Computer Networks, Volume 150, 26 February 2019, Pages 28-45

Implementation and Analysis of QUIC for MQTT

Puneet Kumar and Behnam Dezfouli
Internet of Things Research Lab, Department of Computer Engineering, Santa Clara University, USA
{pkumar,bdezfouli}@scu.edu

***Abstract*—Transport and security protocols are essential to ensure reliable and secure communication between two parties. For IoT applications, these protocols must be lightweight, since IoT devices are usually resource constrained. Unfortunately, the existing transport and security protocols – namely TCP/TLS and UDP/DTLS – fall short in terms of connection overhead, latency, and connection migration when used in IoT applications. In this paper, after studying the root causes of these shortcomings, we show how utilizing QUIC in IoT scenarios results in a higher performance. Based on these observations, and given the popularity of MQTT as an IoT application layer protocol, we integrate MQTT with QUIC. By presenting the main APIs and functions developed, we explain how connection establishment and message exchange functionalities work. We evaluate the performance of MQTTw/QUIC versus MQTTw/TCP using wired, wireless, and long-distance testbeds. Our results show that MQTTw/QUIC reduces connection overhead in terms of the number of packets exchanged with the broker by up to 56%. In addition, by eliminating half-open connections, MQTTw/QUIC reduces processor and memory usage by up to 83% and 50%, respectively. Furthermore, by removing the head-of-line blocking problem, delivery latency is reduced by up to 55%. We also show that the throughput drops experienced by MQTTw/QUIC when a connection migration happens is considerably lower than that of MQTTw/TCP.**

***Index Terms*—Internet of Things (IoT); Transport layer; Application layer; Latency; Security**

I. Introduction

The Internet of Things is the enabler of many applications, such as the smart home, smart cities, remote medical monitoring, and industrial control, by connecting a large number of sensors and actuators to the Internet. Existing studies predict that the number of connected devices will surpass 50 billion by 2020 [1]. To facilitate interconnection and software development, the communication between IoT devices usually employs a protocol stack similar to that of regular Internet-connected devices such as smartphones and laptops. Specifically, IP (or 6LowPAN [2]) and transport layer protocols are provided by various protocol stacks (e.g., μIP [3], LwIP [4]) to enable interconnectivity.

A. TCP and UDP

The primary responsibility of the transport layer is to support exchanging segments between the two end-to-end communicating applications. Among the transport layer protocols, TCP (Transport Layer Protocol) and UDP (User Datagram Protocol) are the most widely used, depending on the application at hand. TCP provides a reliable end-to-end connection and implements congestion control mechanisms to avoid buffer overflow at the receiver. During the past couple of decades, several improved versions of TCP have been proposed to address the increasing demand for throughput [5], [6]. However, these features impose high overhead in terms of connection establishment and resource (i.e., processor, memory, energy) utilization. UDP, on the other hand, does not provide any of the above-mentioned features and therefore, its overhead is significantly lower than that of TCP.

While throughput is the main performance metric for user traffic such as voice and video, the prevalent communication type of IoT, which is machine-to-machine (M2M), is characterized by short-lived bursts of exchanging small data chunks [7], [8]. In addition, compared to user devices such as smartphones and laptops, IoT devices are usually resource constrained in terms of processing, memory, and energy [9], [10]. Using TCP in IoT domains to satisfy reliability and security requirements, therefore, imposes high overhead. Specifically, the shortcomings of TCP when used in IoT applications are as follows:

– Connection startup latency is highly affected by the TCP handshake. This handshake requires 1 Round-Trip Time (RTT) for TCP and 2 or 3 RTTs when TLS (Transport Layer Security) is added to this protocol [11]. The overhead impact is even higher in IoT scenarios where unreliable wireless links cause frequent connection drops [12]. In these scenarios, imposing a high connection establishment overhead for the exchange of a small amount of data results in wasting the resources of devices. TCP Fast Open [13] seeks to address this problem by piggybacking data in SYN segments in repeated connections to the same server. This solution is not scalable since the TCP SYN segment can only fit a limited amount of data [14].
– IoT devices are often mobile, and as such, supporting connection migration is an essential requirement [15]–[18]. However, any change in network parameters (such as IP address or port) breaks the connection. In this case, either the connection must be re-established, or a gateway is required to reroute the data flow. Unfortunately, these solutions increase communication delay and overhead, which might not be acceptable in mission-critical applications such as medical monitoring [8].
– To preserve energy resources, IoT devices usually transition between sleep and awake states [19], [20]. In this case, a TCP connection cannot be kept open without employing *keep-alive* packets. These keep-alive mechanisms, however, increase resource utilization and bandwidth consumption. Without an external keep-alive mechanism, IoT devices are obliged to re-establish connections every time they wake from the sleep mode.
– In disastrous events such as unexpected reboots or a device crash, TCP connections between client and server might

end up out of state. This undefined state is referred to as TCP *half-open connections* [21]. A half-open connection consumes resources such as memory and processor time. In addition, it can impose serious threats such as SYN flooding [22], [23].
– If packets are dropped infrequently during a data flow, the receiver has to wait for dropped packets to be retransmitted in order to complete the packet re-ordering. This phenomena, which impedes packet delivery performance, is called the *head-of-line blocking* [24]–[26].

Despite the aforementioned shortcomings, several IoT application layer protocols rely on TCP, and some of them offer mechanisms to remedy these shortcomings. For example, MQTT [27] employs application layer keep-alive messages to keep the connection alive. This mechanism also enables MQTT to detect connection breakdown and release the resources.

Another transport layer protocol used in IoT networks is UDP. Generally, UDP is suitable for applications where connection reliability is not essential. Although this is not acceptable in many IoT scenarios, several IoT application layer protocols rely on UDP due to its lower overhead compared to TCP. These protocols usually include mechanisms to support reliability and block transmission (e.g., CoAP [28]).

B. TLS and DTLS

In addition to reliability, it is essential for IoT applications to employ cryptographic protocols to secure end-to-end data exchange over transport layer. TLS [29] is the most common connection-oriented and stateful client-server cryptographic protocol. Symmetric encryption in TLS enables authenticated, confidential, and integrity-preserved communication between two devices. The key for this symmetric encryption is generated during the TLS handshake and is unique per connection. In order to establish a connection, the TLS handshake can require up to two round-trips between the server and client. However, since connections might be dropped due to phenomenons such as sleep phases, connection migration, and packet loss, the overhead of establishing secure connections imposes high overhead. In order to address this concern, a lighter version of TLS for datagrams, named DTLS (Datagram Transport Layer Security) [30], has been introduced. Unlike TLS, DTLS does not require a reliable transport protocol as it can encrypt or decrypt out-of-order packets. Therefore, it can be used with UDP. Although the security level offered by DTLS is equivalent to TLS, some of the technical differences include: the adoption of stream ciphers is prohibited, and an explicit sequence number is included in every DTLS message. Compared to TLS, DTLS is more suitable for resource-constrained devices communicating through an unreliable channel. However, similar to TLS, the underlying mechanism in DTLS has been primarily designed for point-to-point communication. This creates a challenge to secure one-to-many connections such as broadcasting and multicasting. In addition, since DTLS identifies connections based on source IP and port number, it does not support connection migration [31]. Furthermore, DTLS handshake packets are large and may fragment each datagram into several DTLS records where each record is fit into an IP datagram. This can potentially cause record overhead [32].

C. Contributions

Given the shortcomings of UDP and TCP, we argue that the enhancement of transport layer protocols is a necessary step in the performance improvement of IoT applications. In order to address this concern, this paper presents the implementation and studies the integration of QUIC [33] with application layer to address these concerns. QUIC is a user space, UDP-based, stream-based, and multiplexed transport protocol developed by Google. According to [14], around 7% of the world-wide Internet traffic employs QUIC. This protocol offers all the functionalities required to be considered a connection-oriented transport protocol. In addition, QUIC solves the numerous problems faced by other connection-oriented protocols such as TCP and SCTP [34]. Specifically, the addressed problems are: reducing the connection setup overhead, supporting multiplexing, removing the head-of-line blocking, supporting connection migration, and eliminating TCP half-open connections. QUIC executes a cryptographic handshake that reduces the overhead of connection establishment by employing known server credentials learned from past connections. In addition, QUIC reduces transport layer overhead by multiplexing several connections into a single connection pipeline. Furthermore, since QUIC uses UDP, it does not maintain connection status information in the transport layer. This protocol also eradicates the head-of-line blocking delays by applying a lightweight data-structure abstraction called *streams*.

At present, there is no open source or licensed version of MQTT using QUIC. Current MQTT implementations (such as Paho [35]) rely on TCP/TLS to offer reliable and secure delivery of packets. Given the potentials of QUIC and its suitability in IoT scenarios, in this paper we implement and study the integration of MQTT with QUIC. First, since the data structures and message mechanisms of MQTT are intertwined with the built-in TCP and TLS APIs, it was necessary to redesign these data structures. The second challenge was to establish IPC (Inter-Process Communication) between QUIC and MQTT. MQTTw/TCP utilizes the available APIs for user space and kernel communication. However, there is no available API for QUIC and MQTT to communicate, as they are both user space processes. To address these challenges, we have developed new APIs, which are referred to as *agents*. Specifically, we implemented two types of agents: server-agent and client-agent, where the former handles the broker-specific operations and the latter handles the functionalities of publisher and subscriber. This paper presents all the functions developed and explains the connection establishment and message exchange functionalities by presenting their respective algorithms. The third challenge was to strip QUIC of mechanisms not necessary for IoT scenarios. QUIC is a web traffic protocol composed of a heavy code footprint (1.5GB [36]). We have significantly reduced the code footprint (to around 22MB) by eliminating non-IoT related code segments such as loop network and proxy backend support.

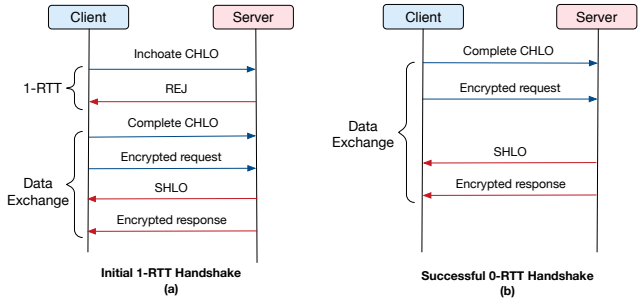

Fig. 1. The messages exchanged by QUIC during (a) 1-RTT and (b) 0-RTT connections.

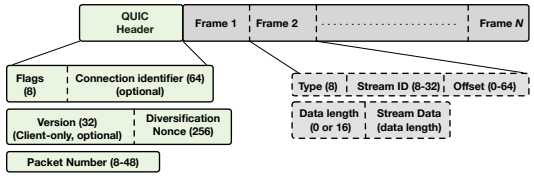

Fig. 2. The solid and dashed lines show the clear-text and encrypted parts of a QUIC packet, respectively. The non-encrypted part is used for routing and decrypting the encrypted part of the packet.

Three types of testbeds were used to evaluate the performance of MQTTw/QUIC versus MQTTw/TCP: wired, wireless, and long-distance. Our results show that, in terms of the number of packets exchanged during the connection establishment phase, MQTTw/QUIC offers a 56.2% improvement over MQTTw/TCP. By eliminating half-open connections, MQTTw/QUIC reduces processor and memory usage by up to 83.2% and 50.3%, respectively, compared to MQTTw/TCP. Furthermore, by addressing the head-of-line blocking problem, MQTTw/QUIC reduces message delivery latency by 55.6%, compared to MQTTw/TCP. In terms of connection migration, the throughput drop experienced by MQTTw/QUIC is significantly lower than that of MQTTw/TCP.

The rest of this paper is organized as follows. Section II explains the QUIC protocol along with its potential benefits in IoT applications. The implementation of QUIC for MQTT is explained in Section III. Performance evaluation and experimentation results are given in Section IV. Section V overviews the existing studies on QUIC and IoT application layer protocols. The paper is concluded in Section VI.

II. QUIC

QUIC employs some of the basic mechanisms of TCP and TLS, while keeping UDP as its underlying transport layer protocol. QUIC is in fact a combination of transport and security protocols by performing tasks including encryption, packet re-ordering, and retransmission. This section overviews the main functionalities of this protocol and justifies the importance of its adoption in the context of IoT.

A. Connection Establishment

QUIC combines transport and secure layer handshakes to minimize the overhead and latency of connection establishment. To this end, a dedicated reliable stream is provided for the cryptographic handshake. Figure 1(a) and (b) show the packets exchanged during the 1-RTT and 0-RTT connection establishment phases, respectively. Connection establishment works as follows:

– *First handshake*. In order to retrieve the server's configuration, the client sends an inchoate client hello (`CHLO`) message. Since the server is an alien to the client, the server must send a `REJ` packet. This packet carries the server configuration including the long-term Diffie-Hellman value, connection ID (`cid`), port numbers, key agreement, and initial data exchange. After receiving the server's configuration, the client authenticates the server by verifying the certificate chain and the signature received in the `REJ` message. At this point, the client sends a complete `CHLO` packet to the server. This message contains the client's ephemeral Diffie-Helman public value. This concludes the first handshake.
– *Final and repeat handshake*. After receiving the complete `CHLO` packet, the client has the initial keys for the connection and starts sending application data to the server. For 0-RTT, the client must initiate sending encrypted data with its initial keys before waiting for a reply from the server. If the handshake was successful, the server sends a server hello (`SHLO`) message. This concludes the final and repeat handshake.

Except some handshake and reset packets, QUIC packets are fully authenticated and partially encrypted. Figure 2 shows the non-encrypted and encrypted parts of the packet using solid and dotted lines, respectively. The non-encrypted packet header is used for routing and decrypting the packet content. The flags encode the presence of `cid` and the length of the Packet Number (PN) field, which are visible to read the subsequent fields.

B. Connection Migration

The QUIC connections are identified by a randomly generated 64-bit Connection Identifier (`cid`). A `cid` is allocated per connection and allows the clients to roam between networks without being affected by the changes in the network or transport layer parameters. As shown in Figure 2, `cid` resides in the header (non-encrypted part) and makes the clients independent of network address translation (NAT) and restoration of connections. The `cid` plays an important role in routing, specifically for connection identification purposes. Furthermore, using `cid`s enables multipath by probing a new path for connection. This process is called *path validation* [37]. During a connection migration, the end point assumes that the peer is willing to accept packets at its current address. Therefore, an end point can migrate to a new IP address without first validating the peer's IP address. It is possible that the new path does not support the current sending rate of the endpoint. In this case, the end point needs to reconstitute its congestion controller [38]. On the other hand, receiving non-probe packets [39] from a new peer address confirms that the peer has migrated to the new IP address.

C. Security

For transport layer encryption, MQTTw/TCP usually relies on TLS/SSL. The primary reasons why TLS/SSL cannot be used in QUIC were described in [40]. In short, the TLS security model uses one session key, while QUIC uses two session keys. This difference, in particular, enables QUIC to offer 0-RTT because data can be encrypted before the final key is set. Thus, the model has to deal with data exchange under multiple session keys [41]. Therefore, MQTTw/QUIC uses its own encryption algorithm named QUIC Crypto [42]. This algorithm decrypts packets independently to avoid serialized decoding dependencies. The signature algorithms supported by Crypto are ECDSA-SHA256 and RSA-PSS-SHA256.

D. Multiplexing

Unlike TCP, QUIC is adept in transport layer header compression by using multiplexing. Instead of opening multiple connections from the same client, QUIC opens several streams multiplexed over a single connection. Each connection is identified by a unique `cid`. The odd `cids` are for client-initiated streams and even `cids` are for server-initiated streams. A stream is a lightweight abstraction that provides a reliable bidirectional byte-stream. A QUIC stream can form an application message up to 2^{64} bytes. Furthermore, in the case of packet loss, the application is not prevented from processing subsequent packets. Multiplexing is useful in IoT applications where a large amount of data transfer is required per transaction. For example, this feature enhances performance for remote updates and industrial automation [43].

E. Flow and Congestion Control

Similar to TCP, QUIC implements a flow control mechanism to prevent the receiver's buffer from being inundated with data [14]. A slow TCP draining stream can consume the entire receiver buffer. This can eventually block the sender from sending any data through the other streams. QUIC eliminates this problem by applying two levels of flow control: (i) Connection level flow control: limits the aggregate buffer that a sender can consume across all the streams on a receiver. (ii) Stream level flow control: limits the buffer per stream level. A QUIC receiver communicates the capability of receiving data by periodically advertising the absolute byte offset per stream in *window update frames* for sent, received, and delivered packets.

QUIC incorporates a pluggable congestion control algorithm and provides a richer set of information than TCP [44]. For example, each packet (original or re-transmitted) carries a new Packet Number (`PN`). This enables the sender to distinguish between the re-transmitted and original ACKs, hence removing TCP's re-transmission *ambiguity problem*. QUIC utilizes a NACK based mechanism, where two types of packets are reported: the largest observed packet number, and the unseen packets with a packet number lesser than that of the largest observed packet. A receive timestamp is also included in every newly-acked ACK frame. QUIC's ACKs can also provide the delay between the receipt of a packet and its acknowledgement, which helps in calculating RTT. QUIC's ACK frames support up to 256 NACK ranges in opposed to the TCP's 3 NACK range [45]. This makes QUIC more resilient to packet reordering than TCP (with SACK). The congestion control algorithm of QUIC is based on TCP Reno to determine the pacing rate and congestion window size [45]. In addition, QUIC supports two congestion control algorithms: (i) Pacing Based Congestion Control Algorithm (PBCCA) [46], and (ii) TCP CUBIC [47]. The superior performance of QUIC's flow control over TCP for HTTP traffic has been demonstrated in the literature [48].

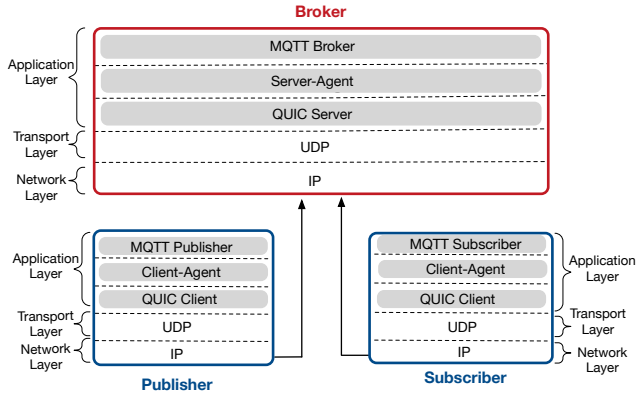

Fig. 3. The high-level architecture of the proposed implementation.

III. Integration and Implementation of MQTT with QUIC

This section presents the integration of MQTT with QUIC and is divided into six sub-sections. The first sub-section overviews the system architecture. The second sub-section describes the definitions, methods, and assumptions. The third and fourth sub-sections explain the operations of the APIs and functions developed for the broker and clients, respectively. We present the common APIs and functions, which are used by the broker and client in the fifth sub-section. A short discussion about code reduction is presented in the sixth sub-section. In order to simplify the discussions, we refer to the publisher and the subscriber as *client*. The presented implementation for the client and broker are based on the open-source Eclipse Paho and Mosquitto [35], respectively.

A. Architecture

Both MQTT and QUIC belong to the application layer. To streamline their integration, either the QUIC library (`ngtcp2` [49]) must be imported into MQTT, or new interfaces must be created. However, the former approach is not suitable for resource-constrained IoT devices as QUIC libraries are developed mainly for HTTP/HTTPS traffic, and therefore, impose a heavy code footprint. In our implementation, we chose the latter approach and built a customized broker and client interfaces for MQTT and QUIC. We refer to these interfaces as *agents*, which enable IPC between the MQTT and QUIC. Figure 3 shows a high level view of the protocol stack architecture.

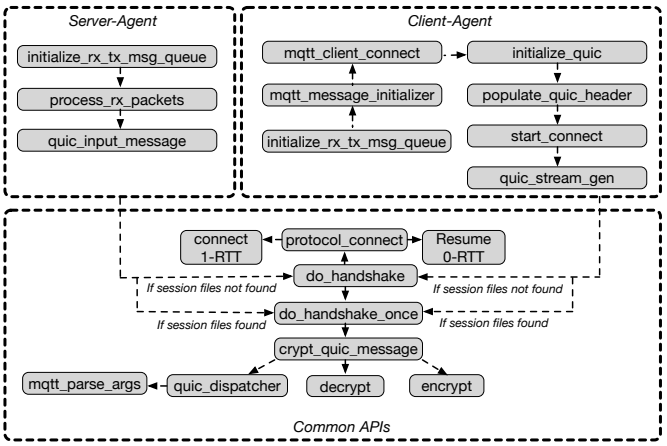

Fig. 4. Server-agent and client-agent are two entities between QUIC and MQTT. This figure represents the packet processing flow by the server and client agents.

Since QUIC uses UDP, connection-oriented features such as reliability, congestion control, and forward error correction are implemented in QUIC. In addition, QUIC incorporates cyptographic shields, such as IP spoofing protection and packet reordering [50]. To keep the QUIC implementation lightweight and abstracted, we segregated the implementation into two parts: (i) the QUIC client and server only deal with UDP sockets and streams; (ii) the agents deal with reliability and security.

Figure 4 shows the high-level view of the server and client agent implementations. The implemented entities are as follows: (i) *Server-Agent APIs and functions*: They handle server specific roles such as accepting incoming UDP connections, setting clients' state, and storing and forwarding packets to the subscribers based on topics. (ii) *Client-Agent APIs and functions*: They perform client-specific tasks such as opening a UDP connection, and constructing QUIC header and streams. (iii) *Common APIs and functions*: They are utilized by both the server and client. However, their tasks differ based on their roles.

After initializing the transmit and receive message queues by `initialize_rx_tx_msg_queue()`, the server and client agents process the messages differently. In the server-agent, the received message queue is fed to `quic_input_message()`. This API begins the handshake process to negotiate the session keys. However, if the client has contacted the server in the past, then `quic_input_message()` is directly available for the `crypt_quic_message()` to parse the incoming MQTT message. On the other hand, the client-agent first processes the MQTT message and then sends that message to the server by using `start_connect()`. Based on the communication history between the server and client, the `start_connect()` API either enters or skips the handshake process with the server. The common APIs handle the handshake control messages.

B. Definitions, Methods, and Assumptions

This section presents the primitives used to present and explain the implementations. Table I shows the main acronyms and notations used in the rest of this paper. We chose most of these notations based on the RFCs published relevant to this work.

Notation $\{0,1\}^*$ represents the set of all finite-length binary strings. If a and b are two uniquely decodable strings, then (a,b) simply represents the concatenation of both. If $\kappa \in \mathbb{N}$, then 1^κ represents the κ consecutive strings of 1 bits. The notation $s \xrightarrow{\$} \mathtt{S}$ represents the uniform random selection of s from a finite set $\mathtt{S}$. The set of integers $[1,.....n]$ is represented by $[n]$, where $n \in \mathbb{N}$. We also assume that a public key infrastructure (PKI) is available. This means that public keys are user-identity bounded, valid, and publicly known. Therefore, certificates and their verification are excluded from the implementation.

A *Digital Signature Scheme* (SS) with message space Msg is used by the broker during a connection establishment to authenticate certain data. The scheme is defined as follows,

$$\mathtt{SS} = (\mathtt{Kg}, \mathtt{Sign}, \mathtt{Ver}) \tag{1}$$

where Kg, Sign and Ver are the randomization key generation algorithm, signing algorithm, and verification algorithm, respectively. The input of Kg is the security parameter λ and its output is a public and secret key pair, as follows,

$$\mathtt{Kg}(\lambda) \xrightarrow{\$} (pk, sk) \tag{2}$$

The signing algorithm returns a signature,

$$\mathtt{Sign}(sk, m) \xrightarrow{\$} \sigma \tag{3}$$

where sk is the secret key and $m \in \mathtt{Msg}$. The verification algorithm is denoted as follows,

$$\mathtt{Ver}(pk, m, \sigma) \rightarrow p \tag{4}$$

where pk is the public key and $p \in \{0,1\}$. The output value p, which is a bit, shows whether the signature is valid or invalid.

The requirement for correctness of SS is Ver(pk,m, Sign(sk,m)) = 1 for every $m \in$ Msg and Kg(λ). Correctness is defined by the requirement that the input of one party's m_{send} be equal to the output of the other party's m_{get}.

Secure channel implementation is based on an authenticated encryption with associative data schemes (AEAD) [52]. AEAD consists of two algorithms: $\mathcal{E}$ and D. First, $\mathcal{E}$ is a deterministic encryption algorithm defined as follows,

$$\mathcal{E}(k, nonc, m, H) \rightarrow c \tag{5}$$

where c is ciphertext, $nonc \in \{0,1\}^n$, message $m \in \{0,1\}^*$, $H \in \{0,1\}^*$ is an additional authenticated data, and key k is defined as

$$\{0,1\}^\lambda \xrightarrow{\$} \kappa \tag{6}$$

Second, D is a deterministic decryption algorithm defined as,

$$\mathtt{D}(\kappa, nonc, H, \mathcal{E}) \rightarrow pl \ or \ \perp \tag{7}$$

where pl is plaintext.

The correctness requirement of AEAD is D(κ,nonc,H,$\mathcal{E}$(κ,nonc,H,m)) $\equiv$ m for all $\kappa \in \{0,1\}^\lambda$, nonc $\in \{0,1\}^n$, H and $m \in \{0,1\}^*$.

TABLE I
KEY NOTATIONS AND ABBREVIATIONS

Variable	Description
$\perp$	Represents a rejected message
λ	Protocol-associated security parameter used to derive the session keys
κ	Input key $\{0,1\}^{\kappa} \xrightarrow{\$} \kappa$
$\mathcal{E}$	Deterministic algorithm used in AEAD
$\mathcal{H}$	SHA-256 function
σ	Signature used in digital signature scheme
(pk, sk)	Key pair representing public key and secret key
a	Derived by $\{primes\ of\ size\ \lambda\} \xrightarrow{\$} a$
$action\_flag$	Flag used to determine if packet is subject to transmission or processing
b	Derived by $\{generators\ of\ \mathbb{Z}_a\} \xrightarrow{\$} b$
C	Client (i.e., publisher or subscriber)
CHLO	Client hello message in QUIC, a.k.a., inchoate hello (c_i_hello)
cid	Connection identifier
$client\_info$	Struct used in storing client info such as cid, socket, etc.
$client\_initial$	Client's initial state, which is the state immediately after receiving REJ message from server
D	Deterministic algorithm used in AEAD
DH	Diffie-Helman public values
dup	Flag in MQTT (value 1 indicates the packet is a retransmission)
HMAC	Key hash message authentication used in expansion of keys [51]
ik	Initial key variable set in the initial phase of QUIC's connection establishment
init	bit initializer, init $\in \{0,1\}$
iv	Initialization vector
k	Key to derive data in final_data() phase
Kg	Key generation algorithm takes λ as security parameter and generates (pk,sk) as key pair in QUIC
k_{stk}	Derived same as session key (for simplicity, in our implementation we treat k_{stk} as random string to replicate unpredictable input)
M	Message to be sent or received by client or server in QUIC
m	The message part of Msg
msgid	Message ID for MQTT queuing
Msg	Message space used in QUIC (Msg consists of bitstrings starting with a 1, while key exchange messages to be encrypted start with a 0)
nonc	Nonce
pub	Public DH Values
pk	Public key
REJ	Reject message (message from server after CHLO in QUIC)
retained	Flag used in MQTT to prevent loosing subscribed topics when a connection loss happens
RTT	Round-trip time
S	Server
scfg	Variable (output by scfg_gen()) to represent the global state of server in QUIC
scid	Server's cid
SHLO	Server Hello packet (s_hello) in QUIC
sk	Secret key
sqn	Sequence number (used in signalling for every segment in QUIC)
stk	Source address token to guard IP-spoofing in QUIC
$strike_{rng}$	Strike variable used by QUIC

C. Server-agent

Algorithm 1 shows the implementation of these APIs and functions. Our server-agent implementation is event-based. When an event $packet\_process$ occurs, the server-agent determines whether to transmit the packet or insert it into the receiving queue. Server-agent APIs and functions are described as follows:

1) main(): If the packet is subject to transmission, then action_flag is set, otherwise the packet will be processed by process_rx_packets(). This function also handles the disconnect events. If the server receives a disconnect event, then particular clients will be disconnected. However, the server might receive a disconnect event for all the clients. This happens, for example, when a reboot event occurs.

2) process_rx_packets(): It processes the packets in the receive queue by quic_input_message().

3) quic_input_message(): This API first checks whether the handshake process with the client has been completed or not. In order to determine this, it checks the $handshake\_completed$ flag. If it has been set, then it assumes that the connection is alive and handshake has been completed. In this case, the API skips the handshake process and enters the do_handshake_once() API to set the flag for encryption or decryption. If the $handshake\_completed$ flag is not set, then the API runs the handshake process by executing do_handshake(), where the client and server either follow the 1-RTT or 0-RTT implementation. This API also sets the client's state according to its operation. For instance, if the client is contacting the server for the first time, then the client's state is set to client_initial. If the handshake fails, then the client's state is set to client_handshake_failed. If the handshake has to be skipped, then it sets the client's state to client_post_handshake.

Algorithm 1: Server-agent APIs and functions

```
function main()
    /* Initialize tx and rx queues */
    initialize_rx_tx_msg_queue();
    is_client = false;
    while true do
        if event = process_packet then
            if packet must be transmitted then
                action_flag = send;
            else
                int sock =recvfrom();
                client_info.sock = sock;
                process_rx_packets();
        else if event = client_disconnect || socket_timeout then
            disconnect socket;
            break;
function process_rx_packets()
    while rx_message_queue do
        quic_input_message();
    return;
function quic_input_message()
    if !(handshake_completed) then
        if !(do_handshake()) then
            client_state = client_handshake_failed;
            return error;
        else
            client_state = client_initial;
    else
        client_state = client_post_handshake ;
        do_handshake_once();
    return;
```

Algorithm 2: Client-agent APIs and functions

```
function main()
    initialize_tx_rx_msg_queue();
    is_client = true;
    while true do
        if event = process_packet then
            if packet must be transmitted then
                int sock =create_udp_socket();
                action_flag = send;
                mqtt_message_initializer();
            else
                insert(rx_msg_queue);
                process rx_msg_queue for received packets;
        else if event = client_disconnect || socket_timeout then
            disconnect socket;
            break;
function mqtt_message_initializer()
    if (!MQTTClient_create()) then
        return error;
    if (!MQTTClient_connectOptions_initializer()) then
        return error;
    if (!mqtt_client_connect()) then
        return error;
    return;
function mqtt_client_connect()
    if !(sanity(this → msg)) then
        return error;
    initialize_quic();
    return;
function initialize_quic()
    /* filling quic header */
    if !(client → cid) then
        client → cid = get_cid(); /*Assign cid to client */
    populate_quic_header();
    start_connect();
    return;
function start_connect()
    quic_stream_gen();
    if !(handshake_completed) then
        do_handshake();
    else
        do_handshake_once();
    return;
function quic_stream_gen()
    /* Check stream presence */
    if stream_present then
        if if_stream_closed then
            clear_entry();
            return error;
        else
            /* Find stream */
            stream =find_stream();
    else
        /* Create a stream and associate it with the cid*/
        stream → cid = cid;
        stream =create_stream_buf();
    return;
```

D. Client-agent

This section deals with the APIs and functions specifically used by the client-agent. Algorithm 2 shows the client agent implementation.

1) `main()`: Similar to the server-agent, the `main()` function initializes both the transmit and receive queues by calling `initialize_rx_tx_queue()`. The client-agent is also event-driven and processes both incoming and outgoing packets based on the *packet_process* event. The `main()` function checks $action\_flag$ to determine if the packet is meant to be sent or processed by `process_rx_packets()`.

2) `mqtt_message_initializer()`: This API creates an MQTT client instance via `MQTTClient_create()` and sets the client ID, the persistence parameter (`retained`), and the server IP address. When a subscriber connects to a broker, it creates subscriptions for all the topics it is interested in. When a reboot or reconnect event occurs, the client needs to subscribe again. This is perfectly normal if the client has not created any persistent sessions. The persistence parameter creates a persistent session if the client intends to regain all the subscribed topics after a reconnection or reboot event.

Immediately after the creation of a client instance, a message is initialized by `MQTTClient_connectOptions_initializer()`. This API sets the `msgid`, `dup`, `retained`, `payload` and `version` fields of this packet. `MQTTClient_connectOptions_initializer()` creates a directory to store all the topics and includes this in $client\_info$ to be easily retrieved later.

3) `mqtt_client_connect()`: At this point, all the message construction functionalities related to MQTT are completed, and finally the client instance is ready for QUIC related operations. Sanity checking (e.g., header size, data length, etc.) is performed before entering the main QUIC API (i.e., `initialize_quic()`).

Algorithm 3: Common APIs and functions

```
function do_handshake()
    if !(protocol_connect) then
        return error;
    handshake_completed = true;
    if !(do_handshake_once()) then
        return error;
    return;
function do_handshake_once()
    if action_flag = send then
        flag = encrypt ;
    else
        flag = decrypt ;
    if !(crypt_quic_message()) then
        return error;
    return;
function crypt_quic_message()
    if flag = decrypt then
        this → msg = decrypt_message(this → msg);
        quic_dispatcher();
    else if flag = encrypt then
        this → msg = encrypt_message(this → msg);
        insert(this → msg, tx_msg_queue);
    return;
function protocol_connect()
    if session files not found then
        connect(); /* 1-RTT Implementation */
    else
        resume(); /* 0-RTT Implementation */
    return;
function quic_dispatcher()
    if is_client then
        /* If the client is calling the API */
        process rx_msg_queue for mqtt application;
        return;
    else
        /* MQTT parsing for received messages */
        if (valid_mqtt_header(this → msg)) then
            mqtt_parse_args (this → msg);
            return;
        else
            return error;
    return;
```

Algorithm 3: Common APIs and functions (*continued*)

```
function encrypt_message()
    if !(handshake_completed) then
        /* Initial session key is used */
        get_iv(H, ik)→ iv;
        if iv was used then
            return ⊥;
        else
            /* For client */
            if is_client then
                return (H, E(ik_c, iv, H, m);
            /* For server */
            else
                return (H, E(ik_s, iv, H, m);
    else
        /* The stored established key is used */
        get_iv(H, k) → iv;
        if iv is used then
            return ⊥;
        /* For client */
        if is_client then
            return (H, E(k_c, iv, H, m);
        /* For server */
        else
            return (H, E(k_s, iv, H, m);
function decrypt_message()
    /* Extracting ciphertext*/
    m → ci;
    if !(handshake_completed) then
        /* initial data phase key is used */
        get_iv(H, ik)→ iv;
        /* For client */
        if is_client then
            if D(ik_c, iv, H, ci) ≠ ⊥ then
                return plain_text;
            else
                return ⊥;
        else
            /* For Server */
            if D(ik_s, iv, H, ci) ≠ ⊥ then
                return plain_text;
    else
        /* Final data phase key is used */
        get_iv(H, k)→ iv;
        /* For client */
        if is_client then
            if D(k_c, iv, H, ci) ≠ ⊥ then
                return plain_text;
            else
                return ⊥;
        else
            /* For server */
            if D(k_s, iv, H, ci) ≠ ⊥ then
                return plain_text;
```

4) initialize_quic(): This API first determines if the client has an assigned `cid`, and if not, then a new `cid` is generated by $\{0,1\}^{64} \xrightarrow{\$}$ `cid`. The QUIC header is initialized after `cid` assignment. The non-encrypted part of the QUIC header consists of `cid`, diversification nonce, packet number or sequence number, pointer to payload, size of the payload, size of the total packet, QUIC version, flags and encryption level. The encrypted part of the QUIC header is made of frames, where each frame has a stream ID and a final offset of a stream. The final offset is calculated as a number of octets in a transmitted stream. Generally, it is the offset to the end of the data, marked as the `FIN` flag and carried in a stream frame. However, for the reset stream, it is carried in a `RST_STREAM` frame [37]. Function `populate_quic_header()` fills both the encrypted and non-encrypted parts of the QUIC header and enters the API `start_connect()` to connect the client.

5) start_connect(): This API is the entering point to the common APIs (described in Section III-E). Its first task is to find an appropriate stream for the connection. Based on the `handshake_completed` flag, `start_connect()` API determines if the client and server have completed the handshake. If the handshake has been completed, then it means that QUIC connection is alive and data can be encrypted or decrypted. If not, then it enters the handshake process by executing the `do_handshake()` API.

6) quic_stream_gen(): The only purpose of this API is to detect and create streams. As mentioned earlier, QUIC is capable of multiplexing several streams into one socket. This API first detects whether there is already an open stream for

Algorithm 4: 1-RTT Implementation

```
function connect()
    if initial_key_exchange() then
        if initial_data() then
            if key_settlement() then
                if final_data() then
                    return success;
    return error;
function initial_key_exchange()
    message m;
    /* For client */
    if is_client then
        switch phase do
            case initial do
                /* pk is public key */
                c_i_hello(pk)→ m1;
                break;
            case received_reject do
                /* m2 received packet from server (REJ) */
                c_hello(m2)→ m3;
                break;
            case complete_connection_establishment do
                /* ik is initial key variable set during initial phase */
                get_i_key_c(m3)→ ik;
                break;
    /* For server */
    else
        switch phase do
            case received_chlo do
                s_reject(m1) → m2;
                break;
            case received_complete_chlo do
                get_i_key_s(m3) → ik;
                break;
    return;
function initial_data()
    /* For client */
    if is_client then
        for each α ∈ [i];
        α + 2 →sqn_c;
        /* sqn_c = Client sequence number */
        /* M_c^α = Client constructed message */
        pak(ik, sqn_c, M_c^α) → m_4^α;
        process_packets(ik, m5);
    /* For server */
    else
        for each β ∈ [j];
        β + 1 → sqn_s;
        /* sqn_s = Server sequence number*/
        /* M_s^α = Server constructed message */
        pak(ik, sqn_s, M_s^β) → m;
        process_packets(ik, m4);
    return;
```

Algorithm 4: 1-RTT Implementation (*continued*)

```
function key_settlement()
    if is_client then
        get_key_c(m6, sqn_s) → k;
    else
        2 + j → sqn_s;
        s_hello(m, ik, sqn_s) → m6;
        get_key_s(m6) → k;
    return;
function final_data()
    if is_client then
        for each α ∈ {i + 1,...,u}
        α + 2 →sqn_c;
        pak(k, sqn_c, M_c^α)→ m_7^α;
        (m_7^{i+1}.....m^u) → m7;
        process_packets(k, m7);
    else
        for each β ∈ {j + 1,...,w}
        β + 2 →sqn_s;
        pak(k, sqn_s, M_s^β) → m_8^β;
        (m_8^{j+1}.....m^w) → m8;
        process_packets(k, m8);
    return;
```

a particular client, and if not, then it creates a new stream by running `create_stream_buf()`.

E. Common APIs and functions

Several APIs and functions such as encryption, decryption and processing transmission packets are mandated on both the server and client sides. Although the packets handled by these functions are different in the client-agent and server-agent, the underlying mechanisms are almost similar. Algorithm 3 shows the implementation.

1) `do_handshake()`: This API is the starting point of the QUIC connection establishment and key exchange process. It behaves as an abstraction of the handshake process. First, it calls `protocol_connect()` for protocol communication. If `protocol_connect()` is executed successfully, then the *handshake_completed* flag is set. Based on this flag, the client determines whether to execute the handshake process or skip it. Lastly, this API calls the `do_handshake_once()` API to set the encrypt or decrypt flag for further processing.

2) `do_handshake_once()`: The primary responsibility of this API is to set the encryption or decryption flags. This decision is made based on the type of the next operation, which is transmission or processing. If *action_flag* is set, then the packet must be encrypted and sent. Finally, it enters the `crypt_quic_message()` API.

3) `crypt_quic_message()`: This API is the starting point in establishing a secure connection. It checks the $flag$ to determine if the packet should undergo decryption (`decrypt_message()`) or encryption (`encrypt_message()`).

4) `protocol_connect()`: In this API, the client (C) checks the existence of a session file to determine whether it has communicated with the server (S) during the last τ_t seconds. If C and S are interacting for the first time, then the `connect()` API completes the 1-RTT scenario in four phases, as shown in Figure 5 and Algorithm 4. If C and S have communicated before, then the `resume()` API follows the 0-RTT scenario shown in Figure 6 and Algorithm 7.

1-RTT Connection. The 1-RTT implementation is divided into four phases. The first phase exchanges the initial keys to encrypt the handshake packets until the final key is set. The second phase starts exchanging encrypted initial data. The third phase sets the final key. Last, the fourth phase starts exchanging the final data. We explain the details of these phases as follows.

[1-RTT]: Phase 1. This phase is handled by

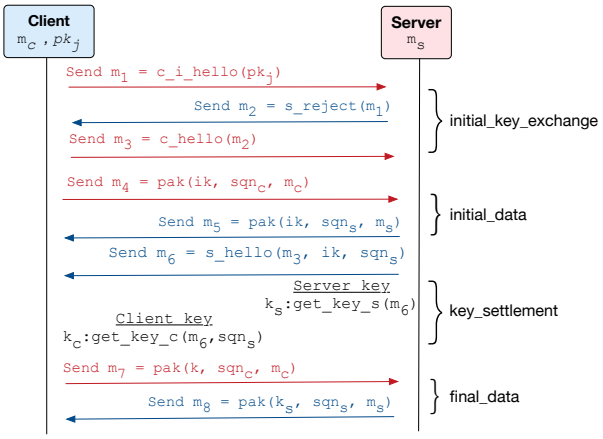

Fig. 5. The four phases of the 1-RTT connection.

Algorithm 5: Server Configuration State

function `get_scfg`(sk, τ_t, λ)
- $\mathbb{Z}_{a-1} \xrightarrow{\$} x_s$;
- $b^{x_s} \rightarrow y_s$;
- $(b, a, y_s) \rightarrow \texttt{pub}_s$;
- $x_s \rightarrow \texttt{sec}_s$;
- $\tau_{t+1} \rightarrow \texttt{expy}$;
- $\mathcal{H}(\texttt{pub}_s, \texttt{expy}) \rightarrow \texttt{scid}$; /* $\mathcal{H}$ is a SHA-256 function */
- *"QUIC Server Config Signature"* $\rightarrow$ `str`;
- `Sign`$(s_k$, (str, 0x00, scid, $\texttt{pub}_s$, expy)$) \rightarrow \texttt{prof}$;
- $(\texttt{scid}, \texttt{pub}_s, \texttt{expy}) \rightarrow \texttt{scfg}^t_{\texttt{pub}}$;
- return `scfg`;

`initial_key_exchange()` and consists of three messages, m_1, m_2 and m_3. The client C runs `c_i_hello`(pk), which returns a packet m_1 with sequence number 1. m_1 is an initial connection packet sent to S, containing a randomly generated `cid`. In response to m_1, S sends a `REJ` packet m_2, generated using `s_reject`(m_1). m_2 contains a source-address token stk (similar to TLS session tickets [53]), which is later used by C to prove its identity to S for the ongoing session and future sessions. This is performed by checking if the source IP address equals the IP address in stk. Fundamentally, the stk consists of an encryption block of C's IP address and a timestamp. In order to generate stk, S uses the same $\mathcal{E}$ deterministic algorithm (i.e., AEAD) with k_{stk} (derived by $\{0,1\}^{128} \xrightarrow{\$} k_{stk}$). The initialization vector for stk (i.e., $\texttt{iv}_{stk}$) is selected randomly and is used in `s_reject`. For simplicity, we implemented a validity range for stk, which is bounded by the time period during which it was either generated or set up. Another important parameter in m_2 is the S's current state `scfg_pub` (refer to Algorithm 5). It contains S's DH values with an expiration date and a signature `prof`. This signature is signed by `SS` over all the public values under the S's secret key sk.

Upon receiving m_2, the client C checks $\texttt{scfg}^t_{pub}$ for its authenticity and expiration. The algorithms for this purpose can be found in [52]. As we mentioned in Section III-B, our implementation assumes that a PKI is in place. After possessing the public key of S, the client generates a `nonc` and DH values by running `c_hello`(m_2), and sends them to the server in message m_3. At this point, both C and S derive the initial key material ik, using `get_i_key_c`(m_3) and `get_i_key_s`(m_3), respectively. The server keeps track of the used `nonc` values in order to make sure that it does not process the same connection twice. This mechanism is referred as *strike-register* or `strike`. The timestamp is included in the `nonc` by the client. The server only maintains the state of a connection for a limited duration of time. Any connection request from a client is rejected by the server if its `nonc` is already included in its `strike` or contains a timestamp that is outside the permitted range $\texttt{strike}_{rng}$. The initial key ik = (ik_c, ik_s, `iv`) is made of two parts: two 128-bit application keys (ik_c, ik_s) and two 4-byte initialization vector prefixes `iv` = ($\texttt{iv}_c$, $\texttt{iv}_s$). The client uses ik_s and $\texttt{iv}_s$ to encrypt the data and send it to S. On the other hand ik_c and $\texttt{iv}_c$ assist in decryption and encryption. This phase happens only once during the time period τ_t until $\texttt{scfg}^t_{pub}$ and stk are not expired.

[1-RTT]: Phase 2. This phase is handled by `initial_data()` and consists of two messages, m_4 and m_5. The client C and server S exchange the initial data message M_c and M_s, which are encrypted and authenticated with ik in function `pak`$(ik, \texttt{sqn}_c, M^i_\alpha)$ for every $\alpha \in [i]$ and `pak`$(ik, \texttt{sqn}_s, M^i_\beta)$ for every $\beta \in [j]$, respectively. Here, $\texttt{sqn}_c$ and $\texttt{sqn}_s$ represent the sequence number of packets sent by C and S, respectively. i and j represent the maximum number of message blocks that C and S can exchange prior to the first phase. The initialization vector `iv` is generated based on the server or client role. When S sends a packet, `get_iv()` outputs `iv` by concatenating $\texttt{iv}_c$ and $\texttt{sqn}_s$. When C sends a packet, `get_iv()` generates `iv` by concatenating $\texttt{iv}_s$ and $\texttt{sqn}_c$. The total length of each `iv` is 12 bytes since both the server and client initialization vector prefixes (i.e., $\texttt{iv}_c$ and $\texttt{iv}_s$) are 4 bytes in length and sequence numbers (i.e., $\texttt{sqn}_s$ and $\texttt{sqn}_c$) are 8 bytes in length. When C receives packets from S, it uses the `process_packets()` function to decrypt those packets to extract their payloads and concatenates them based on their sequence number. The server S performs a similar mechanism for the packets received from C.

[1-RTT]: Phase 3. This phase is handled by `key_settlement()` and involves message m_6. The server S produces new DH values (authenticated and encrypted via AEAD with ik) and transmits them to the client using `s_hello`$(m_3, ik,$ `sqn`$)$. The client verifies the server's new DH public values with the help of ik. At this point, the server and client both derive the session key by `get_key_s`(m_6) and `get_key_c`(m_6) and use `extract_expand()` for key expansion, as defined in Algorithm 6.

[1-RTT]: Phase 4. This phase is handled by `final_data()` and consists of two messages, m_7 and m_8. Instead of initial key ik, the established key k is used to encrypt and authenticate the remaining data (`Msg`) by both C and S. Similar to ik, k is derived from k_c, k_s, and `iv`, and consists of two parts: the two 128-bit application keys (k_c, k_s) and the two 4-bytes initialization vector prefixes `iv` = ($\texttt{iv}_c$,$\texttt{iv}_s$). In order to encrypt the data, C uses k_s and $\texttt{iv}_s$ before sending to S. For decryption, C uses $\texttt{iv}_c$ and $\texttt{k}_c$ to decrypt the data received

Algorithm 6: QUIC messages exchange APIs and functions

function c_i_hello(pk)
  $\{0,1\}^{64} \xrightarrow{\$}$ cid;
  return {IP_c, IP_s, port_c, port_s, cid, 1};
function s_reject(m)
  $\{0,1\}^{96} \xrightarrow{\$} \text{iv}_{stk}$;
  (iv_{stk}, $\mathcal{E}(k_{stk}, \text{iv}_{stk}, \epsilon, 0) \parallel (\text{IP}_c, \text{current\_time}_s)) \rightarrow$ stk;
  return {IP_s,IP_c,port_s,port_c,cid,1,scfg^t_{pub}, prof, stk};
  /* prof is generated by get_scfg */
function c_hello(m)
  if $expy \leq \tau_t$ **then**
    return;
  $str =$ "$QUIC\ server\ config\ signature$";
  if $Ver(pk, (str, 0X00, scid, pub_s, expy), prof)) \neq 1$; **then**
    return;
  $\{0,1\}^{160} \xrightarrow{\$} r$;
  $(current\_time_c, r) \rightarrow$ nonc;
  $\mathbb{Z}_{a-1} \xrightarrow{\$} x_c,\ b^{x_c} \rightarrow y_c,\ (b, a, y_c) \rightarrow pub_c$;
  (IP_c, IP_s, port_c, port_s) $\rightarrow$ pkt_info;
  return (pkt_info, cid, 2, stk, scid, nonc, pub_c);
function get_i_key_c(m)
  $y_s^{x_c} \xrightarrow{\$}$ipm;
  return extract_expand(ipm, nonc, cid, m, 40, 1)
function get_i_key_s(m)
  stk$\rightarrow$(iv_{stk}, tk);
  $\text{D}(k_{stk}, \text{iv}_{stk}, \epsilon, tk) \rightarrow$ d;
  if ($d = \perp \parallel$ ($first\ 4\ bytes\ of\ d \neq 0$) **then**
    if ($first\ 4\ bytes\ of\ d$) $\neq$ IP_c) **then**
      return;
    return;
  if *last 4 bytes corresponds to outside* $strike_{rng}$ **then**
    return;
  if $r \in strike \parallel \tau_t \notin strike_{rng}$ **then**
    return;
  if *scid is unknown* **then**
    return;
  if *scid corresponds to expired* $scfg^{t`}_{pub}$ **then**
    /* where $t` < t$ */
    return;
  if $b, a \in pub_c \neq b, a \in pub_s$ **then**
    return;
  $y_c^{x_s} \rightarrow$ipm;
  return extract_expand(ipm,nonc,cid,m,40,1);
function extract_expand(*ipm, nonc, cid, m, l, init*)
  HMAC(nonc,ipm)$\rightarrow$ ms;
  if $init = 1$ **then**
    "$QUIC\ key\ expansion$" $\rightarrow str$;
  else
    "$QUIC\ forward\ secure\ key\ expansion$" $\rightarrow str$;
  (str, $0X00$, cid, m, scfg^t_{pub}) $\rightarrow$info;
  return first l bytes (octets) of T = (T(1), T(2),...),
  if all $i \in \mathbb{N}$, T(i) = HMAC(ms, (T(i-1), info, 0x0i)) and
  T(0) = ϵ

Algorithm 6: QUIC messages exchange APIs and functions (*continued*)

function get_iv(H, κ)
  /* $\kappa_c =$ Client key,
  * $\kappa_s =$ Server key
  * $iv_c =$ Client initialization vector,
  * $iv_s =$ Server initialization vector */
  $\kappa \rightarrow (\kappa_c,\ \kappa_s,\ iv_c,\ iv_s)$;
  if *is_client* **then**
    $c \rightarrow src, s \rightarrow dst$;
  else
    $s \rightarrow src$, $c \rightarrow dst$;
  /* sqn is packet sequence number */
  $H \rightarrow$(cid, sqn);
  return (iv_{dst}, sqn);
function pak(k, *sqn*, m)
  $\kappa \rightarrow (k_c, k_s, \text{iv}_c, \text{iv}_s)$;
  if *is_client* **then**
    $c \rightarrow src$ and $s \rightarrow dst$;
  else
    $s \rightarrow src$ and $c \rightarrow dst$;
  (IP_{src}, IP_{dst}, port_{src}, port_{dst}) $\rightarrow$ pkt_info;
  (cid, sqn)$\rightarrow H$;
  get_iv(H, κ) $\rightarrow$ iv;
  return (pkt_info, $\mathcal{E}(k_{dst}$,iv,H, $(1 \parallel m)$));
function process_packets($\kappa, p_1, p_2....p_v$)
  $\kappa \rightarrow (k_c, k_s, \text{iv}_c, \text{iv}_s)$;
  if *is_client* **then**
    $c \rightarrow src$ and $s \rightarrow dst$;
  else
    $s \rightarrow src$ and $c \rightarrow dst$;
  for each $\gamma \in [v]$:
    $p_\gamma \rightarrow (H_\gamma, c_\gamma)$;
    get_iv(H_γ, κ) $\rightarrow \text{iv}_\gamma$;
    $\text{D}(k_{src}, iv_\gamma, H_\gamma, c_\gamma) \rightarrow m\gamma$;
    if $m_\gamma \notin Msg$ **then**
      return
  return (m_1, m_2, ... m_v);
function s_hello(m_3, *ik, sqn*)
  $ik \rightarrow (ik_c, ik_s, \text{iv}_c, \text{iv}_s)$;
  $\mathbb{Z}_{a-1} \xrightarrow{\$} \widetilde{x_s},\ b^{\widetilde{x}_s} \rightarrow \widetilde{y}_s,\ (b, a, \widetilde{y}_s) \rightarrow \widetilde{\text{pub}}_s$;
  (cid, sqn)$\rightarrow H$;
  $\mathcal{E}(ik_c, (\text{iv}_c, sqn), H, (0 \parallel (\widetilde{\text{pub}}_s, \text{stk}))) \rightarrow e$;
  return (ip_s,ip_c,port_s,port_c, H, e);
function get_key_s(m)
  $y_c^{\widetilde{x}_s} \rightarrow$pms;
  return extract_expand(pms, nonc, cid, m, 40, 0);
function get_key_c(m)
  $m \rightarrow$(IP_s,IP_c,port_s, port_c, cid, sqn, e);
  if $D(ik_c, (iv_c, sqn), (cid, sqn), e) = \perp$ **then**
    return
  if *first bit of the message* $\neq 0$ **then**
    return
  $\widetilde{y}_s^{x_c} \rightarrow$ pms;
  return extract_expand(pms, nonc, cid, m, 40, 0);

from S.

0-RTT Connection. Another scenario of connection establishment is 0-RTT, as Figure 6 shows. If C has already established a connection with S in the past τ_t seconds, then C skips sending c_i_hello(), and initiates another connection request to the server by sending a c_hello() packet. This packet contains the existing values of stk, scid, cid, nonc, and pub_c. It is important to note that pub_c requires new DH ephemeral public values. After receiving c_hello, S verifies if the nonc is fresh. This is performed against strike-register, provided that stk is valid and scid is not unknown or expired. If these verification steps do not succeed, then S returns to the 1-RTT process by generating and sending out a s_reject message (Algorithm 6). If these verification steps succeed, then the rest of the protocol remains the same. Algorithm 7 shows the implementation for 0-RTT.

5) quic_dispatcher(): This API is the starting point where MQTT-related parsing starts. After the QUIC header is

Algorithm 7: 0-RTT Implementation

```
1  function resume()
2      if (pub_c) then
3          if stk then
4              if scid then
5                  return c_hello(stk, scfg^t_pub);
6      /* Jumping back on 1-RTT */
7      return connect();
8  function c_hello(stk, scfg^t_pub)
9      {0,1}^64 -$-> cid;
10     {0,1}^160 -$-> r, (current_time_c, r) -> nonc;
11     Z_{a-1} -$-> x_c, b^{x_c} -> y_c, (b, a, y_c) -> pub_c;
12     (IP_c, IP_s, port_c, port_s) -> pkt_info;
13     return (pkt_info, cid, 1, stk, scid, nonc, pub_c)
```

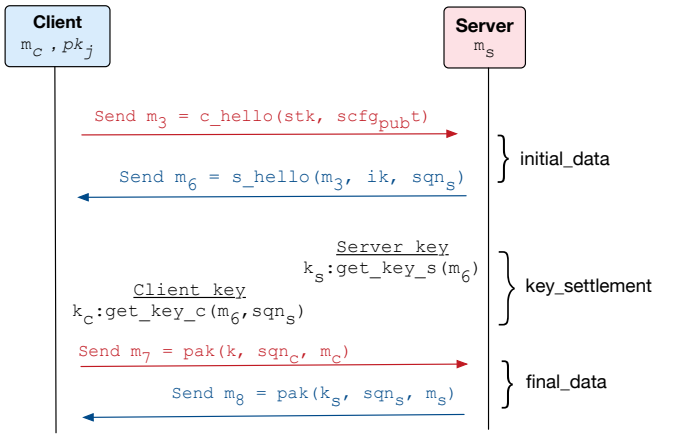

Fig. 6. In 0-RTT scenario, client sends the first packet `c_hello` with previous server global state scfg and strike. This step itself sends initial data to server.

stripped off and decrypted, the packet is delivered to MQTT. Here, the broker finds the topic and delivers the message to all the subscribers that have subscribed to that topic.

F. Code Reduction

QUIC is a part of the Chromium Projects [33]. These projects have a heavy code size because they include several features such as the Chrome browser, SPDY protocol, Chromecast, Native Client, and QUIC with their entwined implementations. In order to reduce the code size, we have removed all these features except those that are essential to the functionality of QUIC. Furthermore, to reduce its overhead, most of the logging features have been removed or disabled. We have also eliminated *alarm_factory*, which generates platform-specific alarms. The *host resolver* has been removed as well. As most clients have an in-built DNS prefetching [54] feature, they can resolve DNS queries without having to communicate with a server. Since MQTT routes data flows based on topics instead of URLs, this feature is not required. We have also removed the backend proxy-related code. Backend proxy is used when the proxy server is behind the firewall and load balances the requests from clients to the servers. Since MQTT is based on the publish/subscribe model, it is futile to involve any proxy or load balancing. Finally, we removed all the unit-testing code.

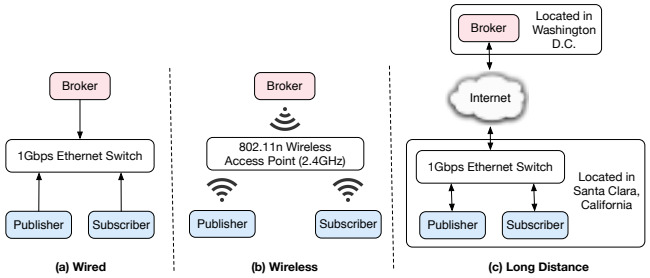

Fig. 7. The three testbeds used for the performance evaluations.

IV. Evaluation

This section presents the performance evaluation of MQTTw/QUIC versus MQTTw/TCP.

In order to measure the impact of the physical layer and distance on performance, three different testbeds are used: *wired*, *wireless*, and *long-distance*.

- **Wired.** All the nodes are connected to a Netgear 1Gbps L2-learning switch.
- **Wireless.** The nodes communicate through an 802.11n network. The access point used is a Linksys AC1200 and operates on channel 6 of the 2.4GHz band. In addition, we placed two other similar access points 3 meters away form the main access point. In order to introduce latency and packet drops, these two access points operate on the same channel and therefore interfere with the main access point. Each interfering access point continuously exchanges a 10Mbps UDP flow with a nearby user.
- **Long Distance.** Our objective in using the long-distance routed network is to introduce longer and unpredictable end-to-end delays. The subscriber and publisher are placed in Santa Clara, California, and the broker is situated across the country in Washington D.C.

Raspberry Pi 3 model B is the device used as the subscriber, publisher, and broker. These devices run Raspbian Stretch as their operating system. As mentioned in the previous section, the MQTT implementation is based on Eclipse Paho and Mosquitto. In addition, the TLS version is 1.2, the socket timeout for TCP and QUIC is 30 seconds, and the keep-alive mechanism of MQTT has been disabled. Figure 7 shows these testbeds.

When presenting the results, each point is the median of values obtained in 10 experiments, where each experiment includes 10 iterations. For example, when measuring the overhead of connection re-establishment, the subscriber and publisher are connected to the broker 10 times, and the result is counted as one experiment. The iterations of an experiment are run consecutively, and the minimum time interval between the experiments is 5 minutes.

The rest of this section studies the performance of MQTTw/QUIC versus MQTTw/TCP in terms of the overhead of connection establishment, head-of-line blocking, half-open connections, resource utilization, and connection migration.

A. Overhead of Connection Establishment

This section evaluates the number of packets exchanged between devices when using MQTTw/TCP and MQTTw/QUIC

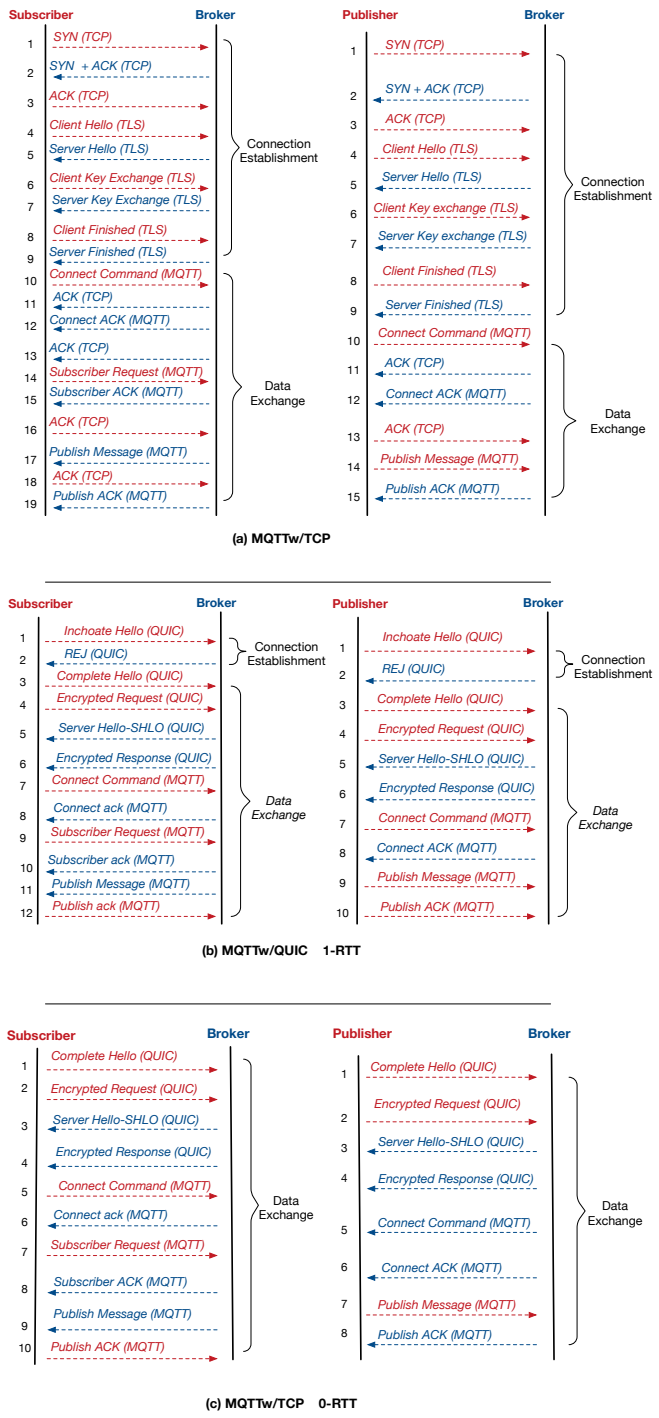

Fig. 8. The connection establishment process using (a) MQTTw/TCP, (b) MQTTw/QUIC 1-RTT, and (c) MQTTw/QIUC 0-RTT.

during the connection establishment process. Before presenting the results, we first study the sequence of packet exchanges between a client and a server. Figures 8(a), (b), and (c) show this sequence in MQTTw/TCP and MQTTw/QUIC's 1-RTT, and 0-RTT, respectively. Please note that these figures demonstrate ideal scenarios when there is no packet drop. In addition, MQTT ping packets are excluded to simplify the evaluations. Figure 9 and Table II summarize the results from

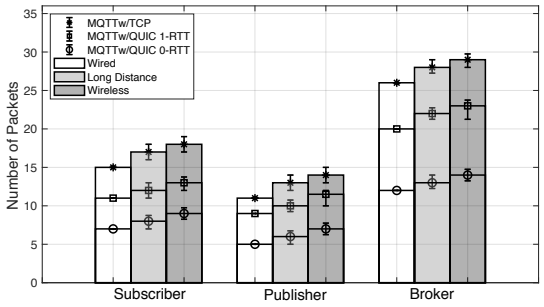

Fig. 9. The number of packets exchanged during the connection establishment phase in the wired, wireless, and long-distance testbeds. The error bars show the lower quartile and higher quartile of the collected results.

the subscriber, publisher and broker point of views using the three testbed types mentioned earlier.

As the results show, MQTTw/QUIC reduces the number of packets exchanged with the broker by up to 56.25%. In addition, the wireless testbed shows the highest performance improvement compared to the other two testbeds, which is due to its higher packet loss rate. Therefore, since the number of packets required by MQTTw/QUIC is lower in this scenario, compared to MQTTw/TCP, the total number of packet retransmissions is lower. Compared to the wired testbed, the long-distance testbed shows a higher performance improvement. In the wired network, all the three devices are in same broadcast domain, therefore, no layer-3 routing is performed. This an ideal situation since all the devices are connected directly through a LAN and the probability of packet loss is almost zero. However, packets might be lost or significantly delayed in the long-distance testbed due to events such as congestion and multipath. As IoT applications such as smart homes usually include lossy links, these results indicate the significant potential benefits of using QUIC in these scenarios.

B. Head-of-Line Blocking

The head-of-line blocking problem occurs when the receiver is waiting for the dropped packets to be re-transmitted in order to complete the packet reordering before delivering them to the application. To simulate this problem in our testbeds, a FreeBSD [55] router was introduced in between the publisher and broker.

FreeBSD enables us to intercept packets based on the firewall rules and their network buffers. During this experiment, which ran for 300 seconds, data packets are intercepted based on their flow ID. The flow ID is a unique tuple of source IP address and source port to identify the connection. The FreeBSD's firewall function `ipfw_chk` is used to intercept the packets and drop them. In order to replicate a real-world scenario, data packets were dropped randomly at fixed intervals. For instance, to simulate a 10% drop scenario, every 10th packet belonging to the same ID is dropped. In these experiments, latency is computed as the interval between the instance a packet leaves the publisher until the reception of that packet by the subscriber. To accurately measure latency without introducing extra traffic, WiringPi [56] has been integrated into our testbed. For each received message, the

TABLE II
PERFORMANCE IMPROVEMENT OF MQTTW/QUIC VERSUS MQTTW/TCP IN TERMS OF THE NUMBER OF PACKETS EXCHANGED DURING THE CONNECTION ESTABLISHMENT PHASE.

Testbed		**Wired**			**Wireless**			**Long Distance**		
Device		Subscriber	Publisher	Broker	Subscriber	Publisher	Broker	Subscriber	Publisher	Broker
Improvement vs MQTTw/TCP	1-RTT	36.84%	33.33%	35.29%	48.18%	42.85%	47.91%	45.83%	35.29%	40%
	0-RTT	47.36%	46.66%	47.05%	55.55%	52.38%	56.25%	54.16%	47.05%	50%

receiver uses the WiringPi library to notify the sender by generating a signal to toggle a pin connected to the sender.

Table III summarizes the results for all the three testbeds. Dropping random packets intermittently forces the sender to retransmit the lost packets. As the results show, TCP's latency is higher than QUIC's in all the scenarios. This is due to the packet re-ordering delay in TCP, which happens when a line of packets is being held up by the receiver when prior packets are lost. In contrast, QUIC is based on UDP, which does not hold up the received packets while the lost packets are being re-transmitted. This enables the receiver's application layer to perform the decryption operation as soon as packets arrive.

In the wireless and long distance testbeds, apart from the intentional drops introduced, both MQTTw/QUIC and MQTTw/TCP show higher latencies as they suffer from the additional packet drops, compared to the wired testbed. Therefore, the improvement margins between MQTTw/TCP and MQTTw/QUIC in wireless and long distance testbeds are lower than the wired network. For example, for the 10% drop rate scenario, MQTTw/QUIC reported 1.3833 ms and 7.1099 ms latencies in the wired and wireless testbeds, respectively. Therefore, the highest improvement achieved is for wired networks.

C. TCP Half-Open Connections

An act of receiving data in TCP is passive, which means that a dropped connection can only be detected by the sender and not by the receiver. In order to simulate the TCP half-open connection problem, 100 connections were established by 10 publishers using different topics and client IDs. One message is transmitted per second over each connection. To generate TCP half-open connections, the keep-alive message transmission mechanism of MQTT has been disabled. The half-open connections are detected using Linux command `ss -a`. In order to reveal the overhead of half-open connections, memory usage and processor utilization level were measured using the Linux's `top` utility. To generate half-open connections, all the publishers are restarted in the middle of a message flow. In this scenario, the broker is unaware of the connection tear-down.

Figures 10 and 11 show the resource utilization of the broker. Both figures present the results of a single experiment to reveal the variations of memory and processor utilization versus time. The duration of this experiment is long enough to include the process of client connection, subscription, publishing, and restart of the clients. After each of the aforementioned operations, we wait for the processor and memory utilization to become stable and then start the next operation. These

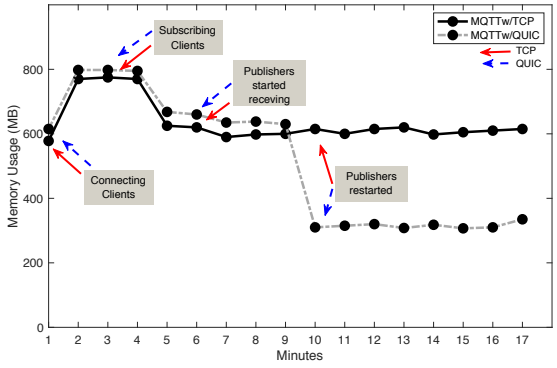

Fig. 10. The memory utilization of broker. While the memory utilization level of MQTTw/QUIC is dropped after the publishers were restarted, the memory utilization of MQTTw/TCP remains high.

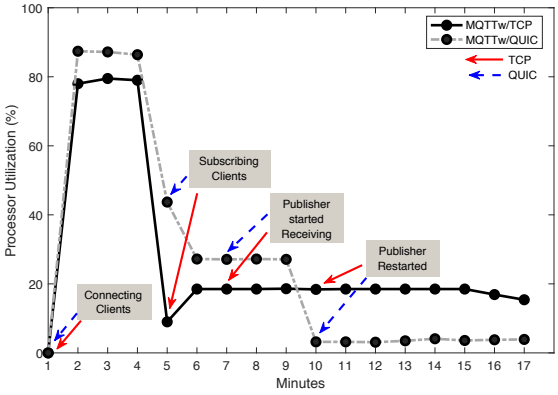

Fig. 11. The processor utilization of broker. Although the processor utilization of MQTTw/QUIC is more than MQTTw/TCP, after the publishers were restarted, MQTTw/QUIC shows a higher drop in processor usage as QUIC runs over UDP and does not require any connection state information.

results shows that MQTTw/QUIC releases the resources as soon as it detects the connection is broken. Specifically, the MQTTw/QUIC's UDP socket times out under 1 minute and QUIC starts its draining period. On the other hand, TCP connections still wait for incoming packets, as there is no mechanism to detect whether the connections opened by the publishers are still active or not. In summary, it is observed that MQTTw/QUIC reduces processor utilization between 74.67% to 83.24% and lowers memory utilization between 45.52% to 50.32%, compared to MQTTw/TCP. It must be noted that MQTTw/QUIC achieves these improvements without introducing any communication overheads such as keep-alive packets.

TABLE III
PERFORMANCE IMPROVEMENT OF MQTTW/QUIC VERSUS MQTTW/TCP IN TERMS OF PACKET DELIVERY LATENCY IN THE PRESENCE OF PACKET DROPS. THE DROP RATE IS CHANGED BETWEEN 10% TO 50%.

Testbed	**Wired**			**Wireless**			**Long Distance**		
Drop Rate	10%	20%	50%	10%	20%	50%	10%	20%	50%
Latency of MQTTw/TCP	3.114 ms	5.234 ms	10.6904 ms	11.2176 ms	15.7451 ms	21.4137 ms	44.3578 ms	54.9340 ms	75.6283 ms
Latency of MQTTw/QUIC	1.3833 ms	3.5680 ms	8.9389 ms	7.1099 ms	10.5313 ms	18.5313 ms	33.0088 ms	44.0462 ms	64.3635 ms
Improvement vs MQTTw/TCP	55.57%	31.83%	16.38%	36.61 %	33.11%	13.46%	25.58%	19.81%	14.89%

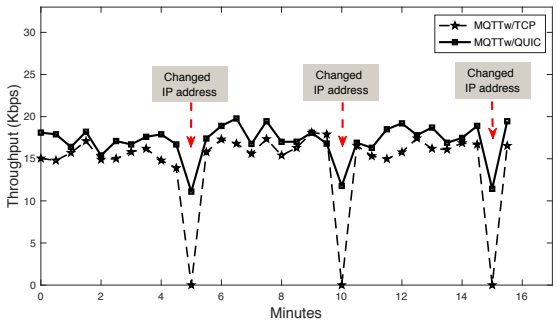

Fig. 12. The effect of connection migration on throughput. After each IP renewal, TCP re-establishes the connections. In contrast, QUIC connections are survived and only a slight throughput degradation can be observed due to the change in interface IP address.

D. Connection Migration

As mentioned earlier, connection reestablishment due to roaming imposes energy overhead. This section evaluates the efficiency of the proposed system during connection migrations. To this end, 100 connections were established from 10 publishers to a broker, and the subscribers were subscribed to all the topics. The `nload` utility was used to measure throughput. In order to emulate the change of network, the IP address of the broker's interface connected to the subscriber is changed every 5 minutes. The total duration of the experiment is 16 minutes to include three connection reestablishment triggers and allow the publishers and broker to stabilize after each migration.

Figure 12 shows the results. Whenever a change in the connection parameters occurs, MQTTw/TCP has to repeat the cumbersome process of connection re-establishment. In this case, its bandwidth drops to zero and picks up again after the connection re-establishment process completes. In contrast, MQTTw/QUIC's throughput shows a slight reduction due to the change in the interface state, but connections were migrated to the new IP unscathed. Specifically, QUIC allows the end point devices to survive in such events, requiring only to have a stable IP address during the handshake process to obtain the 1-RTT keys. This feature is particularly useful in mobile IoT scenarios such as medical and industrial applications [8]. Another useful case is to make the clients resilient against NAT rebinding.

V. RELATED WORK

This section is composed of two sub-sections. The first sub-section reviews the IoT application layer protocols and their employed transport layer protocols. The second sub-section reviews the literature relevant to QUIC and compares the features offered by the proposed MQTTw/QUIC implementation against existing IoT application layer protocols.

A. IoT Application Layer Protocols

This section overviews the main IoT application layer protocols and their adoption of transport layer protocols.

1) MQTT (Message Queuing Telemetry Transport): MQTT is a lightweight IoT application protocol for machine-to-machine connectivity. The protocol specification [27] mandates the use of a connection-oriented transport protocol to ensure packet ordering and reliable delivery. Therefore, most implementations [35], [57]–[60] use TCP/IP with TLS/SSL to satify this requirement. In some cases [61], the Stream Control Transmission Protocol (SCTP), an alternative to TCP, has been used as well. However, the key problems with SCTP when used in IoT domains are: (i) 4-way handshake, which increases latency compared to the TCP's 3-way handshake. (ii) SCTP is mainly used for multihoming and redundancy. Whereas in IoT, redundancy is not an essential aspect compared to latency. (iii) SCTP is not suitable for heterogeneous networks [62]. The authors in [63] studied TCP, UDP and SCTP, and justified why SCTP is not the best fit as an IoT protocol.

2) XMPP (Extensible Messaging and Presence Protocol): XMPP offers both publish/subscribe (asynchronous) and request/response (synchronous) messaging capabilities. This protocol uses TLS/SSL for security, and relies on TCP to ensure reliability. XMPP is most suited for real-time applications, with small message footprints and low latency in message exchange [64]. Although XMPP is preferred over CoAP in data-centric IoT scenarios due to offering the publish/subscribe mechanism, the lack of QoS provisioning might not be acceptable in mission-ciritical applications. Furthermore, its underlying transport protocol, TCP, inherits the shortcomings we discussed earlier. Google has reported various XMPP incompatibilities with their new applications [65] and declared it as an obsolete protocol.

3) REST (Representational State Transfer): REST is an architectural paradigm. REST uses the HTTP [66] methods (GET, POST, PUT and DELETE) to provide a resource-oriented messaging system. All actions are performed by simple request/response (synchronous) HTTP commands. It has a built-in accept HTTP header to determine the format of the data, which is usually JSON (JavaScript Object Notation) or XML (Extensible Markup Language). REST is easy to

implement and is supported by most M2M cloud platforms. Other features supported by REST are caching, authentication and content type negotiation [67]. This protocol, however, does not offer high energy efficiency since polling must be performed in order to check the availability of data.

REST provides a one-way connection between the client and server. Client only connects to the server when it has to either push or pull the data. On the other hand, MQTT relies on a two-way established connection between the server and client. This enables the server to respond to client's request instantly [68]. In contrast, since REST is a request/reply protocol, it does not need to establish a long-term connection. However, in lossy networks, in particular, establishing connections repeatedly increases energy consumption [69]. BevyWise Networks conducted experiments and estimated that their proprietary *MQTTRoute* broker [70] consumes 20% less power than REST. Their findings concluded that the primary reason behind more power consumption in REST was due to the resources used for connecting, re-connecting, and cleaning up both the server and client connection states. As another example, [68] shows that MQTT is up to 25 times faster than REST in terms of data transfer rate.

The availability of a REST server might be limited if the devices are behind a firewall. A common scenario of firewall deployment is when all new incoming connections are blocked, while outgoing connections are always allowed due to zoning. In this case, establishing connection to IoT devices would not be possible. MQTT solves this issue by relying on two-way connections.

4) AMQP (Advanced Message Queuing Protocol): Although AMQP is agnostic to transport protocol, it uses TCP (and TLS) by default. AMPQ offers a publish/subscribe model for messaging. This protocol provides a reliable connection by utilizing a *store and forward* mechanism [71], [72], thereby offering reliability in the presence of network disruptions. The authors in [73] show that AMQP can send a larger amount of messages per second, compared to REST. Studies show that AMQP has a low success rate in low bandwidth, but the success rate increases as bandwidth increases [72], [74].

5) MQTT-SN (Message Queuing Telemetry Transport - Sensor Networks): MQTT-SN is a variant of MQTT, which is mainly designed for very resource-constrained devices. Specifically, this protocol aims to reduce the high energy consumption and bandwidth usage of MQTT networks [75]. Although MQTT-SN can use UDP as its underlying transport protocol, it is agnostic to the underlying transport services. Apart from introducing UDP, MQTT-SN also mandates the inclusion of a gateway (GW) and forwarder. The primary function of the gateway is to transform UDP packets into standard MQTT packets and transmit the packets to a broker. There are two categories for this connection: transparent and aggregating. Using a transparent gateway, every MQTT-SN node owns a dedicated TCP connection to the broker. On the other hand, an aggregating gateway only has one TCP connection to the broker, which is then used in tunneling. The forwarder receives MQTT-SN frames on the wireless side, and encapsulates and forwards them to the gateway. For the return traffic, frames are decapsulated and sent to the clients without applying any modifications. MQTT-SN avails DTLS and TLS to provide security for UDP and TCP, respectively. Using a gateway is mandatory in MQTT-SN. The authors in [76] proved MQTT-SN performs 30% faster than CoAP.

6) CoAP (Constrained Application Protocol): CoAP eliminates the overhead of TCP by relying on UDP [77], [78], and DTLS is used to secure the connections. CoAP implements reliability by defining two bits in each packet header. These bits determine the type of message and the required Quality of Service (QoS) level. The authors in [79] show that the delay of CoAP is higher than MQTT, which is caused by packet losses when UDP is replaced with TCP.

B. QUIC

Various studies have evaluated the performance of QUIC in the context of Internet traffic. Google has deployed this protocol in its front-end servers that collectively handle billions of requests per day [14]. Google claims that QUIC outperforms TCP in a variety of scenarios such as reducing latency by 8% in Google search for desktop users and 3.6% for mobile users. In video streaming, latency in YouTube playbacks is reduced by 18% for desktop and 15.3% for mobile users. Currently, QUIC is used for 30% of Google's egress traffic.

In [80], the authors evaluate the performance of multiple QUIC streams in LTE and WiFi networks. Their experiments show that for mobile web page, median and 95th percentile completion time can be improved by up to 59.1% and 72.3%, respectively, compared to HTTP. The authors in [48] show that QUIC outperforms TCP with respect to Page Load Time (PLT) in different desktop user network conditions such as added delay and loss, variable bandwidth, and mobile environment. The studies of [81] show that QUIC outperforms TLS when used in unstable networks such as WiFi [82]. Specifically, the authors performed experiments to measure PLT for YouTube traffic with different delays (0ms, 50ms, 100ms and 200ms). These values were adopted from [83] for QUIC to simulate real networks. Assuming the PLT is x when the introduced delay is 0, these observations have been made. For the second connection, the PLT is doubled, and it is increased by 400ms for each consecutive connection. However, for HTTP-2 repeat connections, the PLT is multiplied by the connection number for each consecutive connection, i.e., $2x$, $3x$, etc. In another study [84], the authors showed that in more than 40% of scenarios the PLT of QUIC is lower than SPDY [85] and TLS combined.

The authors in [86] have evaluated the performance of HTTP2 with QUIC for Multi-User Virtual World (MUVW) and 3D web. MUVW networks mostly run on UDP [87], [88], and require the network administrator to open 50 or more UDP ports on the firewall. Since these ports carry UDP traffic, an application protocol capable of offering connection-orientated streams and security is required. QUIC is an ideal candidate to fill this void.

The implementation and integration of QUIC with MQTT are complementary to the aforementioned studies since none of them have considered the applicability of QUIC in IoT applications. Specifically, this work reveals the importance

TABLE IV
COMPARISON OF IOT APPLICATION LAYER PROTOCOLS

Protocol	CoAP	MQTT	MQTT-SN	XMPP	REST	AMQP	MQTTw/QUIC
UDP Compatible	√	X	√	X	X	√	√
TCP Compatible	X	√	√	√	√	√	X
Multiplexing Capability	X	X	X	X	X	X	√
0-RTT Capable	X	X	X	X	X	X	√
Fixing Head-of-Line Blocking	N/A	X	X	X	X	X	√
Fixing TCP Half-Open Problem (Adaptability to Lossy Networks)	N/A	X	X	X	X	X	√
Supporting Connection Migration	X	X	X	X	X	X	√

and benefits of integrating MQTT with QUIC when used in various types of IoT networks including local, wireless and long-distance networks. The results presented in this paper, in particular, confirm 56% lower packet exchange overhead, 83% lower processor utilization, 50% lower memory utilization, and 55% shorter delivery delay of MQTTw/QUIC compared to MQTTw/TCP. Table IV compares MQTTw/QUIC versus the existing application layer protocols.

VI. CONCLUSION

Enhancing transport layer protocols is essential to ensure compatibility while addressing the particular demands of IoT networks. Specifically, shifting the implementation of protocols to the user space brings substantial benefits such as offloading the cost of modifying the kernel and enhancing processing speed. In addition, given the resource-constrained nature of IoT devices, reducing communication overhead is essential. Unfortunately, these requirements are not satisfied by TCP/TLS and UDP/DTLS.

In this paper, we justified the potential benefits of QUIC compared to TCP/TLS and UDP/DTLS in IoT scenarios and presented its integration with MQTT protocol. Specifically, we showed the software architecture proposed as well as the agents developed to enable the communication between MQTT and QUIC. Three different testbeds were used to evaluate the performance of MQTTw/QUIC versus MQTTw/TCP. The results confirmed that MQTTw/QUIC reduces connection establishment overhead, lowers delivery latency, reduces processor and memory utilization, and shortens the level and duration of throughput degradation during connection migration significantly compared to MQTTw/TCP.

Some of the potential areas of future work are as follows: First, the processor utilization of QUIC is higher than TCP. This is due to the cost of encryption and packet processing while maintaining QUIC's internal state. To reduce the overhead of encryption, for example, the ChaCha20 optimization technique could be used [89]. Although packet processing cost was minimized by using asynchronous packet reception in the kernel by applying a memory-mapped ring buffer, a.k.a., `PACKET_RX_RING`, the cost is still higher than TCP. To minimize this problem, one solution is to employ *kernel bypass* [90]–[92] to bring packet processing into the user space. Second, the core functionality of QUIC assumes a maximum MTU of 1392 bytes for handshake packets [93]. This includes 14 bytes for Ethernet header, 20 bytes for IP header, 8 bytes for UDP, and 1350 bytes for QUIC. At present, the 1392 bytes is a static value in the client side, which is based on observational testing. All handshake packets are required to be padded to the full size in both directions. This limitation prohibits IP fragmentation and as a result limits the path MTU discovery. Third, as reviewed in this paper, QUIC offers many features that can be employed to further enhance the peroformance of IoT networks. For example, since most IoT applications do not rely on high throughput, we did not evaluate the effect of QUIC's flow control. However, the study of this mechanism is left as a future work, which can be justified by IoT applications such as motion detection and image classification.

REFERENCES

[1] Cisco. (2016) Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update. [Online]. Available: https://www.cisco.com/c/dam/m/en_in/innovation/enterprise/assets/mobile-white-paper-c11-520862.pdf

[2] R. Silva and J. M. Silva. (2009) An Adaptation Model for Mobile IPv6 Support in lowPANs. [Online]. Available: https://tools.ietf.org/id/draft-silva-6lowpan-mipv6-00.html

[3] A. Dunkels, "Full TCP/IP for 8-bit Architectures," in *Proceedings of the 1st international conference on Mobile systems, applications and services*. ACM, 2003, pp. 85–98.

[4] A. Dunkels, "Design and Implementation of the lwIP TCP/IP Stack," *Swedish Institute of Computer Science*, vol. 2, p. 77, 2001.

[5] T. Henderson, S. Floyd, A. Gurtov, and Y. Nishida. (2012) RFC 3782: The New Reno modification to TCP's Fast Recovery Algorithm. [Online]. Available: https://tools.ietf.org/html/rfc3782

[6] L. Xu, K. Harfoush, and I. Rhee, "Binary Increase Congestion Control (BIC) for Fast Long-Distance Networks," in *Twenty-third Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, vol. 4, 2004, pp. 2514–2524.

[7] A. Betzler, C. Gomez, I. Demirkol, and J. Paradells, "CoAP congestion control for the internet of things," *IEEE Communications Magazine*, vol. 54, no. 7, pp. 154–160, 2016.

[8] B. Dezfouli, M. Radi, and O. Chipara, "REWIMO: A Real-Time and Reliable Low-Power Wireless Mobile Network," *ACM Transactions on Sensor Networks (TOSN)*, vol. 13, no. 3, p. 17, 2017.

[9] W. Shang, Y. Yu, R. Droms, and L. Zhang. (2016) Challenges in IoT networking via TCP/IP Architecture. [Online]. Available: https://named-data.net/wp-content/uploads/2016/02/ndn-0038-1-challenges-iot.pdf

[10] B. Dezfouli, I. Amirtharaj, and C.-C. Li, "EMPIOT: An Energy Measurement Platform for Wireless IoT Devices," *Journal of Network and Computer Applications*, vol. 121, pp. 135–148, 2018.

[11] A. Freier, P. Karlton, and P. Kocher. (2011) RFC 6101: The Secure Sockets Layer (SSL) Protocol Version 3.0. [Online]. Available: http://tools.ietf.org/html/rfc6101

[12] L. Atzori, A. Iera, and G. Morabito, "The internet of things: A survey," *Journal of Computer networks (Elsevier)*, vol. 54, no. 15, pp. 2787–2805, 2010.

[13] S. Radhakrishnan, Y. Cheng, J. Chu, A. Jain, and B. Raghavan, "TCP Fast Open," in *Proceedings of the Seventh Conference on Emerging Networking Experiments and Technologies*, 2011, p. 21.

[14] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar *et al.*, "The QUIC Transport Protocol: Design and Internet-Scale Deployment," in *ACM SIGCOMM Computer Communication Review*, 2017, pp. 183–196.

[15] J. Soldatos, M. Serrano, and M. Hauswirth, "Convergence of Utility Computing with the Internet-of-Things," in *Sixth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS)*, 2012, pp. 874–879.

[16] S. Distefano, G. Merlino, and A. Puliafito, "Sensing and Actuation as a Service: A New Development for Clouds," in *11th IEEE International Symposium o Network Computing and Applications (NCA)*, 2012, pp. 272–275.

[17] S. R. Hussain, S. Mehnaz, S. Nirjon, and E. Bertino, "Seamblue: Seamless Bluetooth Low Energy Connection Migration for Unmodified IoT Devices," in *Proceedings of the International Conference on Embedded Wireless Systems and Networks (EWSN)*, 2017, pp. 927–944.

[18] B. Dezfouli, V. Esmaeelzadeh, J. Sheth, and M. Radi, "A review of software-defined wlans: Architectures and central control mechanisms," *IEEE Communications Surveys & Tutorials*, 2018.

[19] B. Dezfouli, M. Radi, S. A. Razak, T. Hwee-Pink, and K. A. Bakar, "Modeling low-power wireless communications," *Journal of Network and Computer Applications*, vol. 51, pp. 102–126, 2015.

[20] A. Nikoukar, S. Raza, A. Poole, M. Güneş, and B. Dezfouli, "Low-power wireless for the internet of things: Standards and applications," *IEEE Access*, vol. 6, pp. 67 893–67 926, 2018.

[21] S. Saini and A. Fehnker, "Evaluating the stream control transmission protocol using uppaal," *arXiv preprint arXiv:1703.06568*, 2017.

[22] S. V. Chimkode, "design of an fpga based embedded system for protecting the server from syn flood attack," Ph.D. dissertation, Goa University, 2017.

[23] Z. Ahmed, M. Mahbub, and S. J. Soheli, "Defense Against SYN Flood Attack Using LPTR-PSO: A Three Phased Scheduling Approach," *International Journal of Advanced Computer Science and Applications*, vol. 8, no. 9, pp. 433–441, 2017.

[24] D. S. S. R. A. R. Nageswara, K LakshmiNadh, "Analysis of TCP Issues in Internet of Things," *International Journal of Pure and Applied Mathematics*, vol. 118, no. 14, 2018.

[25] W. Bziuk, C. V. Phung, J. Dizdarević, and A. Jukan, "On http performance in iot applications: An analysis of latency and throughput," in *41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE, 2018, pp. 0350–0355.

[26] M. Scharf and S. Kiesel, "Head-of-line Blocking in TCP and SCTP: Analysis and Measurements," in *GLOBECOM*, vol. 6, 2006, pp. 1–5.

[27] A. Banks and R. Gupta. (2014) Message Queuing Telemetry Transport (MQTT) v3.1.1. [Online]. Available: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.pdf

[28] Z. Shelby, K. Hartke, and C. Bormann. (2014) RFC 7252: The Constrained Application Protocol (CoAP). [Online]. Available: http://www.rfc-editor.org/info/rfc7252

[29] T. Dierks and E. Rescorla. (2008) RFC 5246: The Transport Layer Security (TLS) Protocol. [Online]. Available: https://tools.ietf.org/html/rfc5246

[30] E. Rescorla and N. Modadugu. (2012) RFC 6347: Datagram Transport Layer Security Version 1.2. [Online]. Available: https://tools.ietf.org/html/rfc6347

[31] H. T. Eric Rescorla. (2017) RFC: 6347 The Datagram Transport Layer Security (DTLS) Connection Identifier. [Online]. Available: https://tools.ietf.org/html/draft-ietf-tls-dtls-connection-id-00#page-3

[32] H. T. Eric Rescorla. (2017) RFC 6347: Datagram Transport Layer Security Version 1.2. [Online]. Available: https://tools.ietf.org/html/rfc6347

[33] Google. (2016) QUIC Protocol. [Online]. Available: https://www.chromium.org/quic

[34] R. Stewart. (2007) RFC 4960: Stream Control Transport Protocol. [Online]. Available: https://tools.ietf.org/html/rfc4960

[35] Eclipse. (2018) Paho Eclipse MQTT Project. [Online]. Available: http://projects.eclipse.org/projects/technology.mosquitto

[36] Google. (2018) The Chromium Project. [Online]. Available: http://www.chromium.org/developers/design-documents/dns-prefetching

[37] Google. (2018) QUIC: A UDP-based Multiplexed and Secure Transport. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-quic-transport/

[38] J. Iyengar and M. Thomson. (2017) Loss Detection and Congestion Control. [Online]. Available: https://www.tools.ietf.org/html/draft-ietf-quic-transport-11#page-41

[39] J. Iyengar and M. Thomson. (2017) Probing a New Path. [Online]. Available: https://tools.ietf.org/html/draft-ietf-quic-transport-11#page-38

[40] H. Krawczyk, K. G. Paterson, and H. Wee, "On the Security of the TLS Protocol: A Systematic Analysis," in *Advances in Cryptology (CRYPTO)*. Springer, 2013, pp. 429–448.

[41] M. Fischlin and F. Günther, "Multi-Stage Key Exchange and the Case of Google's QUIC Protocol," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 1193–1204.

[42] A. Langley and W.-T. Chang. (2016) QUIC Crypto. [Online]. Available: https://docs.google.com/document/d/1g5nIXAIkN_Y-7XJW5K45IblHd_L2f5LTaDUDwvZ5L6g/edit

[43] S. Grüner, J. Pfrommer, and F. Palm, "RESTful Industrial Communication with OPC UA," *IEEE Transactions on Industrial Informatics*, vol. 12, no. 5, pp. 1832–1841, 2016.

[44] I. Swettn. (2016) QUIC General. [Online]. Available: https://docs.google.com/document/d/1gY9-YNDNAB1eip-RTPbqphgySwSNSDHLq9D5Bty4FSU/edit

[45] J. Iyengar and I. Swett. (2017) QUIC Loss Detection and Congestion Control. [Online]. Available: https://tools.ietf.org/html/draft-ietf-quic-recovery-08

[46] A. Aggarwal, S. Savage, and T. Anderson, "Understanding the Performance of TCP Pacing," in *Proceedings of 9th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, vol. 3, 2000, pp. 1157–1165.

[47] I. Rhee, L. Xu, S. Ha, A. Zimmermann, L. Eggert, and R. Scheffenegger. (2018) RFC 8312: CUBIC for Fast Long-Distance Networks. [Online]. Available: https://trac.tools.ietf.org/html/rfc8312

[48] A. M. Kakhki, S. Jero, D. Choffnes, C. Nita-Rotaru, and A. Mislove, "Architectural Considerations for a New Generation of Protocols," in *Proceedings of the Internet Measurement Conference*. ACM, 2017, pp. 209–303.

[49] ngtcp2 team. (2018) ngtcp2 Project. [Online]. Available: https://github.com/ngtcp2/ngtcp2

[50] Y. Cui, T. Li, C. Liu, X. Wang, and M. Kühlewind, "Innovating Transport with QUIC: Design Approaches and Research Challenges," *IEEE Internet Computing*, vol. 21, no. 2, pp. 72–76, 2017.

[51] H. Krawczyk, M. Bellare, and R. Canetti. (1997) RFC:2104 HMAC: Keyed-Hashing for Message Authentication. [Online]. Available: https://tools.ietf.org/html/rfc2104

[52] P. Rogaway, "Authenticated-Encryption With Associated-Data," in *Proceedings of the 9th ACM conference on Computer and communications security (ACM)*, 2002, pp. 98–107.

[53] P. E. H. T. Joseph Salowey, Hao Zhou. (1999) RFC 5077: Transport Layer Security (TLS) Session Resumption Without Server-Side Statel. [Online]. Available: https://tools.ietf.org/html/rfc5077

[54] S. Krishnan and F. Monrose, "DNS Prefetching and its Privacy Implications: When Good Things Go Bad," in *Proceedings of the 3rd USENIX conference on Large-scale exploits and emergent threats: botnets, spyware, worms, and more (USENIX Association)*, 2010, pp. 10–10.

[55] M. K. McKusick, G. V. Neville-Neil, and R. N. Watson, *The Design and Implementation of the FreeBSD Operating System*. Pearson Education, 2014.

[56] G. Henderson. (2018) Wiring Pi GPIO Interface Library for the Raspberry Pi. [Online]. Available: http://wiringpi.com/

[57] S. Obermaier Dominik. (2014) QUIC Loss Detection and Congestion Control. [Online]. Available: https://github.com/hivemq/hivemq-mqtt-web-client

[58] A. Y. Feng Lee. (2018) Cocoa MQTT. [Online]. Available: https://github.com/emqtt/CocoaMQTT

[59] Ralight. (2015) IOSphere Mosquitto. [Online]. Available: https://github.com/iosphere/mosquitto

[60] L. Petersen and S. Clausen. (2018) HaskellMQTT. [Online]. Available: https://github.com/lpeterse/haskell-mqtt

[61] N. Tonpa. (2017) HaskellMQTT. [Online]. Available: https://github.com/synrc/mqtt/blob/master/index.html

[62] S. Fu and M. Atiquzzaman, "SCTP: State of the Art in Research, Products, and Technical Challenges," *IEEE Communications Magazine*, vol. 42, no. 4, pp. 64–76, 2004.

[63] R. Stewart and P. Amer, "Why is SCTP needed given TCP and UDP are widely available?" *Internet Society Member Briefing*, vol. 17, 2004.

[64] S. Bendel, T. Springer, D. Schuster, A. Schill, R. Ackermann, and M. Ameling, "A Service Infrastructure for the Internet of Things Based on XMPP," in *IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOM Workshops)*, 2013, pp. 385–388.

[65] S. Vaughan-Nicholsn. (2013) Google Gives up XMPP. [Online]. Available: https://www.zdnet.com/article/google-moves-away-from-the-xmpp-open-messaging-standard

[66] R. T. Fielding and J. Reschke. (2014) RFC 7231: Hypertext Transfer Protocol–HTTP/1.1: Semantics and Content. [Online]. Available: https://tools.ietf.org/html/rfc7231

[67] B. Upadhyaya, Y. Zou, H. Xiao, J. Ng, and A. Lau, "Migration of SOAP-based Services to RESTful Services," in *13th IEEE International Symposium on Web Systems Evolution (WSE)*, 2011, pp. 105–114.

[68] R. K. Diraviyam. (2018) QUIC Loss Detection and Congestion Control. [Online]. Available: https://www.bevywise.com/blog/mqtt-vs-rest-iot-implementation/

[69] T. Savolainen, N. Javed, and B. Silverajan, "Measuring Energy Consumption for RESTful Interactions in 3GPP IoT Nodes," in *7th IFIP Wireless and Mobile Networking Conference (WMNC)*, 2014, pp. 1–8.

[70] B. Networks, "MQTTRoute Broker," https://www.bevywise.com/mqtt-broker/, 2018.

[71] P. Bhimani and G. Panchal, "Message Delivery Guarantee and Status Update of Clients Based on IOT-AMQP," in *Intelligent Communication and Computational Technologies (Springer)*, 2018, pp. 15–22.

[72] F. T. Johnsen, T. H. Bloebaum, M. Avlesen, S. Spjelkavik, and B. Vik, "Evaluation of Transport Protocols for Web Services," in *IEEE Military Communications and Information Systems Conference (MCC)*, 2013, pp. 1–6.

[73] J. L. Fernandes, I. C. Lopes, J. J. Rodrigues, and S. Ullah, "Performance Evaluation of RESTful Web Services and AMQP Protocol," in *Fifth IEEE International Conference on Ubiquitous and Future Networks (ICUFN)*, 2013, pp. 810–815.

[74] J. E. Luzuriaga, M. Perez, P. Boronat, J. C. Cano, C. Calafate, and P. Manzoni, "A Comparative Evaluation of AMQP and MQTT Protocols Over Unstable and Mobile Networks," in *12th Annual IEEE Consumer Communications and Networking Conference (CCNC)*, 2015, pp. 931–936.

[75] A. Stanford-Clark and H. L. Truong, "MQTT for Sensor Networks (MQTT-SN) Protocol Specification," *International business machines (IBM) Corporation version*, vol. 1, 2013.

[76] M. H. Amaran, N. A. M. Noh, M. S. Rohmad, and H. Hashim, "A Comparison of Lightweight Communication Protocols in Robotic Applications," *Procedia Computer Science*, vol. 76, pp. 400–405, 2015.

[77] J. Iyengar and I. Swett. (2017) Profiling of DTLS for CoAP-Based IoT Applications. [Online]. Available: https://tools.ietf.org/html/draft-keoh-dtls-profile-iot-00

[78] S. L. Keoh, S. S. Kumar, and H. Tschofenig, "Securing the Internet of Things: A standardization perspective," *IEEE Internet of Things Journal*, vol. 1, no. 3, pp. 265–275, 2014.

[79] 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 *IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP)*, 2014, pp. 1–6.

[80] P. Qian, N. Wang, and R. Tafazolli, "Achieving Robust Mobile Web Content Delivery Performance Based on Multiple Coordinated QUIC Connections," *IEEE Access*, vol. 6, pp. 11 313–11 328, 2018.

[81] S. Cook, B. Mathieu, P. Truong, and I. Hamchaoui, "QUIC: Better for what and for whom?" in *IEEE International Conference on Communications (ICC)*, 2017, pp. 1–6.

[82] A. Sathiaseelan and J. Crowcroft, "Internet on the Move: Challenges and Solutions," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 1, pp. 51–55, 2013.

[83] Igvita. (2012) QUIC Experiments for Akamai. [Online]. Available: https://www.igvita.com/2012/07/19/latency-the-new-web-performance-bottleneck/

[84] P. Megyesi, Z. Krämer, and S. Molnár, "How Quick is QUIC?" in *2016 IEEE International Conference on Communications (ICC)*, 2016, pp. 1–6.

[85] M. Belshe and R. Peon. (2012) SPDY Protocoll. [Online]. Available: https://tools.ietf.org/html/draft-mbelshe-httpbis-spdy-00

[86] H. Bakri, C. Allison, A. Miller, and I. Oliver, "HTTP/2 and QUIC for Virtual Worlds and the 3D Web?" *Procedia Computer Science (Elsevier)*, vol. 56, pp. 242–251, 2015.

[87] I. Oliver, C. Allison, and A. Miller, "Traffic Management for Multi User Virtual Environments," in *The 10th Annual PostGraduate Symposium on The Convergence of Telecommunications, Networking and Broadcasting (PGNET 2009)*, 2009.

[88] I. A. Oliver, A. H. Miller, and C. Allison, "Virtual Worlds, Real Traffic: Interaction and Adaptation," in *Proceedings of the first annual ACM SIGMM conference on Multimedia systems (ACM)*, 2010, pp. 305–316.

[89] A. Langley, W. Chang, N. Mavrogiannopoulos, J. Strombergson, and S. Josefsson, "Chacha20-poly1305 cipher suites for transport layer security (tls)," Tech. Rep., 2016.

[90] L. Rizzo, "Netmap: A Novel Framework for Fast Packet I/O," in *21st USENIX Security Symposium (USENIX Security 12)*, 2012, pp. 101–112.

[91] M. Paolino, N. Nikolaev, J. Fanguede, and D. Raho, "SnabbSwitch User Space Virtual Switch Benchmark and Performance Optimization for NFV," in *IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*, 2015, pp. 86–92.

[92] Intel. (2014) Data Plane Development Kit. [Online]. Available: https://www.dpdk.org/

[93] I. Swett. (2015) QUIC: Next generation multiplexed transport over UDP. [Online]. Available: https://www.nanog.org/sites/default/files/meetings/NANOG64/1051/20150603_Rogan_Quic_Next_Generation_v1.pdf