acknowledgement for this subscription (SUBACK). Client 2 publishes to the broker message on the topic "temp" for which the broker then will publish it to Client 1 (PUBLISH) who is already subscribed to the "temp" topic. The message contains the data or payload. Client 1 then acknowledges the receipt of the published message (PUBCREC).

Message reliability is imperative in this publish-subscribe brokering model. That is, an IoT system may require that messages be delivered in a reliable manner where all clients acknowledge the receipt of these messages. MQTT supports a Quality of Service (QoS) level when messages are published through the Connect Flags of the Fixed Header using the Will QoS parameter (bits 4 and 3). These bits specify the QoS level applied when a broker publishes a message to one or more clients. MQTT supports three levels of QoS as presented in Table 6.

As shown in Table 6, as the QoS level increases, the reliability of messages' delivery also increases. However, this also increases the overhead associated with ensuring that

TABLE 6. MQTT message reliability and QOS levels.

QoS Level	Reliability Level	Description
QoS 0	lowest	at **most once** delivery - no response is sent by receiver, no retry is performed by sender - message arrives at the receiver either once or not at all
QoS 1	medium	at **least once** delivery - ensures that a message arrives at the receiver at least once - receiver may send further PUBLISH packets while waiting to receive acknowledgments
QoS 2	highest	**exactly once** delivery - ensures that a message arrives at the receiver exactly once without duplication - increased overhead associated with this level

all clients receive the intended messages. The more clients are subscribed to receive a message with QoS 2, for example, will increase the overhead on the message broker while ensuring the delivery of the message without duplication or retransmission.

Nearly all of the existing cloud platform providers support both HTTP(S) and MQTT. Table 7 provides a feature summary of the Quality of Service (QoS) levels supported by some of the popular IoT cloud platforms. Because MQTT is widely used across a number of M2M applications including home appliances, smart utility management, waste control and logistics, among many others, attackers may exploit vulnerabilities within the protocol to carry out targeted attacks (e.g. denial-of-service (DoS) attacks) or collect proprietary data tunneled through unsecure MQTT transmission channels.

TABLE 7. MQTT QOS support by some IoT cloud service providers.

IoT Provider	HTTP(S)	MQTT QoS Levels		
		QoS 0	QoS 1	QoS 2
Azure IoT Hub	✓	✓*	✓	×
Google IoT Core	✓	✓*	✓	×**
IBM Watson IoT	✓	✓*	✓	✓
AWS IoT Core	✓	✓*	✓	×***

* *signifies default option by service provider platform.*
** *publish at QoS 2 closes connection; subscribe at QoS 2 downgrades to QoS 1.*
*** *devices can publish at QoS 2 but device management server uses pub/sub at QoS 1.*
**** *no PUBACK or SUBACK is sent when QoS 2 is requested (i.e. ignored).*

The MQTT handling of disallowed Unicode code points provides a client or server the option to decide on the validation of these code points (e.g. UTF-8 encoded strings). As a result, an endpoint does not necessarily need to validate UTF-8 encoded strings (e.g. topic name or property). As such, a client could potentially use this as a vulnerability and causes a subscribed client to close the network connection using a topic that contains an invalid Unicode code point. A malicious client can then use this as a security exploit for possibly causing a Denial of Service (DoS) attack. Therefore, enabling UTF-8 encoded strings, for example, can allow these security exploits to occur in cases they are used as control characters or in control packets (see the first byte in Figure 11). Hence, all clients will remain offline and not acknowledge to the MQTT broker if a malicious client sets the QoS level at 2 while retaining the message.

D. ADVANCED MESSAGE QUEUING PROTOCOL (AMQP)

AMQP, both an OASIS and ISO standard, is another lightweight but an extensible messaging protocol designed for M2M messaging [19], [44], [45]. AMQP is used generally in corporate environments and focuses on interoperability [20]. AMQP is a binary, application layer protocol that supports a wide range of messaging applications and communication patterns. In addressing interoperability issues when using Message-Oriented Middleware (MOM), AMQP supports both request-response and publish-subscribe models. Similar to MQTT, it supports a topic-based publish-subscribe messaging. In addition, AMQP provides support for flexible routing and business transactions. Figure 13 presents an AMQP message structure which includes header, message and delivery annotations.

FIGURE 13. AMQP message structure.

AMQP provides two message interaction modes: (a) browse mode and (b) consume mode. In the browser mode, a client can lookup stored messages in a specified queue whereas in the consume mode, a client can consume messages within a queue. These consumed messages are then deleted from the queue. AMQP's message distribution is composed of three components: (a) exchange, (b) binding and (c) queues.

Figure 14 illustrates the message distribution model for AMQP. Through this model, AMQP provides multiple levels for exchanging and delivering messages. An exchange is represented by a routing agent that runs on a virtual host residing on a broker's server. This routing agent accepts messages and then routes them internally or forwards them to appropriate message queues along with a routing key. An exchange type applies a specific matching criterion and a corresponding algorithm.

A message queue is a named FIFO buffer that stores messages on behalf of applications temporarily. Such applications

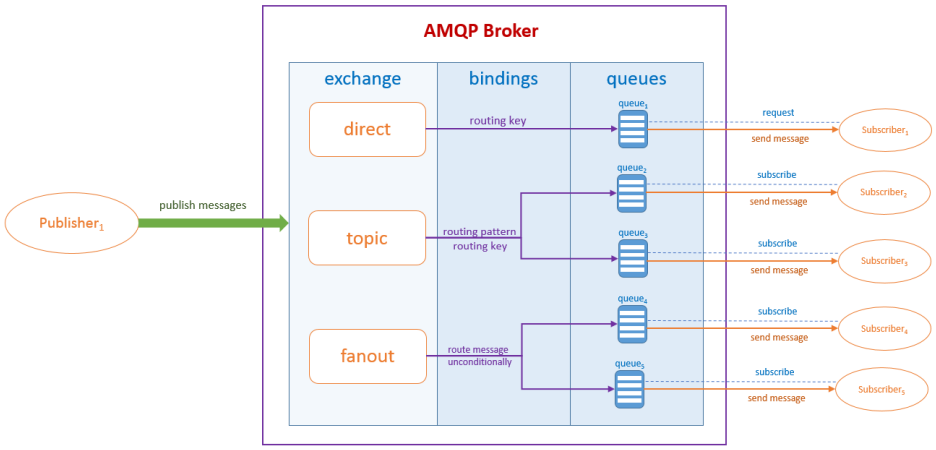

FIGURE 14. AMQP message distribution model.

can, for example, create, share, use or delete messages depending on the granted authorizations. Bindings represent relationships between message exchanges and message queues. In specific, bindings route messages between exchanges to the appropriate queues depending on the message exchange type. Exchanges can be classified as either durable or transient. Durable message exchanges are persistent and continue their operations in case a broker needs to restart while transient exchanges are temporary. Restarting transient exchanges is required after a broker has restarted. Generally, there are four main message exchange types: (a) direct, (b) topic (c) fanout and (d) header. We describe each exchange type in the following sub-sections.

1) Direct Exchange Type

Using the direct exchange type, an exchange instance forwards incoming messages to queues based on the routing key associated with each message. Each binding contains a binding key. When a publisher sends a message to the direct exchange, it must provide a routing key. If the binding key from the message queue is identical to that of the publisher's message routing key (i.e. exact matching), the message then passes through a message queue.

To illustrate the direct exchange type, consider, for example, a publisher that sends a message using direct exchange. In this case, the message published by the publisher will be routed to the queue that is directly bound to this exchange. This example is represented in Figure 14 between Publisher 1 and Subscriber 1 exhibiting a direct exchange type. Direct exchange type is often used to represent cases for point-to-point messaging. In cases the binding key is associated with more than one queue, this exchange type can be used for multicasting operations.

2) Topic Exchange Type

The topic exchange type is similar to that of direct exchange except that the routing key is considered as a routing pattern (i.e. a topic). Unlike direct exchange, the routing key is fixed whereas the routing pattern in the topic exchange allows the use of wildcards. That is, it is possible to identify a binding key pattern to be matched against a routing key. The routing keys can be matched for one or more keywords. Pattern matching characters include * and #.

The * is used to match a single keyword whereas the # is used to match zero or more keywords. Each keyword is delimited by a "." (or a period). Using topic exchange type, when a publisher sends a message to the topic exchange, it provides a routing key. The message then passes to the message queue if the routing pattern matches that of the routing key. An example of topic exchange is represented in Figure 14 between Publisher 1 and Subscribers 2 and 3 since the routing key is matched by some binding key pattern. The topic exchange type is typically used to represent a publish/subscribe messaging pattern.

3) Fanout Exchange Type

The fanout exchange type does not require routing keys for binding messages to queues. That is, it "broadcasts" messages to all subscribers unconditionally. Using the fanout exchange type, when a publisher sends a message to the fanout exchange instance, the exchange will simply forward the messages to all of the queues that are bound to it unconditionally without the need of any topic mapping (i.e. a pattern) or subscribed access (i.e. routing key). An example of topic exchange is represented in Figure 14 between Publisher 1 and Subscribers 4 and 5 where the exchange forwards the message to all subscribers without any verifications or matching conditions. The fanout exchange type is generally used when a broker needs to asynchronously broadcasts event notifications to all endpoints.

4) Header Exchange Type

In this exchange type, routing keys and patterns are ignored. An exchange binds to a message queue based on a list of arguments or properties specified in the header of a message. A message is routed

to a queue when header properties agree with a "`x-match-expression`". The "`x-match`" supports logical both AND and OR matching based on defined list of header properties. Using this exchange type, a publisher sends a message to the exchange matching the properties in the header based on a collection of name/value pairs. A message is then routed based on the header property matching the arguments for binding. The header exchange type is very similar to service binding that occurs in HTTP request/response paradigm.

In addition to the exchange types, AMQP also provides a flow-controlled communication in routing messages with message-delivery guarantees including: (a) at most once where a message is delivered once or not at all, (b) at least once where a message is indeed delivered once but may be received multiple times and (c) exactly once where a message will be guaranteed to be delivered only once without loss or redundancies. Furthermore, AMQP supports message QoS using two modes: (a) unsettled (not reliable) and (b) settled (reliable). In terms of security, AMQP supports authentication and/or encryption based on SASL and/or TLS and uses TCP as the underlying transport layer protocol [20].

In terms of interoperability, AMQP can run with other transport protocols such as Stream Control Transmission Protocol (SCTP). This interoperability is exhibited by the Apache Qpid which aims to provide an AMQP stack implementation in multiple languages including C, Java and C++. In addition, Microsoft provides an AMQP.NET lite library for supporting AMQP messaging queues in the. NET framework. Both the Microsoft Azure Service Bus and the Azure IoT Hub support communication using AMQP. IBM also provides support for AMQP through its MQLight, a lightweight messaging API. Amazon provides support for message queuing through the Amazon Simple Queue Service (SQS).

E. DATA DISTRIBUTION SERVICE (DDS)

The Data Distribution Service (DDS) is a middleware protocol and an API standard that was developed by the Object Management Group (OMG). It is a publish-subscribe model [21] but follows a data-centric approach for data sharing through what is known as a global data space. DDS's brokerless architecture makes it very suitable for M2M communication. The messaging model in DDS consists of two interface layers: (a) Data-Centric, Publish-Subscribe (DCPS) and (b) Data Local, Reconstruction Layer (DLRL).

The DCPS layer is primarily responsible for binding the values of data objects within an application during the publish/subscribe process. That is, it enables a publishing application to associate data objects with values that require publishing. Furthermore, it enables subscribing applications to identify data objects of interest and ways for accessing their values. Similar to other messaging protocols such as MQTT and AMQP, DDS provides a way to define topics of interest. Additionally, through the DCPS interface layer, DDS enables publishers and subscribers to associate Quality of Service (QoS) policies with publisher and subscriber entities [46]. The DLRL, an optional layer in DDS, acts as a connector for integrating DDS at the application level for interfacing with other external entities (e.g. other protocols, applications, among others).

We focus primarily on describing the core layer of DDS, the DCPS. There are two main constructs in DCPS: (a) publisher with DataWriter and (b) subscriber with DataReader. A publisher uses the DataWriter to bind values of data objects per a defined data type. A publisher is responsible for the data distribution while adhering to a list of predefined QoS policies, if any. When an application needs to write data associated with data objects, the DataWriter describes the data that requires publishing.

A subscriber receives published data while making it available, based on a predefined set of QoS policies, to an application. To retrieve data from a subscriber, an application needs to use a DataReader attached to a subscriber. An application subscribes to data described by a DataReader that is supplied by a known subscriber. The relation in which the process of publishing and subscribing to data objects is achieved through topics.

A topic is associated with a (a) name, (b) data-type and (c) (optionally) one or more QoS policies related to the data [46]. The QoS policies are often used to control how data should be distributed in this messaging model. DDS provides twenty-three different QoS levels dealing with various features such as security, priority, durability and reliability, among many others. Figure 15 presents a high-level overview of the communication model of the DCPS layer in DDS.

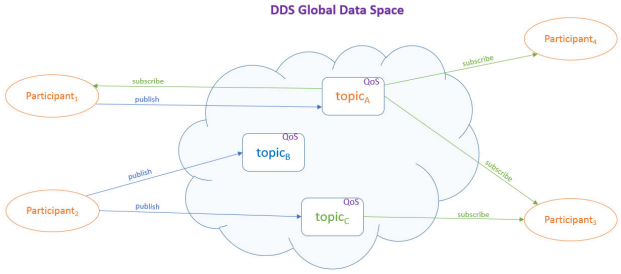

FIGURE 15. DDS data-centric model.

Through this communication model, DCPS entities are associated with DomainParticipants enabling applications to participate in one or more defined domains. Subscribers and publishers are linked through this communication plane such that ones that are joined in the same domain can interact. Through this approach, publishers and clients are time-decoupled which helps reduce the overall latency, bandwidth and processing power consumption. In addition, it enables publishers and subscribers to have their own networks of communication and hence enabling real-time interactions between these entities (i.e. D2D).

The data model of DDS stems from the relational data model. In the DDS middleware protocol, it treats data similar

to a relational database managing both the: (a) structure (or schema) and (b) queries and filters on content when data is requested. DDS allows publishers and subscribers to dynamically discover each other without servers thus directly controlling data flows between DDS entities. In addition, DDS supports a number of trigger patterns for maintaining updates on subscribed content.

Furthermore, DDS supports interoperability among various vendor implementations through the Real-Time Publish Subscribe (RTPS) wired protocol [22], [46], [47]. A Security Model extends the security capabilities of DDS which is enforced by the Service Plugin Interfaces (SPIs). Through SPIs, DDS provides a wide-range of support for out-of-the-box security plugins and connectors that enable the communication between multiple DDS applications [48]. DDS's support for interconnectivity among applications makes this middleware protocol an excellent choice for heterogeneous IoT applications that require communication in real-time.

F. EXTENSIBLE MESSAGING & PRESENCE PROTOCOL (XMPP)

The Extensible Messaging and Presence Protocol (XMPP) is an open standard protocol used for building real-time applications and uses a wide-range of service communication technologies such as instant messaging, multi-party chat, voice and video calls, collaboration, lightweight middleware and generalized routing of XML data [23]. Originally named as Jabber, this client/server architecture was initially designed to enable applications to provide instant messaging capabilities.

XMPP uses XML as the underlying data exchange format and runs over TCP/IP. XMPP provides a set of essential services called XMPP Core Services and extensible services called XMPP Extension Protocols (XEPs) which aim to extend XMPP's core services. Some examples of XEPs includes the Bidirectional-streams Over Synchronous HTTP (BOSH), which offers HTTP binding for XMPP traffic. Currently, over 350 IETF-governed specifications that extend the core of XMPP are part of the XEP Series [23]. One of the key XMPP extension protocols is the PubSub XEP which enables XMPP to support the publish-subscribe model. Through PubSub XEP, it is then possible to use XMPP as a messaging protocol for IoT systems.

The PubSub is a protocol extension of the core XMPP that allows XMPP entities to create topics (nodes) and publish information at those nodes. Subscribed entities will then receive notifications either with or without payload. XMPP entities are associated with JIDs (Jabber IDs) when running over a network. A JID is in the form of an email address with a fully qualified domain name and/or a valid resource (e.g. `xmpp_user@xmpp_server/resource`) where `xmpp_user` is the client's username, xmpp_server is a fully qualified domain name and `resource` is an identifier used to identify the client's device on the network. A *bare JID* is an address that is without the resource whereas a JID that includes a resource identifier is referred to as *full JID*. Multiple resources (e.g. full JIDs) can be associated with one username indicating different devices used or associated with the same ''account'' or user.

XMPP provides point-to-point encryption (TLS) that is built-in within the core specification. In addition, given that XMPP uses XML, which is text-based, this translates into higher network overhead compared to the binary encoding protocols such as CoAP, MQTT, AMQP and DDS. Furthermore, XMPP uses open-ended XML streams via TCP and supports relatively small-sized XML data units called XML stanzas. Table 8 presents the steps involved when sending XML stanzas between nodes using XMPP [23].

TABLE 8. Process of XML stanzas exchange in XMPP.

Step	Description
1	identifying the receiver's IP address and port using a fully qualified domain name
2	establish a TCP connect
3	open an XML stream via TCP
4	negotiate TLS for channel encryption [optional]
5	authenticate using SASL
6	bind a resource to a stream
7	exchange unbound number of XML stanzas with other entities on the same network
8	close an XML stream
9	close the TCP connection

Clients in XMPP send stanzas to exchange messages and it follows a fire-and-forget mechanism (e.g. no guarantee a receiver receives the stanza). A stanza advertises its status (or network availability) to other XMPP entities. This is based on a subscription model in which entities A and B, for example, subscribe to a presence stanza of entity C. When entity C is present online (via the presence stanza), the XMPP server will update all other subscribed entities (in this case A and B) with the presence status of entity C.

Unlike CoAP, AMQP and DDS, XMPP does not allow device-to-device communication. However, XMPP servers can form a federation where each server acknowledges the existence of other servers over the same network. This can be used for dynamic resource discovery in IoT systems. In addition, XMPP supports clustering in which multiple servers within a single domain can form a cluster. Clustering can support interoperability between various systems running XMPP. Figure 16 presents an illustrative example of using XMPP for building an IoT smart home system. Each IoT device (e.g. room light) is associated with a full JID.

V. COMPREHENSIVE COMPARISON OF MESSAGING PROTOCOLS FOR THE INTERNET OF THINGS (IoT)

When developing IoT systems, choosing the most appropriate Messaging Protocols becomes a challenging task. To identify

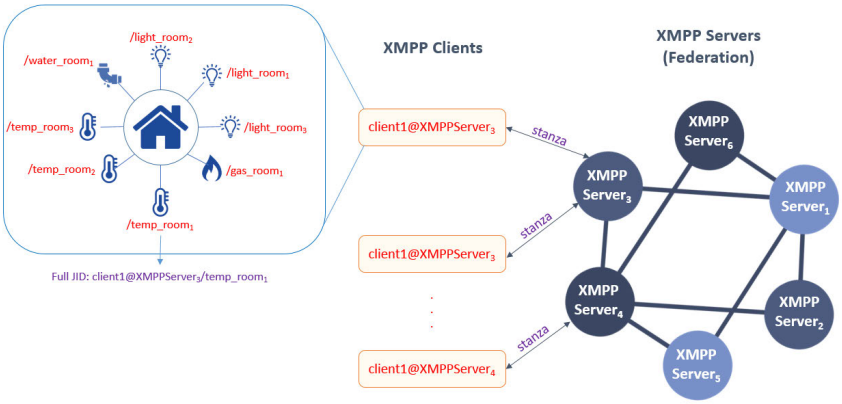

FIGURE 16. An illustrative example of a XMPP-based IoT smart home system architecture.

a suitable protocol requires sufficient background knowledge on how these protocols operate. Due to the fact that these protocols are critical for the interconnectivity among entities in an IoT system, the functioning and overall performance of the system can significantly be impacted by the type of protocol employed. For example, choosing HTTP for low latency or mission-critical IoT systems may not be an ideal choice for a messaging protocol for data connectivity. In addition, choosing a protocol that does not enforce an acceptable degree of quality of service (QoS) such as XMPP while ensuring message delivery guarantees is not an ideal choice for real-time IoT systems.

Although all messaging protocols are used for data communication between two or more entities via some transmission medium, some of the characteristics of each protocol vary. For example, HTTP uses a request/response messaging pattern suitable for a client/server architecture. Low-power or constrained IoT devices may not be able to cope with using this messaging pattern since HTTP is a connectionless and stateless protocol. Hence, when choosing a messaging protocol for building IoT systems, it is essential to consider protocol characteristics and identify ways to which it can help overcome challenges that may arise in cases such as reliable message delivery or scaling up.

Messaging protocols have similarities and differences among a number of properties or characteristics. When building or deploying an IoT system, it is crucial to consider the protocol characteristics to which it can meet functional and operational requirements. To this extent, we provide in Table 9 a detailed comparison of the characteristics such as messaging pattern, network layer support, transactional support, message caching, dynamic discovery, filtering, encoding types and implementation support among others. In addition, we considered the basic elements that make up a communication model such as source, transmitter, receiver, destination and transmission system. We also considered key communication tasks such as interfacing, exchange management, addressing, routing, security, network management, fault tolerance, error detection, synchronization and protocol representation, among many others.

VI. DISCUSSION

Internet of Things (IoT) systems are driven by IoT devices that are typically resource-constrained having limited power, networking and processing capabilities. These low-bandwidth devices are often equipped with wired or wireless radio technologies that enable them to transmit data and receive instructions. To this extent, it is imperative that the messaging protocols employed in IoT systems maintain high-levels of quality for data transmission.

In addition, messaging protocols need to be optimized such that they require minimal resources (e.g. processing power, memory, storage, network bandwidth) which are often needed by IoT devices when communicating data or receiving control signals. The degree to which messaging protocols can offer the anticipated levels of quality will likely to vary since many of the existing protocols were developed by different organizations and designed for different purposes. Some protocols were designed for corporate environments (e.g. AMQP) while others were designed for mission-critical systems (e.g. DDS). This makes the task of identifying which protocol is suitable to use in IoT systems a challenging task.

Given the fact that IoT systems are heterogeneous, supporting more than one protocol may be an option. Hence, it is critical to identify those messaging protocols that are extensible. However, extensibility may yield to an extra overhead on part of IoT devices. In this case, it would be desirable to identify protocols that are not only extensible but run on constrained environments. The choice can become easy if there exists a protocol that can accommodate these two factors: extensibility and support for devices running in constrained environments. The problem, however, is that without considering the various degrees to which what each protocol can offer IoT systems, choosing a single suitable protocol is unlikely to be an easy design decision.

To identify the key similarities and differences that exist among communication protocols, we determine key features that need to be considered when choosing an application layer protocol for interfacing. Such features relate mainly to challenges that occur when designing and maintaining services in IoT systems. In addition, our selection strategy

TABLE 9. Comparison of existing communication protocols' characteristics.

Feature	HTTP	CoAP	MQTT	AMQP	DDS	XMPP
Year Introduced	1991	2013	1999	2003	2001	2002
Standardized	1997	2014 (*ongoing*)	2013	2014	2004	2004
Messaging Pattern	request/response	request/response	publish-subscribe	request/response; publish-subscribe	publish-subscribe	request/response; publish-subscribe
Architecture	client/server	tree	tree	star	bus	client/server
Transport	TCP	UDP	TCP (MQTT-S: UDP)	TCP	UDP or TCP	TCP
Network Layer	IPv4 or IPv6	IPv6	IPv4 or IPv6	IPv4 or IPv6	IPv6	IPv4 or IPv6
M2M Communication	×	✓	✓	✓	✓	✓
Asynchronous Messaging	×	✓	✓	✓	✓	✓
Transaction Support	×	×	×	✓	✓	×
Extensibility	×	×	×	✓	✓	✓
Data Prioritization	×	×	×	✓	✓	✓
QoS Support	×	✓	✓	✓	✓	×
Message Caching	✓	✓	✓	✓	✓	✓
RESTful	✓	✓ Observe Option	×	×	×	×
Dynamic Discovery	×	✓	×	×	✓	✓
Content Awareness	×	×	×	×	✓	×
Distributed Tracing	✓	×	×	×	✓	×
QoS Levels	*(based on TCP)*	2 Levels	3 levels	3 levels	23 levels	none
Communication Scope	Device-to-Cloud; Cloud-to-Cloud	Device-to-Device	Device-to-Cloud	Device-to-Device; Device-to-Cloud; Cloud-to-Cloud	Device-to-Device; Device-to-Cloud; Cloud-to-Cloud	Device-to-Cloud; Cloud-to-Cloud
Addressing	URI	URI	topic only	queue, topic/ routing key	topic/key	Jabber Identification
Filtering	Resource Identifier	Resource Identifier	topic	queue	topic, time, content	{user: to, from}, type, iq, presence packets
Fault Tolerance	Server is SPoF	Decentralized	Broker is SPoF	Broker is SPoF	Decentralized	Server is SPoF *(node can be a server)*
Security	HTTPS over TLS	DTLS, IPSec	TLS	SASL/TLS	TLS, DTLS, DDS Security	SASL/TLS
Interoperability	Semantic	Semantic	Foundational	Structural	Semantic	Structural
Header Size	undefined *	4 byte	2 byte	8 byte	16 byte (RTPS)	variable
Data distribution	1-to-1	1-to-N; N-to-1;	1-to-N; N-to-N;	1-to-1; N-to-N	1-to-N; 1-to-1; N-to-1; N-to-N;	1-to-1; N-to-N
Encoding	Text	Binary	Binary	Binary	Binary	Text (binary data out-of-bound)
Low-Power and Lossy (1000s)	Fair	Excellent	Good	Good	Poor	Fair
Payload Format	unclear	JSON, XML	unclear	unclear	Strongly Defined Types, Mixed	XML
Max. Message Size	undefined	64KB *(UDP)*	256MB	undefined; RabbitMQ: 512MB [93]	64KB *(UDP)* [96]	Undefined; 64KB (stanza size)
[Broker/Server] Implementation Languages	C, C++, C#, PHP, Elixir/Erlang, Java, Scala, Lisp, Lua, Node.js, Dart, Rust, Swift, Ruby, Go, Python, Haskell, Perl	Java, C, C++, C#, Erlang, Go, Python, JavaScript, Ruby, Rust, Swift	C, C++, C#, Java, JavaScript, Erlang, Go, Lua, Python	C, C++, C#, Java, Python, Ruby	Ada, C, C++, C#, Java, Python, Scala, Lua, Pharo, Ruby	C, C++, C#, Go, Elixir/ Erlang, Java, Python, Swift,
Governing Body	IETF, W3C	IETF	OASIS	OASIS	OMG	IETF

** there is no limit defined but some web servers limit the maximum header size (e.g. Apache 2.4: 8KB, Internet Information Services or IIS 6.0: 16KB, Tomcat up to 48KB).*

for examining the features that differentiate between existing messaging protocols or standards is based on results from the IoT developer surveys conducted between 2015 and 2019 by the Eclipse Foundation [155]. In particular, we examine the response rates for the question "What are your top 2 concerns for developing IoT solutions?" Additionally, we consider limitations and deficiencies that were published by the International Electrotechnical Commission (IEC) in its recent outlook on the next important steps in developing smart and secure IoT platforms [176].

To this extent, we identify based on our IoT messaging protocol feature selection strategy the following characteristics including: (a) interoperability, (b) service provisioning, (c) support for microservices and distributed tracing, (d) scalability and performance, (e) reliability, (f) security, (g) adaptability and extensibility, (h) language implementations and (i) protocols' adoption rates. We believe that investigating and comparing protocols such as HTTP, CoAP, MQTT, AMQP, DDS and XMPP in the context of these features will provide guidance into choosing an appropriate protocol when building and deploying IoT systems.

A. INTEROPERABILITY

There are basic similarities among the several messaging protocols that exist today. However, given the fact that IoT systems are heterogeneous, interoperability becomes a major challenge. As Table 9 illustrates, protocols vary in the degree to which they offer features that support device communication. For example, devices that are developed by different manufacturers which are used in an IoT system may vary in the support of the communication pattern (e.g. one-to-one, many-to-one, etc.). Therefore, it becomes a challenging task to enable all of these IoT devices to communicate using a unified pattern. That is, an IoT device may not be equipped or does not support a protocol that provides a communication feature that is needed for another device to communicate properly. This is an example of a communication pattern interoperability.

There are also other interoperability issues that may arise in case of having heterogeneous devices in an IoT system. Syntactic interoperability exists when multiple IoT devices exchange data in a non-uniform payload format or structure. Data formats or structure vary depending on the communication protocol employed. For example, CoAP supports JavaScript Object Notation (JSON), a popular syntax for storing and exchanging data. XMPP on the other hand supports Extensible Markup Language (XML) for data exchange.

Furthermore, there are a number of IoT applications across different sectors that use application specific data formats or structures. The American National Standard for Utility Industry End Device Data Tables (ANSI C12.9) is a common standard that defines a data structure for transferring data between end user devices (e.g. smart grid utility device) [42]. In [43], researchers have devised a prototype that acts as a proxy for using the ANSI C12.19 implementation over DDS. The authors use an interface definition language (IDL) that utilizes two topics: (a) an electric element IDL definition and (b) a utility element IDL definition [43]. However, this approach is limited to smart grid applications. Support for a wider range of data formats or payloads spanning across heterogeneous IoT systems becomes inevitable.

To illustrate the importance of interoperability among data formats, consider an IoT system that uses devices which support CoAP over 6LoWPAN network that may need to communicate with other devices that support XMPP (i.e. these devices do not support CoAP). That is, an IoT device that supports CoAP, for example, may need to send a message whose content is serialized over a network channel in JSON format. Another IoT device in the same system that uses XMPP as a communication protocol will not properly retrieve the message since the syntactic rules used in serializing the message are different (e.g. JSON versus XML). Therefore, exchanging data in different formats or structures across heterogeneous IoT devices raises important syntactic interoperability issues that need to be addressed.

In addition, multiple IoT cloud platforms provide support for proprietary protocols other than HTTP, MQTT and AMQP. For example, Azure IoT Edge Runtime can act as a protocol translator where devices that do not have support for HTTP, MQTT or AMQP would use the gateway for sending data on their behalf [143]. The gateway in this case understands the protocol of the device generating the data and creates a message in a format that is understood by the Azure IoT Hub (e.g. HTTP, MQTT or AMQP). Figure 17 presents an illustrative example of a predictive maintenance IoT system architecture that can be used for industrial automation using Azure IoT Edge.

Through the support of custom or proprietary protocols (e.g. BLE, RJ485) on part of the Azure IoT Edge as shown in Figure 17, it is then possible to support the interoperability among heterogeneous IoT devices that need to connect to the cloud. That is, the protocol translation would allow the coexistence of IoT nodes that have different protocols. In addition, the edge gateway can act as a middleware that can perform edge analytics for running deep learning or artificial intelligence high-performance workloads or tasks. Hence,

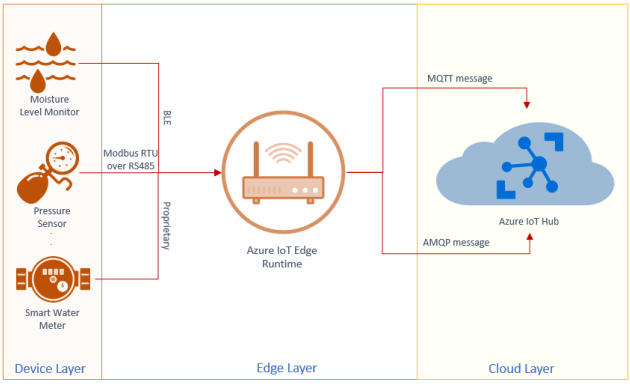

FIGURE 17. A predictive maintenance IoT system architecture using the Azure IoT Edge which acts a middleware for allowing devices to send their IoT data in proprietary formats.

intelligence can then be built at the edge of the network and insights can be discovered locally rather than on the cloud. The architecture shown in Figure 17 would be ideal, for example, for low-latency IoT applications that require support for the communication among devices that require exchanging data in proprietary formats.

Interoperability is not limited to protocol mapping but also extends to architecture, workflow, services and data access [144]. Consider an IoT system that is composed of a number of heterogeneous data sources each of which has a different data model. Figure 18 presents an example of an IoT system that consists of a number of heterogeneous data sources with different data models.

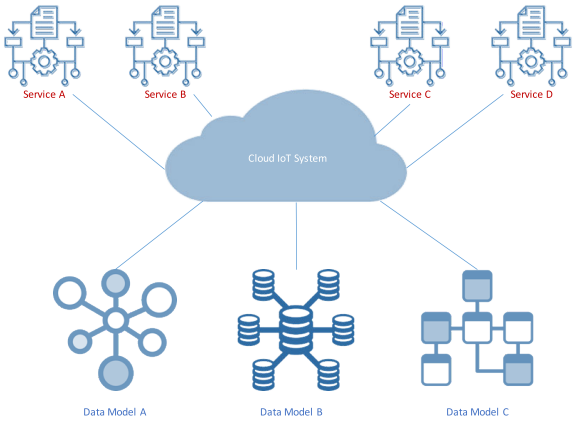

FIGURE 18. An illustrative example of an IoT system architecture composed of heterogeneous data sources (A, B, C and D).

Data generated by data sources (A, B and C) in Figure 18 have different hardware and software components which causes technical interoperability such that these devices or nodes are not able to communicate directly (i.e. no direct D2D support). This type of interoperability can also apply to devices operating during service or runtime discovery (e.g. mDNS) where metadata about data exchanges is likely be different. In addition, syntactical interoperability exists when data formats generated by the sources is different (e.g. XML, JSON, among others). Semantic interoperability may also exist since the context or meaning of data generated by the three independent data sources in the example shown in Figure 18 can be interpreted differently by various components within the IoT system.

To overcome interoperability challenges, the authors in [144] proposed a number of solutions to solve this problem. Inspired by the Linked Open Data (LOD), the authors in [144] proposed an extension to LOD for the Internet of Things or LOD4IoT. In addition, the authors also introduced Linked Open Vocabulary of Internet of Things (LOV4IoT) where the primary focus is not only on data but also includes catalogs, ontologies, datasets and rules [144].

Furthermore, in attempt to combine Semantic Web Services for unifying services, the authors in [144] introduced the Semantic Web of Things (SWoT) which primarily focuses on the interoperability of data and ontologies [145]. SWoT combines concepts from both the Semantic Web and the Web of Things (WoT), a standard introduced by the W3C [146]. In addition, The Web of Things Working Group has recently introduced a recommendation called a Web of Things (WoT) Thing Description [147]. A WoT Thing Description document is used to describe metadata and interfaces of Things. Through the Thing Description, it is then possible to allow heterogeneous applications or services to interoperate [147]. The Things Description is encoded by default in a JSON format allowing JSON for Linked Data (JSON-LD) to be processed.

In addition to the efforts in solving semantic interoperability for IoT, the DDS middleware protocol supports a lightweight encoding scheme called the Concise Binary Object Representation (CBOR) [21], [148], [149]. CBOR is a binary serialization format whose data objects are associated with name-value pairs. What distinguishes CBOR from JSON is the small message sizes [149]. Additionally, the CBOR format is extensible where tags can be used to identify data objects of models other than the ones specified in predefined data model. This enhances the interoperability of IoT systems, for example, that involve different types of data models across a number of data sources as shown in Figure 18.

B. SERVICE PROVISIONING

An IoT application is likely to integrate or consume one or more services to achieve specific functionalities (e.g. via SOAP or REST services). Interoperability for IoT service provisioning therefore becomes essential. In recent years, HTTP has become the de facto standard or protocol for consuming RESTful services. Unlike RESTful services, SOAP services communicate via the Simple Object Access Protocol (SOAP). The Device Profile for Web Services (DPWS), an OASIS standard, enables web service messaging, discovery and description on resource-constrained endpoints while supporting SOAP over UDP [29]. However, HTTP and SOAP may not be ideal for service-binding across all types of IoT systems.

CoAP provides support for low-powered IoT devices to interoperate via HTTP in order to communicate in a RESTful manner with services. Because SOAP, an XML-based protocol that works at the application layer, is often associated with very large message sizes, this makes the SOAP protocol inadequate for resource-constrained IoT devices that require service bindings while maintaining small message sizes. Hence, CoAP becomes an excellent choice for IoT devices that need to coexist or interoperate with HTTP for service provisioning. That is, CoAP becomes an ideal protocol for IoT systems that require constrained devices to perform real-time service bindings in a RESTful manner.

XMPP, a protocol that is often used for near real-time data packet exchanges, supports the XML format which makes it another suitable protocol choice for provisioning services in IoT systems. For example, assume an IoT system requires

IoT devices to communicate via XMPP in order to resolve SOAP- and REST-based service endpoints via a firewall proxy. Running RESTful services over XMPP may not be an ideal choice, although they can be accomplished with proper protocol design considerations. Providing REST support in protocols can enable IoT systems to support the provisioning of services particularly on mobile devices, industrial automation and robot control, among many others.

The SOAP-over-XMPP is an XMPP binding (XEP-0072) which provides binding support for SOAP messages to XMPP [30], [31]. XMPP provides support for transporting XMPP datagrams using HTTP which enables XMPP's support in languages such as JavaScript running in browsers [32]. In contrast, DDS does not support data distribution to operate via HTTP (e.g. access DDS via a web browser to provision services). However, a RESTful DDS was introduced for filling this gap where a RESTful service connects to a DDS backbone listening for any incoming HTTP requests [33]. As for the rest of the IoT messaging protocols, there are a number of tools or libraries that can be used for supporting RESTful interfacing such as RoboMQ over REST for AMQP [34], RabbitMQ message broker [35] and Eclipse Paho for supporting REST interfaces via MQTT [36], among many others.

Because web services are modular entities that communicate via the web, HTTP becomes the natural protocol for service provisioning. In fact, early web services used to communicate using XML standards (e.g. SOAP). Given the overhead associated with XML and popularity of miniaturized devices (e.g. mobile devices) and ubiquitous computing, an apparent shift to utilize protocols that are commonly used for web browsing is inevitable. Hence, RESTful services have become in recent years more popular due to the wide acceptance of HTTP. Whether RESTful HTTP remains the protocol of choice for service communication or not, the need for more efficient service protocols that can accommodate IoT application deployment is desirable.

C. MICROSERVICES AND DISTRIBUTED TRACING

In recent years, microservices have become fundamental building blocks for developing many IoT applications. Whether these microservices are deployed on edge-based networks or the cloud, they are loosely coupled and provide high level of modularity. As the level of intelligence increases on an IoT device (i.e. fog- or edge-based), the more likely that there exists a number of cooperating entities in the form of modular microservices that are capable of delivering multiple different functionalities. An IoT application in such scenarios may need to utilize distributed tracing for observing and monitoring the behavior of these microservices. To this extent, messaging protocols need to provide support for this new form of end-to-end observability of IoT applications.

Support for distributed tracing in non-HTTP protocols or beyond the use of HTTP headers has been inadequate or inexistent. For instance, HTTP currently provides extensive support for distributed tracing through a number of tools such as Zipkin, OpenTrace, Dapper and Jaeger, among many others. Through HTTP headers, it is then possible to trace data and service requests across a distributed IoT application. However, support of distributed tracing across non-HTTP protocols is very limited.

In recent years, a number of efforts or initiatives have been proposed for supporting distributed tracing in non-HTTP protocols such as MQTT and AMQP. These initiatives depend primarily on the use of the TraceContext acting as a proxy [39], [40]. However, the application of distributed tracing to publish-subscribe interaction pattern has been very rare or inexistent. Real-Time Innovations (RTI) provides a framework that enables the support for near real-time communication between microservices while maintaining some levels of QoS through Connext DDS Micro [41], [79]. However, this framework does not provide real-time observability or monitoring of IoT devices. As the dependency of IoT systems on using modular microservices to execute traditional tasks or workloads (e.g. data analysis, machine learning, artificial intelligence), the need for distributed tracing support in non-HTTP messaging protocols (e.g. MQTT or AMQP) becomes increasingly essential.

D. SCALABILITY & PERFORMANCE

Scalability is an integral software design property in which a system is capable of maintaining to deliver its functionality within an acceptable degree of quality as the number of users or workloads that needs to be accomplished increases. When dealing with IoT devices, for example, scaling system resources to accommodate an increasing magnitude of real-time data becomes an important design element of IoT systems.

To illustrate the importance of scalability, consider for example an IoT system that has a number of IoT devices that can capture images and videos attached around a multipurpose arena. These IoT devices have limited resources to perform image processing and artificial intelligence techniques. The frequency at which these IoT devices are programmed to capture images or videos depends primarily on the number of objects detected in a captured image. As more objects are detected (e.g. more faces or humans) in captured images, the frequency to capture images or videos increases. Assume the frequency is originally set to 30 seconds. When there is an increase in number of objects, the frequency of capturing images or videos is set to increase (i.e. time duration is shorter or 5 seconds). This enables the system to capture more images and therefore can provide real-time security and monitoring of events surrounding the arena. The IoT system in this case needs to scale up to the increasing demand of resources while maintaining an acceptable degree to which the system can respond (i.e. response time). Choosing a protocol that supports scalability in this case is essential to the functioning of this real-time detection system.

Furthermore, when choosing a message protocol such as MQTT, the number of message transmissions increases significantly as more clients subscribe to topics. In the

multipurpose arena IoT system scenario, assume that the number of IoT devices is 100 each of which is subscribed to all of the existing topics. When an IoT device is triggered, it sends a message to the broker (one transmission). The broker would then notify all subscribers (in this case 99 transmissions). This is a total of 100 transmissions. If another device sends a message to the broker, the number of total message transmissions increases to 200. In the worst case scenario where all IoT devices are transmitting messages, the total number of messages to be transmitted is equivalent to the product of the number of messages per device (i.e. 100) and total number of subscribed clients (i.e. 100). Hence, this would yield 100×100 or 100^2.

Assuming that there is a set of clients defined as C that communicate with the message broker such that $C = \{c_1, c_2, c_3, \ldots, c_n\}$ where n represents the total number of clients and the average number of topics each client is subscribed to is `topic_average`. In order to calculate the total number of message transmissions required (or throughput), we define `trans_msg` as follows:

$$trans\_msg = n \times topic\_average$$

In the worst case scenario, the `topic_average` equals to the total number of clients. In this case, the total message transmissions is equivalent to n^2. Assume the average time it takes to transmit a message is $\texttt{time}_{\text{msg}}$. Then, the total time for all possible message transmissions can be defined as follows:

$$trans_{time} = trans_{msg} \times time_{msg}$$

In the previous example, assuming that it takes on average 10ms to transmit a message and the average number of subscribed topics is equivalent to all clients subscribed to all topics. Also assume that there are 100 clients. Therefore, the total time for all possible message transmissions would be:

$$trans_{time} = 100 \times 100 \times 10ms = 1.67minutes.$$

This example illustrates that as the number of messages increase, the total transmission time also increases. The case can become worse if the broker's resources are maximized and hence a broker becomes a source for a Single Point of Failure (SPoF). Because MQTT is designed primarily for a Device-to-Cloud (D2C) communication scope, the problem of SPoF can be resolved by serverless computing which increases the agility of the deployed IoT application. Figure 19 provides an example of a serverless solution for an air quality fog-based IoT system that uses the AWS IoT Core platform.

As can be seen in Figure 19, the device layer is composed of a number of IoT nodes represented by the ESP32 modules equipped with air quality sensors for measuring outdoor air quality. The fog-based system utilizes the multicast DNS (mDNS) which resolves host names to IP addresses within a small sized network [159]. The mDNS protocol operates on

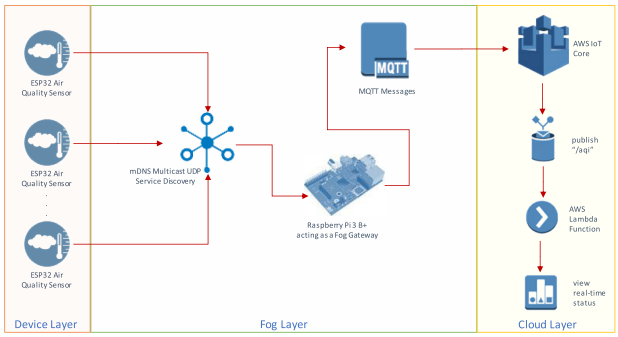

FIGURE 19. A fog-based serverless air quality monitoring IoT system using AWS.

a local and not a global network which makes it ideal for the device layer of the IoT reference architecture (see Figure 1).

In addition, the DNS Service Discovery (DNS-SD) protocol [160] which complements that of the mDNS offers a zero-configuration networking (zeroconf) feature which makes the discovery of devices over the network automatic [161]. The CoAP, DDS and XMPP protocols support dynamic discovery of devices on a local network. Through mDNS, each node can act as a server and it supports multicast and unicast operations. Avahi is among the popular zeroconf network implementations that exist for Linux and BSDs [161]. Bonjour is also another zeroconf technology developed by Apple [162]. Bonjour is capable of identifying devices and services on a local network using mDNS service records.

Another service discovery approach is the W3C's Web of Things (WoT) used to publish URIs representing things, devices or nodes [146]. There are a number of technologies that adopt the WoT approach including HyperCat [163], Physical Web [164] and Universal Plug and Play (UPnP) [165]. HyperCat is an open-source, lightweight JSON-based hypermedia catalog format whereas the Physical Web is a technology that is used to broadcast their URIs or localized data within web pages. UPnP provides a group of networking protocols which enable IoT nodes to discover one another within a local network. Unlike HyperCat and Physical Web, UPnP supports data sharing.

As illustrated in Figure 19, when IoT nodes are running, a local Raspberry Pi that acts as a fog-based gateway will forward the data on part of these air quality sensors. The gateway will send the data using MQTT to AWS IoT core which is used for telemetry. The geographical distance between IoT nodes and the cloud server deployment plays an increasingly important role when considering network latency. Because MQTT is D2C communication scope, the further these nodes are from the deployed serverless solution in terms of distance, the longer is the travel time of the MQTT messages and the higher the latency. The cloud layer in Figure 19 provides a serverless solution to maintain high levels of scalability for published MQTT messages

However, existing implementations of MQTT vary in terms of performance and scalability. In [111], the authors

compared Mosquitto, HiveMQ and BevyWise MQTT broker deployed on the cloud. Using QOS level 1, the data rate is designed to be approximately one message per second for all of the three brokers (minor variations) using a single client subscribed to one topic. Using our analogy, if 10 concurrent clients are subscribed to all topics, the rate would be 1/10 or 0.1 msg/sec. Hence, as we increase the number of clients and subscribed topics, the rate at which messages are delivered significantly decreases which is a major scalability problem as the system scales up. As such, using MQTT as a messaging protocol for data transmission in IoT applications such as mission-critical IoT systems or IoT autonomous vehicles may not be ideal.

There are a number of studies that have examined the performance and scalability of communication protocols. Some studies have found that the throughput in MQTT drops significantly as the number of clients' subscriptions increases [112], [113]. In [114], the authors compared the performance of DDS and MQTT. They have found that DDS delivers high performance whereas MQTT shows significant deterioration in sending round-trip time packages [114].

In [115], the authors have found that MQTT outperforms CoAP in terms of delay, the number of messages lost and the number of bytes used in messages. In [116], the authors evaluated the performance of CoAP and MQTT-SN and determined that MQTT-SN outperforms CoAP. In [118], the author compared the performance of AMQP and MQTT and have found that AMQP slightly outperforms the MQTT and CoAP protocols. Furthermore, the research study concludes that CoAP uses more computing resources compared to MQTT whereas AMQP's use of resources is much higher when compared to both MQTT and CoAP protocols' resource consumption.

The performance and scalability of messaging protocols vary. CoAP and HTTP have an additional overhead associated with transporting messages. CoAP and HTTP should be used in cases where low latency is not of critical importance. Hence, this makes these two protocols not very suitable for real-time IoT edge- or fog-based systems. MQTT follows a device-to-cloud data flow and therefore introduces network latency dependent on the cloud service performance.

Generally, message brokers may deliver 100 to 1000 messages per second per subscriber [20], [117]. However, these messaging brokers can vary significantly. For example, XMPP is generally used for client/server or instant messaging applications and is known to be unsuitable for low latency networks. On the other hand, DDS is generally deployed on a local area network (LAN) where publishers and subscribers can utilize the UDP's multicast feature. Hence, DDS can scale to 1000 messages/second for every peer device on a network consisting of thousands of devices [20]. DDS is very suitable for low-latency IoT systems consisting of thousands of IoT nodes. Furthermore, due to the fact that DDS is deployed on a LAN, this makes the protocol suitable for edge- and fog-based IoT distributed systems' deployments.

There are a number of research efforts that attempted to compare the performance of existing messaging protocols. In the subsequent sections, we provide a summary of the existing studies that published performance metrics when testing these protocols.

1) PERFORMANCE COMPARISON ON LATENCY

The authors in [151] conducted a performance analysis in terms of the latency for MQTT and HTTP. The study involved two testbeds: (a) a fog-based MQTT broker and (b) a cloud MQTT broker based on an existing IoT cloud service provider platform. Results from this study show that HTTP was associated with having higher response times of 12.1 and 4.76 when compared to that of MQTT fog and cloud based deployments, respectively. Additionally, the authors identified that MQTT with QoS Level 1 is associated with a packet loss of 6.2 times higher when compared to that of MQTT QoS Level 0 in a local, fog-based testing environment.

In another study conducted by authors in [152], a home automation testbed was used to compare MQTT and HTTP where results show that MQTT had a much lower latency compared to HTTP. In another study that involved CoAP, the authors in [4] compared the performance of MQTT and CoAP. Results from this study show that MQTT messages were associated with lower delays than that of CoAP messages at lower packet loss rates whereas MQTT has a higher delay than that of CoAP at higher packet loss rates. The authors in [153] also conducted experiments comparing MQTT and CoAP concluding that CoAP performs better in terms of bandwidth, usage and response time. This suggests that CoAP is more suitable for applications that require low latency and reduced resource consumption which makes it ideal for fog- and edge-based environments.

In comparing the performance of other protocols, the authors in [154] focused on the publish-subscribe messaging protocols including MQTT, AMQP, XMPP and DDS. Using a JavaScript client implementation, the authors recommend MQTT as a reliable protocol that fulfills IoT web application requirements. Results from this study show that MQTT had the lowest latency followed by AMQP. Both XMPP and DDS had higher latency and their results were very comparable. However, the difference among the latency performance is small due to the limited number of messages used in the comparison study.

In a more in-depth study, the author in [118] tested MQTT, CoAP, AMQP and DDS for small payload with 10 messages and 1000 messages. Results show that in the small payload, CoAP outperforms all of the other protocols followed by MQTT, AMQP and finally DDS. However, when the payload increases to 1000 messages, MQTT performs the worst. In this test, DDS outperforms all of the other protocols followed by CoAP, AMQP and finally MQTT. This suggests that DDS performs better as the number of messages increases. Because DDS and CoAP are decentralized approaches, this makes these protocols more reliable as the number of messages increases.

In addition, DDS and CoAP use UDP as the transport layer protocol which is associated with much lower overhead compared to that of TCP that is used in MQTT and AMQP. The study also examined increasing the payload size for 10 and 1000 message tests. Results from the 10 messages with message size greater than 50Kbytes reveal the ordering from low to high latency as CoAP, DDS, AMQP and MQTT. Running the same test for 1000 messages with message size greater than 50Kbytes show that DDS outperforms other protocols followed by CoAP, AMQP and MQTT. Figure 20 presents a summary of the results presented in the study in [118].

Small Message Size (< 5kBytes)

10 messages (low to high latency)	1000 messages (low to high latency)
CoAP → MQTT → AMQP → DDS	DDS → CoAP → AMQP → MQTT

Large Message Size (> 50Kbytes)

10 messages (low to high latency)	1000 messages (low to high latency)
at 20Kbytes: CoAP → AMQP → MQTT → DDS	at 20Kbytes: DDS → CoAP → AMQP → MQTT
at 50Kbytes: CoAP → DDS → AMQP → MQTT	at 50Kbytes: DDS → CoAP → AMQP → MQTT

FIGURE 20. Summary of the results from the performance tests conducted in the study in [118].

The summary in Figure 20 show that MQTT performs the worst when compared to all of the examined protocols particularly having larger message sizes and increasing the number of messages. Results from this study [118] coincide with our mathematical results in Section V.D which suggested that the performance of MQTT in terms of latency degrades significantly as the number of messages increases. It is worth noting that DDS performs well at higher message sizes and having large number of messages. For example, for a small message size with 10 messages, DDS performs worst. However, when increasing the number of messages and significantly increasing the size of the payload for smaller messages, DDS outperforms all of the other protocols. This suggests that DDS's decentralized architecture as well as the use of the CBOR encoding scheme significantly enhance the protocol's performance compared to CoAP, AMQP and MQTT. Furthermore, results also show that CoAP is a stable protocol across all tests conducted.

2) PERFORMANCE COMPARISON ON THROUGHPUT

In terms of throughput, the authors in [152] report that MQTT had the highest number of messages per hour when comparing it with CoAP and HTTP. In addition, MQTT used the least power consumption in delivering large volumes of messages compared to CoAP and HTTP. In [157], the authors presented a performance comparison study of (RESTful) HTTP and AMQP. The performance testbed used RabbitMQ with executing three sets of experiments. The study concludes that AMQP offers much higher bandwidth compared to RESTful HTTP. In a similar study, the authors in [158] compared HTTP and CoAP while assessing the data transmission based on dynamic network conditions. The study concludes that CoAP outperforms HTTP with respect to the delivery rate, delay and overhead [158]. In [5], the authors reached the same conclusion when comparing MQTT and CoAP with CoAP being best performer for small payloads. However, in the study, the authors report that CoAP performance is bad as the payload size increases.

3) PERFORMANCE COMPARISON ON CPU, MEMORY AND POWER CONSUMPTION

As for CPU and memory consumption, the author in [118] reports in the study which examined MQTT, CoAP, AMQP and DDS that DDS consumed the highest memory consumption, followed by AMQP, CoAP and finally MQTT. For CPU consumption, AMQP had the highest, followed by DDS, then MQTT and finally CoAP. The same results were obtained when testing for small payload (∼5Kbytes) and for high payload (>50Kbytes) both test running with 10 and 1000 messages, respectively. In [152], the authors report consistent ordering such that MQTT had the least power consumption in terms of battery life followed by CoAP and then HTTP. That is, HTTP had 5.3 times the power consumption of that of MQTT and 2.8 times the power consumption of that of CoAP [152]. In addition, results from the study also show that CoAP had 1.9 times the power consumption comparing it to MQTT.

Although there exists a number of research studies that examined the performance of IoT messaging protocols, these studies varied in the degree to which the testing environments were deployed. That is, some of the results presented in these studies reflect different QoS levels for various protocols. Furthermore, some studies have considered excluding the initialization time to run the tests while others did not provide clear details about whether the latency time includes initialization time. Additionally, testing environments may be affected by the implementation language of the protocol used.

E. RELIABILITY

Relying on User Datagram Protocol (UDP) at the transport level provides constrained IoT devices an optimal method for data transmission by removing the TCP overhead and thus reducing bandwidth [24]. However, UDP does not provide support for flow or error control nor retransmission. When choosing an IoT communication protocol, it is essential to identify the level of message reliability required for the system.

Majority of the IoT messaging protocols listed in Table 6 operate over TCP. CoAP operates over UDP and provides retransmission to overcome this limitation by having two bits in the header indicating the type of message (e.g. confirmable, non-confirmable, etc.). The problem, however, is that CoAP does not have a built-in mechanism to verify whether a message has been received in its entirety or decoded correctly. In addition, CoAP does not provide built-in security and uses Datagram Transport Layer Security (DTLS). Furthermore, DTLS does not support multicast and requires supplementary packets during handshakes [25]. As a result, more processing power is utilized to compensate for increasing network

traffic during handshakes. Additionally, CoAP devices may not work properly due to resource discovery problems behind Network Address Translation (NAT) since devices' IP addresses can be dynamically assigned over time [26].

To illustrate the limitations of dynamic resource discovery in CoAP, consider for example an edge-based IoT system that has a number of CoAP IoT devices each assigned an IP address. When a CoAP device moves across different networks, a new dynamic IP address is likely to be assigned to the CoAP device. In this case, the CoAP device may not be reachable by the edge-based IoT system. That is, the system would not be able to communicate with the CoAP device due to the fact that it is not aware of the new dynamically configured IP address when the IoT device is moving across various networks. Unfortunately, CoAP does not provide mobility management when CoAP IoT nodes move across networks [27]. When building IoT systems that require mobility, it is essential to consider messaging protocols that support dynamic discovery. Not all messaging protocols support dynamic discovery as presented in earlier Table 9.

XMPP's baseline provides a very reliable stream transports. XMPP operates over a continuous TCP stream which means that every stanza arrives safely preventing the Two Generals Problem from occurring [37]. In addition, XMPP provides XEP-0198, an extension that enhances TCP's reliability over a number of open sessions in order to provide reliable data transmissions via mobile networks [38]. DDS provides a reliability policy for indicating the level of reliability by a DataReader or DataWriter [21]. In addition, DDS provides redundancy (or retries) to provide guarantees of the continued delivery of operations in order to increase the resiliency and reliability of applications. However, with the increase of reliability comes the increase in the overhead associated with processing messages on part of the IoT devices.

F. SECURITY

As the number of microservices consumed by an IoT system increases, so does the level of communication between them which augments the need for security. As outlined in Table 9, nearly all of the protocols examined are based on the transport layer security (TLS) cryptographic protocol. Hence, these Messaging Protocols are vulnerable to attacks that may occur or can be performed to the TLS protocol. Furthermore, IoT devices are generally used by humans which makes them vulnerable to intruders that attempt to gain unsolicited access or collect confidential personal data in a malicious manner. Since IoT devices are resource-constrained, they may not be equipped with the necessary processing power or computing resources required to execute or run complex security operations. As a result, these IoT devices become an easy target for attackers or intruders.

There are five possible sources that make IoT systems vulnerable to attackers including: (a) devices or things, (b) connectivity medium, (c) computing, (d) storage and (e) microservices. Furthermore, a security function for IoT systems can be built at the local level (i.e. edge-based) at the physical hardware or software levels. We identify three types of threats that may occur in IoT systems while examining security measures that exist across messaging protocols and mapping possible sources of vulnerabilities to common security practices including the CIA Triad (confidentiality, integrity and availability) and the IEEE AAA (authentication, authorization and accounting).

When software running on IoT devices is compromised, device integrity becomes an issue since data can be modified by individuals who are not authorized to make changes on the device. In terms of messaging protocols, device integrity can occur when an intruder or attacker subscribes to an existing publisher to collect data and use it maliciously. An IoT communication protocol needs to ensure that only authorized users regardless if they are publishers or subscribers. This feature is not provided at the TLS level since its main responsibility is for securing the communication via a computer network.

Furthermore, such vulnerabilities may occur when offering QoS level 2 which may explain why many IoT cloud providers not to provide support at this level as presented in Table 7. Currently, AMQP and DDS provide mechanisms for authorizing entities. AMQP uses SASL to perform this task while DDS offers authorization for both DataReaders and DataWriters through the DDS security model. In addition, the DDS security model offers extensions for implementing authentication (e.g. certificate management), cryptography and access control. ARM provides an open-source library called DDS Security library (libddssec) that supports security services for DDS implementations [108]. However, this library is limited to operations that use ARM's TrustZone technology [108], [109].

In addition, DDS provides Extending the DDS through the Service Plugin Interfaces (SPIs) which enables applications to provide support for security functions such as authentication, access control, encryption, message authentication and digital signing, among many other features. For example, an authentication service plugin in DDS provides support for verifying application or user identity when invoking DDS operations (e.g. DataReader, DataWriter). Table 10 provides a list of the five SPIs defined in DDS Security Model.

Additionally, XMPP provides an extension called Authorization Tokens for issuing authentication tokens to client applications [110]. During the process of stream negotiation between XMPP client and server, a client typically needs to provide a password when a connection is established. This means that a client needs to store this password and reuse it every time a connection is made to the XMPP server. The problem with this security function, however, is that this mechanism increases the risks associated with the security threats of storing an account password on the IoT device. To this extent, XMPP offers support for clients to obtain tokens in XML structure when a request is made to a XMPP server. When a client establishes a connection with a XMPP server, the server then responds with a list of supported proto-

TABLE 10. DDS security model service plugin interfaces [49].

SPI	Description
Authentication	identify verifications of application/user invoking DDS operations *example: mutual authentication and establish shared secret*
Access Control	enforce policy decisions for DDS operations *example: topics that a user/application is able publish/ subscribe to*
Cryptographic	implement (or can interface with entities that can perform) cryptographic operations *examples: encryption, decryption, hashing, among others*
Logging	stores DDS security-related events or logs for auditing
Data Tagging	a method for associating tags to data object samples

cols for authentication. The client then send the next element for authentication including: (a) user name and (b) the issued authorization token (usually in base64 encoding). Instead of applying authentication with a password using a plain method, a token is used to replace the password [110].

Although each protocol provides different levels of security measures, DDS provides the most comprehensive security levels for authentication, access control and cryptography among others. In addition, DDS provides a framework for extending the security by supporting a wide range of SPIs. XMPP also offers a robust authentication process using tokens (based on SASL) followed by AMQP.

G. ADAPTABILITY AND EXTENSIBILITY

As IoT systems increase in size and complexity, the need for preserving and extending existing software components is essential. In addition, software requirements of IoT systems may change as new use cases or features transpire and existing software components need replacements. To this extent, increasing the agility of IoT systems and adapting to changes in requirements becomes inevitable. That is, software systems need to be extensible such that organizations are able to make additions or modifications [98]. In order for IoT systems to offer good levels of extensibility, it is imperative that components which are integrated into these systems support extensibility. In particular, integrating a messaging protocol that supports or provides custom protocol extensibility will undoubtedly increase the overall level of extensibility of IoT systems while enabling them to interact across heterogeneous environments.

Among all of the examined protocols, XMPP provides the most protocol native support for extensibility. The XMPP messaging framework leverages XML as the basic communication format which translates into high levels of extensibility. For example, the XMPP core protocol provides support to over 300 extensions such as Malicious Stanzas (XEP-0076) [101], Service Discovery (XEP-0030) [102] and User Location (XEP-0080) [103].

The XMPP malicious stanzas extension provides support for determining if a packet was transferred over the network with a malicious intent [101]. It attempts to identify the evil bit in IPv4 and hence identify evil messages in message stanzas. The service discovery extension provides support for discovery information about XMPP entities including: (a) identity and capabilities of an entity and (b) items associated with an entity [102]. The user location extension, which is still in draft state at the time of writing this paper, aims to capture data with respect to the entity's geographic location (geoloc) [103]. The geoloc information is determined based on a Global Positioning System (GPS) coordinates. Although this feature may be useful for IoT devices operating using cellular or wireless networks, IoT systems with constrained devices may not be able to have capabilities of identifying GPS coordinates. However, fog-based gateways in an autonomous vehicle systems can take advantage of this feature extensively.

DDS also provides some level of extensibility through a specification called Extensible and Dynamic Topic Types for DDS (DDS-XTypes) [104]. Using extensible types, one can extend the addition of new elements that can be associated during publish/subscribe process without impacting or making changes to applications or end devices. To illustrate this extensibility, consider a fog-based autonomous vehicle system where a fog-gateway uses the publish-subscribe model using a DDS bus for seamless connectivity between vehicles, sensors and cloud-based applications. Figure 21 presents a high-level overview of a device-to-gateway-to-cloud autonomous vehicle system.

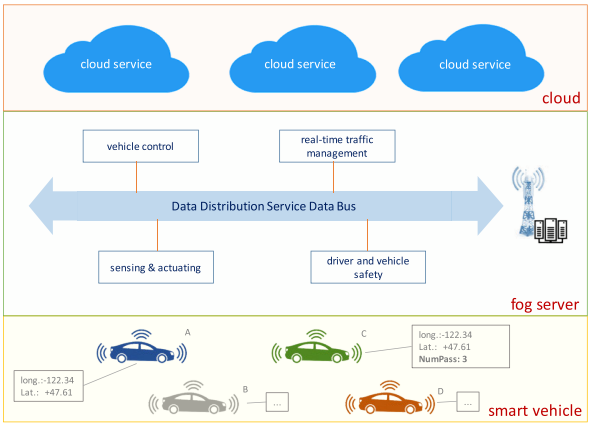

FIGURE 21. A fog-based autonomous vehicle system using DDS.

As shown in Figure 21, vehicle A is programmed to transmit via a DDS data bus localized data such as longitude and latitude information whereas vehicle C requires the transmission of geographic coordinates in addition to the number of passengers within the vehicle. In this scenario, autonomous vehicles that subscribe through the DDS-enabled fog-based gateway (e.g. cellular base station) to share only geographic data (e.g. longitude and latitude) will not interoperate with those vehicles that share geographic data in addition to number of passengers. However, through DDS-XTypes, it is then possible to extend the capability of this autonomous vehicle system by extending and evolving the data types without

TABLE 11. CoAP client-server implemetations.

Language	CoAP Implementations
.NET	CoAPSharp [50], CoAP.NET [51], Waher.Networking [52]
Java	Eclipse Californium [53], nCoAP [54]
JavaScript	node-coap [55]
Python	aiocoap [56], CoAPthon [57], txThings [58]
Go	Go-CoAP [59]

TABLE 12. AMQP client implemetations.

Language	AMQP Implementations
.NET	AMQP.Net Lite [60], .NET RabbitMQ [61]
Java	Spring AMQP [62], RabbitMQ Java Client Library [63]
JavaScript	amqplib [64], amqp.node [65], rabbit.js [66]
Python	Pika [67], Rabbitpy [68], Celery [69], aioamqp [70]
Go	Go RabbitMQ Client Library [71]

making changes to the programs or applications. That is, DDS provides support to extend data objects to have multiple data types. Therefore, both data types associated with vehicle A and vehicle C can coexist within the same IoT system. Volkswagen [106] and Audi [107] are among vehicle manufacturers that have recently adopted DDS for connectivity and testing of their smart cars.

AMQP implementations such as RabbitMQ offer a number of extensions to the native specification in an attempt to enhance the extensibility of AMQP. For example, a Per-Queue Message TTL provides support to determine the duration of an unconsumed message waiting in a queue prior to its deletion [105]. This enables an IoT system, for example, to identify the life of a message. This can also be used in supporting transactional processing in IoT systems and potentially identifying possible deadlocks (e.g. when a message never gets consumed). Other protocols such as HTTP, CoAP and MQTT have very limited support for extensibility which impacts the evolvability and interoperability levels of IoT systems using them.

H. PROTOCOL IMPLEMENTATIONS

Messaging protocols examined throughout this paper vary in terms of the functionalities they support. However, they all offer common features that make them suitable for various communication types such as device-to-device (D2D), device-to-cloud (D2C) and cloud-to-cloud (C2C) interactions. While many of the existing protocols have been used in a number of platforms across many organizations, some protocols are released with adequate features published in their specifications or standards' documentation while others are still evolving. Some of the proposed features in CoAP's standard (RFC7252) [49], for example, are still under development. Nonetheless, additional language support or implementations for CoAP has been growing. Tables 11 presents a partial list of implementations that support CoAP.

In Table 12, we present a partial list of the AMQP client implementations that are available across various languages. In addition, there are a wide range of AMQP broker implementations that exist across various platforms including, but not limited to, the following:

- Apache: Qpid [72], ActiveMQ [73],
- Azure: Event Hubs and Service Bus [74],
- Solace: PubSub+ [75],
- VMWare: RabbitMQ [76],
- Red Hat: Red Hat AMQ [78], Enterprise MRG [99],
- StormMQ [77],
- OpenMQ [100],
- IronMQ [94] and
- Amazon SQS [95] *(no support of publish/subscribe).*

DDS language support is very limited compared to other protocols. RTI Connext provides extensive support for the DDS implementation in languages including C, C++, .NET and Java [79]. RTI also provides a RTI Connector for enabling scripting languages such as Python and JavaScript (Node.js) to access a DDS network via a WebSocket [80]. In addition, OpenDDS is an open source C++ implementation of DDS [81]. OpenDDS also provides support for Java through JNI bindings and JavaScript through IDL mapping.

TABLE 13. XMPP implemetations.

Language	XMPP Implementations
.NET	artalk-xmpp [82], Matrix vNext [83], Waher Networking [84]
Java	Apache Vysper [85], Stroke [86]
JavaScript	xmpp.js [87]
Python	aioxmpp [88]

As for XMPP, there are a number of client/server XMPP implementations available. Table 13 presents a list of libraries that provide support for XMPP offered in different languages. In addition, there is a number of XMPP server implementations that support a wide range of platforms including, but not limited to, the following:

- Apache Vysper [85],
- Openfire [89],
- Tigase XMPP Server [90],
- MongooseIM [91] and
- IoT Broker [92].

It should be noted that the size of a protocol's implementation may vary significantly from one protocol to the other. This also depends on the language to be used for implementation due to library or code interdependencies. For example, CoAP.NET is a.NET-based client-server CoAP implementation that requires over 10MB of disk space whereas a Windows-based DDS implementation from RTI requires over

640MB of disk space. CoAP-Lite, a lightweight implementation of CoAP, is a Linux-based client-server implementation which consumes approximately 14KB of disk space [97]. The size of this library would be ideal, for example, for low-power or constrained IoT devices particularly those that are running close to the edge of a network (i.e. edge- or fog-based IoT devices).

Considering the type of IoT devices and choosing a suitable protocol therefore becomes an important decision when designing and deploying IoT systems. For example, constrained edge-based IoT devices typically have limited storage. As a result, these devices may not be able to consume a client version of DDS whereas CoAP may be more appropriate to employ on such devices. Nonetheless, CoAP may not perform well when compared to other messaging protocols (e.g. MQTT, AMQP or XMPP) which makes it inacceptable to use for mission critical IoT systems. Therefore, determining an appropriate protocol to be employed in IoT systems is influenced by a number of factors that not only include physical device or operating system features such as storage capacity or preferred language of implementation but also factors that relate to performance, communication overhead, network latency, security, ease of use, reliability, popularity, dependability, supportability, interoperability and scalability, among many others.

I. ADOPTION RATE OF MESSAGING PROTOCOLS

In previous sections, we presented feature comparison of the messaging protocols HTTP, CoAP, MQTT, AMQP, XMPP and DDS. Irrespective of the features that differentiate each protocol from one another, it would be worthwhile to determine any usage patterns or adoption rates for all of the protocols we examined in this research study. To this extent, we summarize the results from an annual survey conducted by the Eclipse Foundation and Eclipse IoT Working Group between 2015 and 2018 [155]. The survey contained a number of questions to developers and solution architects. We focus on the response from a question that we believe is relevant to our study which is "*What messaging protocol(s) do you use for your IoT solution?*" The response rate in answering this question for the surveys conducted between 2015 and 2018 is shown in Figure 22.

Although HTTP remains to be among the top two protocols across all years, there is an apparent drop in its usage with approximately 9% decrease in 2018 as illustrated in Figure 22. On the contrary, MQTT has witnessed a significant increase in its usage from 2017 to 2018. The survey results from the 2019 Eclipse Foundation and Eclipse IoT Working Group is partial but report that HTTP and MQTT remain among the top three messaging protocols [156]. In addition, the AMQP protocol has been witnessing a gradual increase over the years in terms of its usage by developers who have completed the surveys and as illustrated in Figure 22. DDS remains steady at the same rate while CoAP has witnessed a slight decline between 2017 and 2018. In the

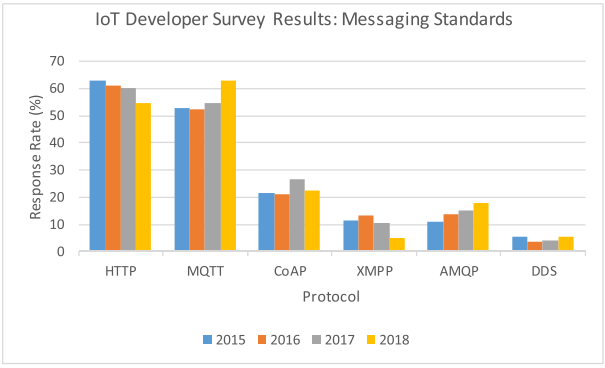

FIGURE 22. The eclipse foundation IoT developer survey results showing the response rates for the messaging protocol question between the years 2015 and 2018.

results from the 2019 survey, the report show that CoAP has dropped below a 15% response rate.

The results provided by the Eclipse Foundation and Eclipse IoT Working Group provide a measure of the extent to which the messaging protocols examined in this research study are used in the development and deployment of IoT applications. However, there are a number of factors that need to be considered when analyzing these results. For example, the response rate is provided as a percentage while the number of participants vary from one year to another. The number of participants in the 2015 survey was 394 while in 2016 this rate increased to 528. In 2017, there was another increase in the number of participants reaching 713. However, the 2018 survey decreased by more than 30% having 502 participants. This is contrary, for example, to the number of participants in 2019 where they were 1717 participants [156]. However, the released survey results from the 2019 year does not provide comparable result presentation strategy as to those from previous years [156].

Furthermore, the results from the IoT Developer Survey [155] aligns with those that we investigated with respect to the current support of IoT messaging protocols across existing IoT cloud providers. Based on our findings as shown in Figures 3 and 4, HTTP and MQTT are the two top messaging protocols that are supported by all of the ten cloud platforms we investigated. This is followed by AMQP (ranked third) whereas CoAP is ranked fifth. This may be attributed to a number of factors or could be interpreted potentially as an evident decline in the adoption rate of the CoAP protocol in recent years whereas there exists an apparent growth to the adoption rate of the AMQP protocol. Moreover, XMPP is ranked fifth as can be seen in Figure 22 which also aligns with our results shown in Figure 3 whereas XMPP is also ranked fifth.

VII. ADVANTAGES AND DISADVANTAGES OF IoT MESSAGING PROTOCOLS

As part of investigating the characteristics of IoT messaging protocols, it would be desirable to identify the key strengths and weaknesses of each protocol in the context of

IoT. To this extent, in this section, we identify key advantages and disadvantages for each of the examined messaging protocol. We believe that this summary would be useful to highlight important and unique features offered by each protocol. We also summarize the strengths and weaknesses of each protocol in Table 14.

TABLE 14. Summary of strengths and weaknesses of examined protocols based on our selection strategy.

Criteria	HTTP	CoAP	MQTT	AMQP	XMPP	DDS
RESTful	high	high	none	none	none	none
Interoperability	low	low	low	moderate	high	high
Service Provisioning	low	low	low	moderate	moderate	low
Distributed Tracing	high	none	none	none	none	moderate
Scalable	low	low	moderate	moderate	moderate	high
Reliability	low	low	moderate	moderate	moderate	high
Security	low	low	low	moderate	high	high
Extensibility	low	low	low	moderate	high	moderate

A. HTTP

Advantages:

- supports RESTful architecture with URI addressing
- supports distributed tracing to identify failures across a distributed system architecture
- can work behind closed firewalls
- uses a push approach which involves a persistent connection between client and server

Disadvantages:

- header size is large: this adds processing overhead for devices particularly constrained IoT devices
- high network latency: this makes the protocol not suitable for real-time or mission-critical IoT systems
- uses text not binary encoding
- protocol does not provide any QoS support
- protocol is not easily scalable
- computational overhead due to encrypting and decrypting (secure) messages

B. CoAP

Advantages:

- supports RESTful architecture with URI addressing
- protocol is very suitable for low-power or constrained IoT devices
- supports dynamic resource discovery
- protocol can be used for M2M communication
- reduced header size (4 byte): CoAP packets are much smaller than HTTP
- due to UDP, CoAP can be used over packet-based technologies (e.g. SMS) on mobile networks

Disadvantages:

- protocol has very limited quality of service levels
- protocol is not suitable for device-to-cloud communication (suitable for device-to-device only)
- protocol has limited level of encryption because of UDP (SSL and TLS are not available, only DTLS)
- device-to-device protocol: protocol has no support for broadcasting capabilities (only through extensions)
- protocol is not easily scalable or extensible
- protocol is vulnerable to spoofing and malicious attacks; an endpoint can freely read/write messages in a constrained network

C. MQTT

Advantages:

- has a transient data message transmission cycle
- good for cloud-based IoT applications (D2C)
- a lightweight protocol and works fairly well over constrained networks
- provides an adequate level of QoS support (0, 1, 2)
- protocol supports asynchronous messaging
- protocol is an event-driven which enables an IoT system to scale up (or down)
- protocol can be used for M2M communication
- supported by many IoT cloud providers (Table 3)

Disadvantages:

- does not support large payloads
- topic names are often long; inappropriate for low-rate wireless personal area networks (LR-WPANs)
- unsuitable for IoT devices that require sending multimedia content (e.g. audio, images or videos)
- protocol is not suitable for device-to-device communication (suitable for device-to-cloud only)
- lack of encryption; can use TLS/SSL for security and encryption, however, extra connection overhead
- no dynamic discovery (discovery based on topics) and broker can be a Single Point of Failure (SPoF)
- MQTT clients needs to support TCP; connections remain open with brokers (always on); limited sleep mode for constrained devices

D. AMQP

Advantages:

- event-driven protocol which enables some scaling up
- efficient workload distribution via queues (maximizes scalability)
- communication multiplexing: more than one session can be carried out in the same connection
- supports [distributed] transactions: messages can be published as transactions
- offline fetching: clients can fetch data when offline
- supports message header annotations

Disadvantages:

- reliability: only limited to what is provided by transport protocol
- discovery: offers no dynamic device discovery
- no multicast: messaging depends on connections between clients and brokers
- broker can be Single Point of Failure (SPoF)
- message flow can be slow and often complex

- standard does not provide mechanisms extensions with other protocols (e.g. majordomo protocol)
- standard was first developed in the finance sector, was not intended for M2M or IoT

E. XMPP

Advantages:

- designed to work in distributed client/server environments
- supports Simple Authentication and SASL
- based on XML which makes is extremely extensible

Disadvantages:

- does not offer end-to-end encryption
- no QoS support; no message delivery guarantees
- XML message format adds extra overhead in terms of IoT device processing and communication
- server can be Single Point of Failure (SPoF)
- computational overhead due to encrypting and decrypting messages

F. DDS

Advantages:

- scalability: can scale up to a large number of queues
- provides near real-time communication
- supports interoperable data formats
- supports multiple data types for data objects
- QoS support: offers a rich set of QoS policies
- efficiency: 1-to-1 latency as low as 30 microseconds
- comprehensive security mechanism through the DDS Security Model and SPIs
- no single point of failure (SPoF)
- end-to-end security and logging capabilities
- supports peer-to-peer (e.g. D2D), D2C, C2C

Disadvantages:

- implementation libraries are extremely large in size (sometimes over 2GB)
- complexity: can be quite difficult to implement
- not suitable for edge-based IoT devices with constrained processing and computing capabilities

VIII. CONCLUSION

Basic similarities among the several IoT messaging protocols that exist today suggest the potential that they can coexist throughout the design and deployment of IoT systems. Given the diversity among IoT device types and the protocols they can support, it is not uncommon that IoT systems may employ several protocols to be used for IoT data exchange. Due to the fact that messaging protocols are key components for the connectivity of IoT devices, understanding the strengths and weaknesses of each protocol plays an increasingly important role in choosing an appropriate protocol to use in deploying IoT systems while reducing maintenance costs.

In this paper, we presented a thorough investigation of the plurality of messaging protocols that exist today for building Internet of Things (IoT) systems. Throughout the paper, we examined six common messaging protocols that are used in the development of IoT systems including HTTP, CoAP, MQTT, AMQP, DDS and XMPP. Although each of these messaging protocols has a different primary focus, they share key similarities that encouraged us to explore common features for possible interoperability or coexistence of these protocols within IoT systems. In addition, we identified key communication tasks such as interfacing, exchange management, addressing, security, network management, fault tolerance, error detection, synchronization and protocol representation, among many others. Furthermore, we described fundamental protocol characteristics that need to be considered when designing and deploying IoT systems.

As IoT systems proliferate, ineffectively choosing an appropriate communication protocol makes it increasingly challenging to build reliable, scalable, interoperable and secure systems. Understanding the details of how these protocols are similar to each other presents an opportunity for the potential of protocol interoperability. In addition, understanding the context to which these applications protocols can be applied to IoT systems' deployments is critical. By investigating the protocols' distinctive approaches for building IoT systems, it is then possible to determine whether these standards may merge or coexist, particularly given the limitations exhibited in terms of essential design features for building IoT systems. For future work, we plan to conduct performance comparison of the existing messaging protocols employing a series of IoT use cases. In addition, we plan to investigate HTTP 2.0 and WebSocket as part of the messaging protocols that can be used for IoT data streams in IoT systems.

REFERENCES

[1] *MQTT Protocol*. Accessed: Apr. 18, 2020. [Online]. Available: http://mqtt.org 2020

[2] S. Imane, M. Tomader, and H. Nabil, "Comparison between CoAP and MQTT in smart healthcare and some threats," in *Proc. Int. Symp. Adv. Electr. Commun. Technol. (ISAECT)*, Nov. 2018, pp. 1–4.

[3] N. Naik, "Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP," in *Proc. IEEE Int. Syst. Eng. Symp. (ISSE)*, Vienna, Austria, Oct. 2017, pp. 1–7

[4] D. Thangavel, X. Ma, A. Valera, H.-X. Tan, and C. K.-Y. Tan, "Performance evaluation of MQTT and CoAP via a common middleware," in *Proc. IEEE 9th Int. Conf. Intell. Sensors, Sensor Netw. Inf. Process. (ISSNIP)*, Apr. 2014, pp. 1–6.

[5] U. Tandale, B. Momin, and D. P. Seetharam, "An empirical study of application layer protocols for IoT," in *Proc. Int. Conf. Energy, Commun., Data Analytics Soft Comput. (ICECDS)*, Aug. 2017, pp. 2447–2451.

[6] P. Thota and Y. Kim, "Implementation and comparison of M2M protocols for Internet of Things," in *Proc. 4th Intl Conf. Appl. Comput. Inf. Technol./3rd Int. Conf. Comput. Sci./Intell. Appl. Inform./1st Int. Conf. Big Data, Cloud Comput., Data Sci. Eng. (ACIT-CSII-BCD)*, Dec. 2016, pp. 43–48.

[7] J. Dizdarević, F. Carpio, A. Jukan, and X. Masip-Bruin, "A survey of communication protocols for Internet of Things and related challenges of fog and cloud computing integration," *ACM Comput. Surv.*, vol. 51, no. 6, p. 116, 2019.

[8] *International Organization for Standardization (ISO), Open Systems Interconnection (OSI) Standard (35.100)*. Accessed: Apr. 18, 2020. [Online]. Available: https://www.iso.org/ics/35.100/x

[9] *The Internet Society, Hypertext Transfer Protocol (HTTP)*. Accessed: Apr. 18, 2020. [Online]. Available: https://tools.ietf.org/html/rfc2616

[10] *HTTP version 3.0 (H3) IETF Draft*. Accessed: Apr. 18, 2020. https://quicwg.org/base-drafts/draft-ietf-quic-http.html

[11] *Constrained Application Protocol (CoAP) Standard*. Accessed: Apr. 18, 2020. [Online]. Available: https://tools.ietf.org/html/rfc7252

[12] *Message Queue Telemetry Transport (MQTT)*. Accessed: Apr. 18, 2020. [Online]. Available: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html
[13] *IBM Watson IoT Platform*. Accessed: Apr. 18, 2020. [Online]. Available: https://developer. ibm.com/tv/mqtt-client-library-watson-iot-platform
[14] *Microsoft Azure IoT Hub*. Accessed: Apr. 18, 2020. [Online]. Available: https://docs. microsoft.com/en-us/azure/iot-hub/iot-hub-mqtt-support
[15] *Google Cloud IoT Core*. Accessed: Apr. 18, 2020. [Online]. Available: https://cloud.google.com/iot-core
[16] *Bosch IoT Hub*. Accessed: Apr. 18, 2020. [Online]. Available: https://developer.bosch-iot-suite.com/service/hub
[17] *AWS IoT*. Accessed: Apr. 18, 2020. [Online]. Available: https://docs.aws.amazon.com/iot/ latest/developerguide/mqtt.html
[18] *IoTivity*. Accessed: Apr. 18, 2020. [Online]. Available: https://iotivity.org 2020
[19] *Advanced Message Queuing Protocol (AMQP)*. [Online]. Available: https://www.amqp.org
[20] A. Foster. (2015). *Messaging Technologies for the Industrial Internet and the Internet of Things Whitepaper*. PrismTech. Accessed: Apr. 18, 2020. [Online]. Available: https://www.smartindustry.com/assets/Uploads/SI-WP-Prismtech-Messaging-Tech.pdf
[21] *OMG Data-Distribution Service for Real-Time Systems (DDS)*. Apr. 18, 2020. [Online]. Available: https://omg.org/omg-dds-portal
[22] *The Real-Time Publish-Subscribe Protocol (RTPS)*. Apr. 18, 2020. [Online]. Available: https://www.omg.org/spec/DDSI-RTPS/2.3
[23] *Extensible Messaging and Presence Protocol (XMPP)*. Apr. 18, 2020. [Online]. Available: https://xmpp.org
[24] S. L. Keoh, S. S. Kumar, and H. Tschofenig, "Securing the Internet of Things: A standardization perspective," *IEEE Internet Things J.*, vol. 1, no. 3, pp. 265–275, Jun. 2014.
[25] S. Raza, H. Shafagh, K. Hewage, R. Hummen, and T. Voigt, "Lithe: Lightweight secure CoAP for the Internet of Things," *IEEE Sensors J.*, vol. 13, no. 10, pp. 3711–3720, Oct. 2013.
[26] A. Minteer, *Analytics for the Internet of Things (IoT): Intelligent Analytics for Your Intelligent Devices*. Birmingham, U.K.: Packt Publishing, 2017.
[27] S.-M. Chun, H.-S. Kim, and J.-T. Park, "CoAP-based mobility management for the Internet of Things," *Sensors*, vol. 15, no. 7, pp. 16060–16082, 2015.
[28] *Designing MQTT Topics for AWS IoT Core*. Accessed: Apr. 18, 2020. [Online]. Available: https://d1.awsstatic.com/whitepapers/Designing_MQTT_Topics_for_AWS_IoT_Core.pdf
[29] *Devices Profile for Web Services (DPWS)*. Accessed: Apr. 18, 2020. [Online]. Available: http://docs.oasis-open.org/ws-dd/ns/dpws/2009/01
[30] *XEP-0072: SOAP Over XMPP*. Accessed: Apr. 18, 2020. [Online]. Available: https://xmpp. org/extensions/xep-0072.html
[31] P. Saint-Andre, K. Smith, and R. Tronçon, *XMPP: The Definitive Guide: Building Real-Time Applications with Jabber Technologies*.Newton, MA, USA: O'Reilly Media, 2009.
[32] P. Waher. *XEP-0332*. Accessed: Apr. 18, 2020. [Online]. Available: https://xmpp.org/extensions/xep-0332.html
[33] *RESTful DDS*. Accessed: Apr. 18, 2020. [Online]. Available: https://code.google.com/arc hive/p/restful-dds/
[34] *RoboMQ over REST*. Accessed: Apr. 18, 2020. [Online]. Available: https://robomq.readthe docs.io/en/latest/REST
[35] *RabitMQ Multi-Protocol Support*. Accessed: Apr. 18, 2020. [Online]. Available: https:// cloudamqp.com/docs/protocols.html
[36] *Eclipse Paho*. Accessed: Apr. 18, 2020. [Online]. Available: https://www.eclipse.org/paho
[37] P. Gmytrasiewicz and E. Durfee, "Decision-theoretic recursive modeling and the coordinated attack problem," in *Proc. 1st Int. Conf. Artif. Intell. Planning Syst.* San Francisco, CA, USA: Morgan Kaufmann, 1992, pp. 88–95.
[38] *XEP-0198 Stream Management*. Accessed: Apr. 18, 2020. [Online]. Available: https://xmpp. org/extensions/xep-0198.html
[39] *Trace Context: MQTT Protocol*. Accessed: Apr. 18, 2020. [Online]. Available: https://w3c. github.io/trace-context-mqtt
[40] *Trace Context: AMQP Protocol*. Accessed: Apr. 18, 2020. [Online]. Available: https://w3c. github.io/trace-context-amqp
[41] *Connext DDS Micro*. Accessed: Apr. 18, 2020. [Online]. Available: https://www.rti.com/ products/connext-dds-micro
[42] *IEEE Standard for Utility Industry Metering Communication Protocol Application Layer (End Device Data Tables)*. Standaard 1377-2012, Accessed: Apr. 18, 2020. [Online]. Available: https://standards.ieee.org/standard/ 1377-2012.html
[43] B. AL-Madani and H. Ali, "Data distribution service (DDS) based implementation of smart grid devices using ANSI C12.19 standard," *Procedia Comput. Sci.*, vol. 110, pp. 394–401, 2017.
[44] *AMQP Advanced Message Queuing Protocol, Protocol Specification*. Accessed: Apr. 18, 2020. [Online]. Available: https://www.rabbitmq.com/res ources/specs/amqp0-9-1.pdf
[45] *CloudAMQP*. Accessed: Apr. 18, 2020. [Online]. Available: https://www.cloudamqp.com/ docs/index.html
[46] *Data Distribution Service (DDS) version 1.4*. Accessed: Apr. 18, 2020. [Online]. Available: https://www.omg.org/spec/DDS/1.4/PDF
[47] *DDS Interoperability Wire Protocol Specification version 2.3*. Accessed: Apr. 18, 2020. [Online]. Available: https://www.omg.org/spec/DDSI-RTPS/About-DDSI-RTPS
[48] *DDS Security Version 1.0*. Accessed: Apr. 18, 2020. [Online]. Available: https://www.omg. org/spec/DDS-SECURITY/1.0/PDF
[49] *CoAP RFC7252*. Accessed: Apr. 18, 2020. [Online]. Available: https://tools.ietf.org/html/rfc7252
[50] *CoAPSharp*. Accessed: Apr. 18, 2020. [Online]. Available: http://idaax.com/coapsharp
[51] *CoAP.NET*. Accessed: Apr. 18, 2020. [Online]. Available: http://open.smeshlink.com/CoAP.NET
[52] *Waher.Networking.CoAP*. Accessed: Apr. 18, 2020. [Online]. Available: https://github.com /PeterWaher/IoTGateway/tree/master/Networking/Waher.Networking.CoAP
[53] *Eclipse Californium*. Accessed: Apr. 18, 2020. [Online]. Available: https://www.eclipse.org /californium, Accessed on: Apr. 18, 2020
[54] *nCoAP*. Accessed: Apr. 18, 2020. [Online]. Available: https://github.com/okleine/nCoAP
[55] *Node-CoAP*. Accessed: Apr. 18, 2020. [Online]. Available: https://github.com/mcollina/node-coap
[56] *Aiocoap*. Accessed: Apr. 18, 2020. [Online]. Available: https://github.com/chrysn/aiocoap
[57] *CoAPthon*. Accessed: Apr. 18, 2020. [Online]. Available: https://github.com/Tanganelli/CoAPthon
[58] *txThings*. Accessed: Apr. 18, 2020. [Online]. Available: https://github.com/mwasilak/txThings
[59] *Go-CoAP*. Accessed: Apr. 18, 2020. [Online]. Available: https://github.com/go-ocf/go-coap
[60] *AMQP.Net Lite*. Accessed: Apr. 18, 2020. [Online]. Available: https://github.com/Azure/amqpnetlite
[61] *NET RabbitMQ*. Accessed: Apr. 18, 2020. [Online]. Available: https://www.rabbitmq.com/dotnet.html
[62] *Spring AMQP*. Accessed: Apr. 18, 2020. [Online]. Available: https://spring.io/projects/spring-amqp
[63] *RabbitMQ Java Client Library*. Accessed: Apr. 18, 2020. [Online]. Available: https://rabbit mq.com/java-client.html
[64] *amqplib*. Accessed: Apr. 18, 2020. [Online]. Available: https://npmjs.com/package/amqplib
[65] *amqp.node*. Accessed: Apr. 18, 2020. [Online]. Available: https://github.com/squaremo/amqp.node
[66] *rabbit.js*. Accessed: Apr. 18, 2020. [Online]. Available: https://github.com/squaremo/rabbit.js
[67] *Pika*. Accessed: Apr. 18, 2020. [Online]. Available: https://github.com/pika/pika
[68] *RabbitPy*. Accessed: Apr. 18, 2020. [Online]. Available: https://github.com/gmr/rabbitpy
[69] *Celery*. Accessed: Apr. 18, 2020. [Online]. Available: http://docs.celeryproject.org/en/latest
[70] *aioamqp*. Accessed: Apr. 18, 2020. [Online]. Available: https://github.com/Polyconseil/aioamqp
[71] *Go RabbitMQ Client Library*. Accessed: Apr. 18, 2020. [Online]. Available: https://github. com/streadway/amqp
[72] *Apache Qpid*. Accessed: Apr. 18, 2020. [Online]. Available: https://qpid.apache.org/
[73] *ActiveMQ*. Accessed: Apr. 18, 2020. [Online]. Available: https://activemq.apache.org/
[74] *Azure Event Hubs*. Accessed: Apr. 18, 2020. [Online]. Available: https://docs.microsoft. com/en-us/azure/service-bus-messaging/service-bus-amqp-protoc ol-guide
[75] *Solace PubSub+*. Accessed: Apr. 18, 2020. [Online]. Available: https://docs.solace.com /Open-APIs-Protocols/AMQP/AMQP-get-started.htm
[76] *RabitMQ*. Accessed: Apr. 18, 2020. [Online]. Available: https://www.rabbitmq.com/ protocols.html
[77] *StormMQ*. [Online]. Available: http://www.stormmq.com/

[78] *Red Hat AMQ*. Accessed: Apr. 18, 2020. [Online]. Available: https://www.redhat.com/ en/technologies/jboss-middleware/amq
[79] *RTI Connext DDS*. Accessed: Apr. 18, 2020. [Online]. Available: https://rti.com/products
[80] *RTI Connector*. Accessed: Apr. 18, 2020. [Online]. Available: https://community.rti.com/glossary-tags/nodejs
[81] *OpenDDS*. Accessed: Apr. 18, 2020. [Online]. Available: https://opendds.org
[82] *artalk-xmpp*. Accessed: Apr. 18, 2020. [Online]. Available: https://github.com/araditc/Artalk.Xmpp
[83] *Matrix vNext*. Accessed: Apr. 18, 2020. [Online]. Available: https://matrix-xmpp.io/
[84] *Waher Networking*. Accessed: Apr. 18, 2020. [Online]. Available: https://github.com/Peter Waher/IoTGateway/tree/master/Networking/Waher.Networking.XMPP
[85] *Smack*. Accessed: Apr. 18, 2020. [Online]. Available: https://igniterealtime.org/projects/smack/
[86] *Stroke*. Accessed: Apr. 18, 2020. [Online]. Available: https://swift.im/swiften.html
[87] *xmpp.js*. Accessed: Apr. 18, 2020. [Online]. Available: https://github.com/xmppjs/xmpp.js
[88] *aioxmpp*. Accessed: Apr. 18, 2020. [Online]. Available: https://github.com/horazont/aioxmpp
[89] *Openfire*. Accessed: Apr. 18, 2020. [Online]. Available: https://igniterealtime.org/projects/ openfire
[90] *Tigase XMPP Server*. Accessed: Apr. 18, 2020. [Online]. Available: https://tigase.net/xmpp-server
[91] *MongooseIM*. Accessed: Apr. 18, 2020. [Online]. Available: https://erlang-solutions.com /products/mongooseim,html
[92] *IoT Broker*. Accessed: Apr. 18, 2020. [Online]. Available: https://waher.se/Broker.md
[93] *RabbitMQ-Common*. Accessed: Apr. 18, 2020. [Online]. Available: https://github.com/ rabbitmq/rabbitmq-common/blob/master/include/rabbit.hrl#L255
[94] *IronMQ*. Accessed: Apr. 18, 2020. [Online]. Available: https://www.iron.io/mq
[95] *AmazonSQS*. Accessed: Apr. 18, 2020. [Online]. Available: https://aws.amazon.com/sqs/
[96] *RTI TypeObject*. Accessed: Apr. 18, 2020. [Online]. Available: https://community.rti. com/kb/how-send-big-typecodes-and-typeobjects
[97] *CoAP-Lite*. Accessed: Apr. 18, 2020. [Online]. Available: https://crates.io/crates/coap-lite
[98] C. Strîmbei, O. Dospinescu, R. Strainu, and A. Nistor, "Software architectures-present and visions," *Inform. Econ.*, vol. 19, no. 4, pp. 13–27, 2015.
[99] *Red Hat Enterprise MRG*. Accessed: Apr. 18, 2020. [Online]. Available: http://redhat.com/ products/mrg
[100] *OpenMQ*. Accessed: Apr. 18, 2020. [Online]. Available: http://www.openamq.org
[101] *XEP-0076: Malicious Stanzas*. Accessed: Apr. 18, 2020. [Online]. Available: https://xmpp. org/extensions/xep-0076.html
[102] *XEP-0030: Service Discovery*. Accessed: Apr. 18, 2020. [Online]. Available: https://xmpp. org/extensions/xep-0030.html
[103] *XEP-0080: User Location*. Accessed: Apr. 18, 2020. [Online]. Available: https://xmpp. org/extensions/xep-0080.html
[104] *Extensible and Dynamic Topic Types for DDS*. Accessed: Apr. 18, 2020. [Online]. Available: https://omg.org/spec/DDS-XTypes/1.3/Beta1/PDF
[105] *RabbitMQ: Per-Queue Message TTL in Queues*. Accessed: Apr. 18, 2020. [Online]. Available: https://rabbitmq.com/ttl.html#per-queue-message-ttl
[106] *DDS at Volkswagen*. Accessed: Apr. 18, 2020. [Online]. Available: https://info.rti.com/ hubfs/Collateral_2017/Customer_Snapshots/RTI_Customer-Snaps hot_60016_Volkswagen_V6_0718.pdf
[107] *Future of Automotive Industry*. Accessed: Apr. 18, 2020. [Online]. Available: https://rti.com/ blog/the-future-of-automotive
[108] *DDS Security Library*. Accessed: Apr. 18, 2020. [Online]. Available: https://github.com/ ARM-software/libddssec
[109] *TrustZone*. Accessed: Apr. 18, 2020. [Online]. Available: https://developer.arm.com/ip-pro ducts/security-ip/trustzone
[110] *XEP-0235: Authorization Tokens*. Accessed: Apr. 18, 2020. [Online]. Available: https://xmpp.org/extensions/attic/xep-0235-0.2.html
[111] B. Mishra, "Performance evaluation of MQTT broker servers," in *Computational Science and Its Applications* (Lecture Notes in Computer Science), vol. 10963, O. Gervasi, Ed. Cham, Switzerland: Springer, 2018.
[112] Z. Laaroussi, R. Morabito, and T. Taleb, "Service provisioning in vehicular networks through edge and cloud: An empirical analysis," in *Proc. IEEE Conf. Standards for Commun. Netw. (CSCN)*, Oct. 2018, pp. 1–6.
[113] A. Larmo, A. Ratilainen, and J. Saarinen, "Impact of CoAP and MQTT on NB-IoT system performance," *Sensors*, vol. 19, no. 1, p. 7, 2019.
[114] S. Profanter, A. Tekat, K. Dorofeev, M. Rickert, and A. Knoll, "OPC UA versus ROS, DDS, and MQTT: Performance evaluation of industry 4.0 protocols," in *Proc. IEEE Int. Conf. Ind. Technol. (ICIT)*, Feb. 2019, pp. 955–962.
[115] M. Zorkany, K. F., and A. Yahya, "Performance evaluation of IoT messaging protocol implementation for E-Health systems," *Int. J. Adv. Comput. Sci. Appl.*, vol. 10, no. 11, pp. 412–419, 2019.
[116] M. Martí, C. Garcia-Rubio, and C. Campo, "Performance evaluation of CoAP and MQTT_SN in an IoT environment," *Proceedings*, vol. 31, no. 1, p. 49, 2019.
[117] *Cisco Unified Presence 8.0*. Accessed: Apr. 18, 2020. [Online]. Available: https://www.cisco.com/c/en/us /products/collateral/unified-communications/unified-presence/data _sheet_c78-586995.html
[118] P. Cui, *Comparison of IoT Application Layer Protocols*. Auburn, AL, USA: Auburn Univ., 2017.
[119] Q. Hassan, *Internet of Things A to Z: Technologies and Applications*. Hoboken, NJ, USA: Wiley, 2018.
[120] S. Al-Sarawi, M. Anbar, K. Alieyan, and M. Alzubaidi, "Internet of Things (IoT) communication protocols: Review," in *Proc. 8th Int. Conf. Inf. Technol. (ICIT)*, May 2017, pp. 685–690.
[121] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, "Internet of Things: A survey on enabling technologies, protocols, and applications," *IEEE Commun. Surveys Tuts.*, vol. 17, no. 4, pp. 2347–2376, 4th Quart., 2015.
[122] N. Kushalnagar, G. Montenegro, and C. Schumacher, *IPv6 Over Low-Power Wireless Personal Area Networks (6LoWPANs): Overview, Assumptions, Problem Statement, and Goals*, document RF 4919, vol. 10, 2007.
[123] G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler, *Transmission of IPv6 packets over IEEE 802.15.4 Networks*, Internet Proposed Standard RFC document 4944, 2007.
[124] A. Souri, A. Hussien, M. Hoseyninezhad, and M. Norouzi, "A systematic review of IoT communication strategies for an efficient smart environment," *Trans. Emerg. Telecommun. Technol.*, pp. 2161–3915, Aug. 2019.
[125] D. Mocrii, Y. Chen, and P. Musilek, "IoT-based smart homes: A review of system architecture, software, communications, privacy and security," *Internet Things*, vols. 1–2, pp. 81–98, Sep. 2018.
[126] P. Lopez, D. Fernandez, A. J. Jara, and A. F. Skarmeta, "Survey of Internet of Things technologies for clinical environments," in *Proc. 27th Int. Conf. Adv. Inf. Netw. Appl. Workshops*, Mar. 2013, pp. 1349–1354.
[127] P. Fraga-Lamas, T. Fernández-Caramés, M. Suárez-Albela, L. Castedo, and M. González-López, "A review on Internet of Things for defense and public safety," *Sensors*, vol. 16, no. 10, p. 1644, 2016.
[128] J. Hui, D. Culler, and S. Chakrabarti. (Jan. 2009). *6LoWPAN: Incorporating IEEE 802.15.4 into the IP Architecture*. White Paper #3, Internet Protocol for Smart Objects (IPSO) Alliance. [Online]. Available: http://www.ipso-alliance.org
[129] A. Juels, "RFID security and privacy: A research survey," *IEEE J. Sel. Areas Commun.*, vol. 24, no. 2, pp. 381–394, Feb. 2006.
[130] I. Akyildiz, J. Xie, and S. Mohanty, "A Survey on Mobility Management in Next Generation All-IP Based Wireless Systems," *IEEE Wireless Commun. Mag.*, vol. 11, no. 4, pp. 16–28, 2004.
[131] R. Khan, S. U. Khan, R. Zaheer, and S. Khan, "Future Internet: The Internet of Things architecture, possible applications and key challenges," in *Proc. 10th Int. Conf. Frontiers Inf. Technol.*, Dec. 2012, pp. 257–260.
[132] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of Things (IoT): A vision, architectural elements, and future directions," *Future Gener. Comput. Syst.*, vol. 29, no. 7, pp. 1645–1660, Sep. 2013.
[133] D. Yang, F. Liu, and Y. Liang, "A survey of the Internet of Things," in *Proc. 1st Int. Conf. E-Bus. Intell.*, 2010, pp. 358–366.
[134] Z. Sheng, S. Yang, Y. Yu, A. Vasilakos, J. Mccann, and K. Leung, "A survey on the ietf protocol suite for the Internet of Things: Standards, challenges, and opportunities," *IEEE Wireless Commun.*, vol. 20, no. 6, pp. 91–98, Dec. 2013.
[135] X. Xiaojiang, W. Jianli, and L. Mingdong, "Services and key technologies of the Internet of Things," *ZTE Commun.*, vol. 8, no. 2, pp. 26–29, 2010.
[136] N. Komninos, E. Philippou, and A. Pitsillides, "Survey in smart grid and smart home security: Issues, challenges and countermeasures," *IEEE Commun. Surveys Tuts.*, vol. 16, no. 4, pp. 1933–1954, 4th Quart., 2014.

[137] Waviot. *Smart Gas Metering*. Accessed: Apr. 18, 2020. [Online]. Available: https://waviot. com/iot/solutions/smart-metering/smart-gas-metering, Accessed on: Apr. 18, 2020
[138] A. Seferagić, J. Famaey, E. De Poorter, and J. Hoebeke, "Survey on wireless technology trade-offs for the industrial Internet of Things," *Sensors*, vol. 20, no. 2, p. 488, 2020.
[139] J. Akerberg, M. Gidlund, and M. Bjorkman, "Future research challenges in wireless sensor and actuator networks targeting industrial automation," in *Proc. 9th IEEE Int. Conf. Ind. Informat.*, Jul. 2011, pp. 410–415.
[140] A. Varghese and D. Tandur, "Wireless requirements and challenges in industry 4.0," in *Proc. Int. Conf. Contemp. Comput. Informat. (IC3I)*, Nov. 2014, pp. 634–638.
[141] *IBM SoftLayer*. Accessed: Apr. 18, 2020. [Online]. Available: https://www.ibm.com/cloud/ info/softlayer-is-now-ibm-cloud
[142] *Google Cloud Networking*. Accessed: Apr. 18, 2020. [Online]. Available: https://cloud. google.com/blog/products/networking/google-cloud-networking-in-depth-cloud-cdn
[143] Azure IoT Edge Gateway. *How an IoT Edge Device Can be Used as a Gateway*. Accessed: Apr. 18, 2020. [Online]. Available: https://docs.microsoft.com/en-us/azure/iot-edge/iot-edge-as-gateway
[144] A. Gyrard and M. Serrano, "A unified semantic engine for Internet of Things and smart cities: From sensor data to end-users applications," in *Proc. IEEE Int. Conf. Data Sci. Data Intensive Syst.*, Dec. 2015, pp. 718–725.
[145] A. J. Jara, A. C. Olivieri, Y. Bocchi, M. Jung, W. Kastner, and A. F. Skarmeta, "Semantic Web of things: An analysis of the application semantics for the IoT moving towards the IoT convergence," *Int. J. Web Grid Services*, vol. 10, nos. 2–3, p. 244, 2014.
[146] *Web of Things (WoT) Architecture*. Accessed: Apr. 18, 2020. [Online]. Available: https:// www.w3.org/TR/wot-architecture
[147] *Web of Things (WoT) Thing Description*. Accessed: Apr. 18, 2020. [Online]. Available: https://www.w3.org/TR/wot-thing-description
[148] F. Károly, P. Zoltán, M. Gergely, and S. Ferenc, "Data Interoperability Across IoT Domains," *Int. J. Comput.*, vol. 12, pp. 60–65, 2018.
[149] *Concise Binary Object Representation (CBOR)*. Accessed: Apr. 18, 2020. [Online]. Available: https://tools.ietf.org/html/rfc7049
[150] R. Startin. *Concise Binary Object Representation (CBOR)*. IETF. Accessed: Apr. 18, 2020. [Online]. Available: https://tools.ietf.org/html/rfc7049
[151] I. Al-Joboury and E. Al-Hemiary, "Performance analysis of Internet of Things protocols based fog/cloud over high traffic," *J. Fundam. Appl. Sci.*, vol. 10, no. 6S, pp. 176–181, 2018.
[152] J. Joshi, V. Rajapriya, S. R. Rahul, P. Kumar, S. Polepally, R. Samineni, and D. G. Kamal Tej, "Performance enhancement and IoT based monitoring for smart home," in *Proc. Int. Conf. Inf. Netw. (ICOIN)*, 2017, pp. 468–473.
[153] N. De Caro, W. Colitti, K. Steenhaut, G. Mangino, and G. Reali, "Comparison of two lightweight protocols for smartphone-based sensing," in *Proc. IEEE 20th Symp. Commun. Veh. Technol. Benelux (SCVT)*, Nov. 2013, pp. 1–6.
[154] Z. B. Babovic, J. Protic, and V. Milutinovic, "Web performance evaluation for Internet of Things applications," *IEEE Access*, vol. 4, pp. 6974–6992, 2016.
[155] *IoT Developer Surveys*. Accessed: Apr. 18, 2020. [Online]. Available: https://iot.eclipse.org/ community/resources/iot-surveys
[156] *2019 IoT Developer Survey*. Accessed: Apr. 18, 2020. [Online]. Available: https://iot.eclipse.org/community/resources/iot-surveys/assets/iot-developer-survey-2019.pdf
[157] J. L. Fernandes, I. C. Lopes, J. J. P. C. Rodrigues, and S. Ullah, "Performance evaluation of RESTful Web services and AMQP protocol," in *Proc. 5th Int. Conf. Ubiquitous Future Netw. (ICUFN)*, Jul. 2013, pp. 810–815.
[158] W. Gao, J. H. Nguyen, W. Yu, C. Lu, D. T. Ku, and W. G. Hatcher, "Toward emulation-based performance assessment of constrained application protocol in dynamic networks," *IEEE Internet Things J.*, vol. 4, no. 5, pp. 1597–1610, Oct. 2017.
[159] *Multicast DNS (mDNS)*. Accessed: Apr. 18, 2020. [Online]. Available: https://tools.ietf.org/ html/rfc6762
[160] *DNS-Based Service Discovery*. Accessed: Apr. 18, 2020. [Online]. Available: https://tools. ietf.org/html/rfc6763
[161] *Avahi*. Accessed: Apr. 18, 2020. [Online]. Available: https://www.avahi.org
[162] *Apple Bonjour*. Accessed: Apr. 18, 2020. [Online]. Available: https://en.wikipedia.org/ wiki/Bonjour_(software)
[163] *HyperCat (BSI)*. Accessed: Apr. 18, 2020. [Online]. Available: http://www.hypercat.io
[164] *Physical Web*. Accessed: Apr. 18, 2020. [Online]. Available: https://google.github.io/physical-web
[165] *Universal Plug and Play (UPnP)*. Accessed: Apr. 18, 2020. [Online]. Available: https://tools. ietf.org/html/rfc6970
[166] *Azure IoT Hub*. Accessed: Apr. 18, 2020. [Online]. Available: https://azure.microsoft.com/ en-us/services/iot-hub
[167] *Google IoT Core*. Accessed: Apr. 18, 2020. [Online]. Available: https://cloud.google.com/ iot/docs/concepts/overview
[168] *IBM Watson IoT Platform*. Accessed: Apr. 18, 2020. [Online]. Available: https://ibm.com/ internet-of-things/solutions/iot-platform/watson-iot-platform
[169] *AWS IoT Core*. Accessed: Apr. 18, 2020. [Online]. Available: https://aws.amazon.com/iot-core
[170] *Alibaba IoT Platform*. Accessed: Apr. 18, 2020. [Online]. Available: https://alibabacloud.com/product/iot
[171] *Oracle IoT*. Accessed: Apr. 18, 2020. [Online]. Available: https://www.oracle.com/internet-of-things
[172] *Siemens Mindsphere*. [Online]. Available: https://www.plm.automation.siemens.com/media/global/en/Siemens_MindSphere_Whitepaper_tcm27-9395.pdf
[173] *Bosch IoT Hub*. Accessed: Apr. 18, 2020. [Online]. Available: https://docs.bosch-iot-suite.com/hub/introduction/ossfoundation.html
[174] *Cisco Kinetic*. Accessed: Apr. 18, 2020. [Online]. Available: https://www.cisco.com/c/dam/ en/us/td/docs/cloud-systems-management/kinetic/tech_notes/kine tic-security.pdf
[175] *Eclipse Hono*. Accessed: Apr. 18, 2020. [Online]. Available: https://www.eclipse.org/ hono/docs/user-guide/mqtt-adapter
[176] *IoT 2020: Smart and Secure IoT Platform*. IEC. Accessed: Apr. 18, 2020. [Online]. Available: https://www.iec.ch/whitepaper/iotplatform

EYHAB AL-MASRI (Member, IEEE) received the B.Sc. and M.Sc. degrees from Florida International University and the Ph.D. degree from the University of Guelph. He is currently an Assistant Professor with the School of Engineering and Technology, University of Washington Tacoma. His research interests include distributed systems, the Internet of Things (IoT), fog computing, edge computing, cloud computing, service-oriented computing, and big data analytics.

KARAN RAJ KALYANAM received the B.Tech. degree in computer science and engineering degree from K L University, India, in 2017, and the M.Sc. degree in computer science and systems (MSCSS) from the School of Engineering and Technology, University of Washington Tacoma, in 2019. He has been a Jr. Data Scientist with Amazon, since 2019, where he is working in the big data domain—visualizing, analyzing and processing massive scale datasets along with predictive modeling to foresee events, and observing pattern in the data. His research interests include distributed systems, cloud computing, machine learning, and the Internet of Things.

JOHN BATTS is currently pursuing the B.Sc. degree in computer science and systems (CSS) with the School of Engineering and Technology, University of Washington Tacoma. His research interests include big data analytics and machine learning.

JONATHAN KIM is currently pursuing the B.Sc. degree in computer science and systems (CSS) with the School of Engineering and Technology, University of Washington Tacoma. His research interests include Internet of Things, edge computing and big data analytics.

SHARANJIT SINGH is currently pursuing the B.Sc. degree in computer science and systems (CSS) with the School of Engineering and Technology, University of Washington Tacoma. His research interests include the Internet of Things, cloud computing and services computing.

TAMMY VO is currently pursuing the B.Sc. degree in computer science and systems (CSS) with a minor in mathematics with the School of Engineering and Technology, University of Washington Tacoma. Her research interests include edge computing and cloud computing.

CHARLOTTE YAN is currently pursuing the double major B.Sc. degree in computer science and systems (CSS) and mathematics with the School of Engineering and Technology, University of Washington Tacoma. Her research interests include Internet of Things, data science, cloud computing and mathematical modeling.