Application-layer Characterization and Traffic Analysis for Encrypted QUIC Transport Protocol

Qianqian Zhang and Chi-Jiun Su
Advanced Development Group, Hughes Network Systems, Germantown, MD, USA
Emails: {Qianqian.Zhang,Chi-Jiun.Su}@hughes.com.

arXiv:2310.10676v1 [cs.CR] 10 Oct 2023

***Abstract*—Quick UDP Internet Connection (QUIC) is an emerging end-to-end encrypted, transport-layer protocol, which has been increasingly adopted by popular web services to improve communication security and quality of experience (QoE) towards end-users. However, this tendency makes the traffic analysis more challenging, given the limited information in the QUIC packet header and full encryption on the payload. To address this challenge, a novel rule-based approach is proposed to estimate the application-level traffic attributes without decrypting QUIC packets. Based on the size, timing, and direction information, our proposed algorithm analyzes the associated network traffic to infer the identity of each HTTP request and response pair, as well as the multiplexing feature in each QUIC connection. The inferred HTTP attributes can be used to evaluate the QoE of application-layer services and identify the service categories for traffic classification in the encrypted QUIC connections.**

I. Introduction

Passive monitoring over the network traffic is essential for Internet service providers (ISPs) and network operators to perform a wide range of network operations and management activities [1]. Given the monitored network status, ISPs can adjust the capacity planning and resource allocation to ensure a good quality of experience (QoE). Network monitoring also facilitates intrusion detection and expedites troubleshooting to guarantee a stable service connectivity for the customers. Due to the lack of access to user applications, devices, or servers, passive monitoring is generally challenging. As concerns on the privacy violation continually grow, popular applications start to adopt encrypted protocols. For example, most prominent web-based services apply hypertext transfer protocol secure (HTTPS) to protect the security for bi-directional communications between the Internet users and servers. Consequently, encryption on the one hand protects users' privacy, but also disables the current network management mechanisms for QoE monitoring and optimization.

Among all current efforts to incorporate encryption, a new transport-layer protocol, called Quick UDP Internet Connections (QUIC), has emerged to improve communication security and QoE for end-users [2]. QUIC is a UDP-based, reliable, multiplexed, and fully-encrypted protocol. As a user-space transport, QUIC can be deployed as part of various applications and enables iterative changes for application updates. Compared with Transmission Control Protocol (TCP), QUIC uses a cryptographic handshake that minimizes handshake latency, and eliminates head-of-line blocking by using a lightweight data structure called streams, so that QUIC can multiplex multiple requests/responses over a single connection by providing each with its own stream ID, and therefore loss of a single packet blocks only streams with data in that packet, but not others in the same QUIC connection. HTTP-over-QUIC is standardized as HTTP/3 and attracted wide interest from the industry [3]. Historical trend in [4] shows that over 7% of websites are already using QUIC, and QUIC is expected to grow in the mobile networks and satellite communication systems.

Compared with other encryption technologies, QUIC brings tougher challenges on passive traffic monitoring. For example, TCP header provides useful information, including flags and sequence number, which enable ISPs to inspect the TCP communication status. However, the encryption applied to the QUIC headers leaves very limited information to infer their connection states. Meanwhile, in the satellite-based network systems, TCP traffic is usually optimized with Performance Enhancing Proxies (PEPs) [5]. However, QUIC's end-to-end encryption disables PEP optimizations, which results in an underperformance, compared with TCP PEP, even with QUIC's fast handshake. To address the aforementioned challenges, several recent works in [3] and [6]–[12] have studied the passive monitoring over encrypted network traffic. Authors in [6] and [7] investigated the HTTP request and response identification for the application-layer characterization. However, both approaches only support the TCP protocol, which cannot be easily extended to QUIC, due to the limited information in the QUIC transport header. Previous works in [8] and [9] focused on the QUIC traffic analysis for website fingerprinting and traffic classification. However, both analytic results relied on large-scale statistics of IP packets, but failed to extract the application-layer attributes. To infer the application-level information, the authors in [3] and [10]–[12] studied the network monitoring for HTTP-based encrypted traffic, including both TCP and QUIC. Although these works successfully modeled the application-layer QoE for video applications, their approaches cannot be applied to other types of web services, such as web browsing or bulk traffic. Therefore, existing literature shows distinct limitations in terms of QUIC traffic analysis on estimating the application-layer attributes.

The main contribution of this work is, thus, a novel rule-based general-purpose framework to explore the application-level traffic attributes without using any decryption towards QUIC header or payloads, for various web services. Our key contributions include:

- Based on the size, timing, and direction information visible in the encrypted QUIC packet, our proposed algorithm analyzes the associated network traffic to infer the attributes of each HTTP request and response pair, including the start and end time, size, request-response association, and multiplexing feature in each QUIC connection. Once HTTP multiplexing is detected, several requests will be matched as a group with their corresponding responses, to form a super HTTP request-response pair.
- The proposed algorithm supports both online and offline estimations for HTTP request-response pairs over QUIC protocol. In the online setting, the real-time traffic is processed by a three-module state machine to determine the instant status of the HTTP request-response communication. In the offline setting, we consider all QUIC packets at the end of the connection, where the proposed approach first infers the packets corresponding to client requests, and then identifies server's responses, and finally, pairs each request with its associated response, given the network-dependent constraints of inter-packet time and round-trip time (RTT).
- The proposed algorithm can identify QUIC control messages versus HTTP request/response data packets. To avoid overestimation of HTTP request/response size, a dynamic threshold on the QUIC packet length is designed to filter out the acknowledgment packets, setting packets and control information in the HTTP traffic when estimating the HTTP request/response data objects. Meanwhile, the proposed algorithm can handle special features in QUIC protocol, such as 0-RTT request.
- The proposed algorithm can be applied to different applications, including video traffic, website browsing, interactive web traffic, such as user login authentication, and bulk traffic for file upload and download. We tested our algorithm under various network conditions, given different maximum transfer size (MTU) and RTT, and the proposed approach gives highly accurate estimation results in both terrestrial and satellite network systems.

The rest of this paper is organized as follows. Section II provides the system overview. The algorithm design , including request estimation, response estimation, and request-response match models, is given in Section III. In Section IV, the performance evaluations are presented, and Section V discusses the limitation and future work. In the end, Section VI draws the conclusion.

II. System Architecture

In this section, we first define the input and output of the traffic monitoring task, and provide a system overview of the QUIC characterization algorithm. As shown in Fig. 1, we consider a passive monitoring module implemented at a middlebox between the client and server. The middlebox should be able to perceive complete bi-directional traffic without any omission. For example, to observe the traffic of a user, the module can be placed at the user's network access point, while to study the traffic for a cluster of clients, the algorithm can be implemented at the network gateway. Relying on the discriminative attributes that is visible in the encrypted QUIC packets, we aim to identify each HTTP pair or HTTP object, consisting of an HTTP request and its corresponding response, which contains key information for the passive monitor to infer the application-layer characterization.

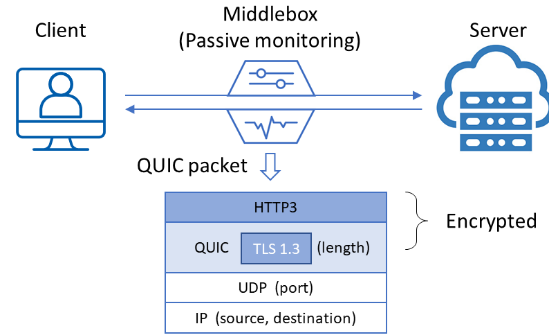

Fig. 1: Passive monitoring the bi-directional QUIC packets at a middlebox to infer the application-layer metrics.

A. Input features

Useful information in the encrypted QUIC packets mainly comes from the network layer and transport layer, including the source and destination IP addresses, source and destination port numbers, packet length, and the limited header information that is still visible in the encrypted packet. Meanwhile, packet arrival time and packet position in the sequence of a QUIC flow can also provide essential information for our application-layer characterization. In order to support a real-time estimation, the input features require only the information of individual QUIC packets. In our proposed approach, no window-based feature or large-scale statistical analysis is required. If needed, the window-based feature can be calculated in the post-processing stage using our estimation results.

In the network trace, each QUIC connection can be identified by 6-tuples, i.e., source IP, destination IP, source port, destination port, protocol, and QUIC connection ID. Within a connection, a sequence of bi-directional QUIC packets with their timing and length information can be observed, and the network operator can extract a small set of features from the network and transport layer headers as the input for the application characterization algorithm, which includes **QUIC header type**, **QUIC packet length**, **packet arrival time**, and **packet order and position**, for upstream and downstream traffic, separately. The definition of each input and the reason for choosing these features are given as follows.

1) QUIC header type: A QUIC packet has either a long or a short header. The most significant bit of a QUIC packet is the Header Form bit, which is set to 1 for long headers, and 0 for short headers. This header type is always available in the encrypted QUIC packets and stays invariant across QUIC versions [13]. The long header is used in the handshake stage to expose necessary information for version negotiation and

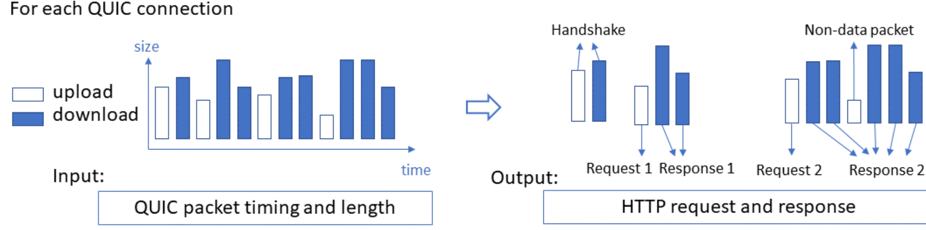

Fig. 2: Time and length information of each QUIC packet forms the input to estimate HTTP requests and responses.

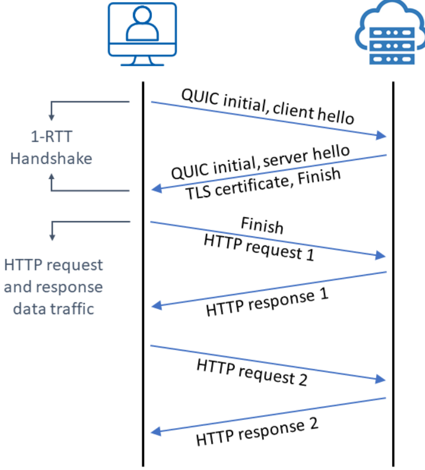

Fig. 3: 1-RTT handshake and HTTP transmission.

establishment of 1-RTT keys between two ends. Therefore, the header type provides key information on whether handshake is finished or not. Except for 0-RTT resumption, most of the HTTP requests and responses occur only after the handshake is finished. Thus, once a QUIC packet with short header is observed in newly-built QUIC connection, soon the HTTP request and response packets are expected to arrive.

2) QUIC packet length: The QUIC packet size can be used to infer whether a QUIC packet contains HTTP data content. First, let us define what an HTTP data packet is. Typically, HTTP communications follow a pattern of client's request first, and then server's response. Thus, the QUIC protocol for HTTP web applications always uses **client-Initiated bidirectional stream** [14], with a stream ID that is a multiple of four in decimal, i.e., 0, 4, 8, and etc. And the corresponding response will be transmitted over the stream with the same ID as its request. In this work, we call the client-Initiated bidirectional stream as data stream, and all the other kinds as non-data stream. Although the stream ID can provide accurate information to identify HTTP request and response, this information is encrypted and invisible at the middlebox. Therefore, to distinguish the QUIC packet with data content, we must rely on the explicit information, such as the QUIC packet length.

After the handshake stage, a QUIC packet with HTTP data content usually has a larger length, compared with non-data packets. For example, acknowledgment (ACK) is a common type of QUIC frames which usually has a much shorter size. Thus, by setting a proper threshold to the QUIC packet length, it is possible to filter out non-data packets. Considering a QUIC packet from the server to the client, if its length is smaller than the threshold $L_{\text{resp}} \in \mathbb{Z}^+$, then we consider this packet as non-HTTP-response packet, and exclude it from forming the input features of the estimation algorithm. A typical value of response length threshold is $L_{\text{resp}} = 35$ bytes. Note that, throughout this paper, the packet length specifically denotes the overall size of a QUIC packet in bytes.

3) Packet arrival time: The packet arrival time can be used to tell whether two packets belong to the same HTTP object, and whether a QUIC connection is still active. First, an HTTP response usually consists of multiple packets. When two sequential packets are transmitted from the server to the client, the middlebox needs to tell whether they belong to the same response or not. Thus, a threshold is applied to their inter-arrival time. For example, if the inter-arrival time of two response packets is less than a threshold ΔT_{resp}, then these two packets belong to the same HTTP response; Otherwise, they are associated with two different responses. A typical value for ΔT_{respe} is one RTT, and a similar threshold ΔT_{req} is applied to consolidate or separate request packets.

Second, given a detected request and an estimated response, we need to know whether they are associated HTTP request-response pair. Here, we propose a necessary requirement, where the time difference between the first response packet and the first request packet must be greater than one RTT, but smaller than 20 RTTs. If this requirement is not satisfied, then the request and response are not an associated HTTP pair. Instead, they should belong to different HTTP request-response pairs.

Furthermore, in a QUIC connection, if there is no packet transmission in any direction for more than 20 RTTs, then, we consider the QUIC connection as idle. In the idle state, before any new request is observed, all response packets from the server to the client will be disgarded.

4) Packet order and position: The packets' positions in the sequence of QUIC flow can provide guidelines for a middlebox to form HTTP request-response pairs, as shown in Fig. 2. For example, an HTTP response usually consists of multiple QUIC packets with a noticeable pattern. Based on our observation, the first response packet usually has a length that is slightly smaller than MTU size; then, the response is followed with a sequence of multiple MTU-sized packets; finally, the response

TABLE I: Input features

Input features	Purposes
Packet direction	Separate request and response packets.
QUIC Header type	Check whether handshake is finished.
QUIC Packet length	Check whether a QUIC packet contains HTTP request or response data.
Packet arrival time	Check whether two packets belong to the same object, whether an HTTP request is associated with a response, and whether a QUIC connection is still active.
Packet position and order	Build HTTP request-response pairs from a sequence of individual QUIC packets.

ends with a packet with much smaller size. The cause for this special pattern is that the response data content can have a much larger size than one MTU's payload, therefore the content will be separated into multiple data frames and transmitted via multiple QUIC packets. The slightly small length of the first response packet is caused by the combination of control frame and data frame into one UDP payload, while the last packet contains the left-over response content in this transmission which is usually much less than one MTU. Note that, this pattern is an empirical summary based on our observation and experience, which may not be always true. Later, we will apply this rule as a baseline to design the estimation algorithm with further details to cope with exceptions, such as the first response packet has a MTU length, or the last packet has a larger size. Therefore, based on the pattern, we can consolidate responses from a sequence of individual response packets, together with the requirement of inter-arrival time threshold, to form a HTTP response object. A similar pattern can be observed in the HTTP request as well. However, since most of HTTP requests have very limited content, whose size is smaller than MTU, thus, most of the HTTP requests consist of a single packet with length smaller than MTU but greater than the request length threshold $L_{\text{req}} \in \mathbb{Z}^{+}$.

Till now, we have introduced four types of inputs, and the rational for choosing these features is summarized in Table I.

B. Output metrics

Given the input features, we aim to design a rule-based algorithm to estimate the **object-level** HTTP request-response information, and the **connection-level** QUIC link information, by passively monitoring the encrypted packet sequences.

1) HTTP object level output: An HTTP pair consists of an HTTP request and its corresponding HTTP response. For the request part, our designed algorithm will output the start time, size, and the number of request packets in the estimated HTTP request. Similarly, the response output metrics include the start time, end time, size, and the number of response packets in the HTTP response. The reason to exclude the request end time from the output metric is the fact that most HTTP requests consist of single-packet, thus, the end time of a request usually coincides its start time.

TABLE II: Output metrics

Output type	Estimated output
Object-level	Request start time
	Request size
	Number of request packets
	Response start time
	Response end time
	Response size
	Number of response packets
	Number of individual HTTP request-response pairs
	Max length of last ten ACK packets
Connection-level	Connection start time
	Connection duration
	Total request size
	Total response size
	Total number of request packets
	Total number of response packets
	Number of individual HTTP request-response pairs
	Number of estimated HTTP objects
	Level of multiplexing

Since QUIC protocol supports HTTP request and response multiplexing by creating multiple streams in the same connection, thus, we will see in Fig. 4 that before the transmission of an existing HTTP response is finished, another request can be sent from the client to server using a new stream ID. In the case of multiplexing, the sequence of request or response packets belonging to different HTTP objects may be interleaved with each other, thus, it might be impossible to separate the packets for each individual HTTP object, based on their length and timing information only. In this case, we will group the interleaved HTTP request-response objects together to form a super HTTP object. And, the output meaning of an estimated super object changes slightly, where the request (or response) start time is the time stamp of the first request (or response) packet in the super object, the response end time is the time stamp of the last response packet, the request (or response) size is the total size of all request (or response) packets in the super object, and the request (or response) packet number is also the total number of all request (or response) packets. Moreover, the number of HTTP pairs denotes the number of individual HTTP request-response pairs grouped in the super object, and only in the case of multiplexing, this value is greater than one. When HTTP multiplexing happens, the response estimation can be very confusing, but the request detection is still reliable, thus the number of detected requests is counted to represent the number of individual HTTP pairs in the super object.

Lastly, the length of the ACK packets contains meaningful information for packet filtering. If a packet loss is detected at the client side and the lost packet contains key information that requires re-transmission, then the client will inform the server with the loss information by sending an ACK packet. If the

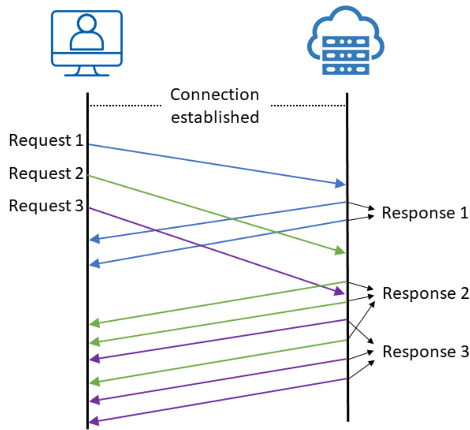

Fig. 4: HTTP request-response multiplexing.

number of lost packets keeps increasing, the ACK frame needs to contain more information, which yields an increased packet length. Therefore, by monitoring the ACK packet length in a real-time manner, the passive observer can accurately determine the threshold for the HTTP data packets, and filter out the non-date frames properly. Usually, we keep the length information of the last ten ACK packets for both directions.

2) QUIC connection level output: Once a QUIC connection has been quiet for more than 20 RTTs, we consider it as inactive, and the overall HTTP transmission will be summarized into a QUIC-connection output, and after that, all memory for this connection will be cleared. The connection-level output is shown in Table II, where the connection start time is the timestamp of the first packet from client to server, the connection duration is the time different between the first packet to the last over the QUIC connection, the total request (or response) size is the length sum of all HTTP request (or response) data packets, the total number of request (or response) packets counts the number of all HTTP request (or response) packets in the QUIC connection, the number of individual HTTP pairs equals to the number of detected requests, and the number of estimated HTTP objects equals to the number of object-level outputs estimated within this QUIC connection. For example, in Fig. 4, the number of individual HTTP pairs is three, while the number of estimated HTTP objects is only one, due to multiplexing, and in Fig. 3, the number of individual HTTP pairs and the number of estimated objects both equal to two. In the end, we define the level of multiplexing as the ratio of the number of individual HTTP pairs to the number of estimated HTTP objects. The value of the multiplexing level ranges in $[1, N_{\text{req}}]$, where $N_{\text{req}} \in \mathbb{Z}^+$ denotes the maximum number of individual HTTP pairs that our algorithm can tolerant in each super object. When multiplexing happens, the level of multiplexing is greater than one; otherwise, its value equals to one. Here, the level of multiplexing helps a network operator to classify the traffic category of a QUIC connection. For example, a web-browsing link usually has a higher multiplexing level than a video link. Key information of object-level and connection-level outputs is summarized in Table II.

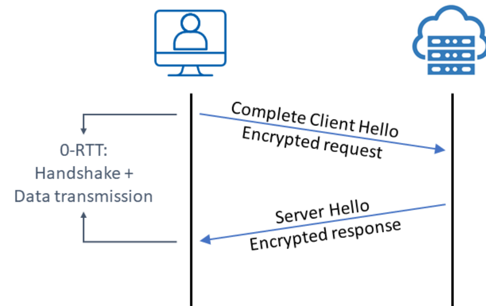

Fig. 5: 0-RTT connection resumption.

III. Algorithm and Approaches

In this section, we aim to design a rule-based algorithm so that given the input features in Table I, we can estimate the output metrics in Table II. To this end, we design a state machine with three modules, where the request estimation module infers the client requests, the response estimation module consolidates the QUIC packets into server's responses, and a match module pairs the estimated requests with their corresponding responses, under the network-dependent constraints of inter-arrival time and RTT. Furthermore, to extend the application range and increase the robustness of our algorithm, three supporting modules are introduced to automatically adjust the threshold for data packet size, detect the MTU size, and estimate the value of RTT, respectively, so that the proposed algorithm supports an accurate estimation in various network systems under different communication conditions.

A. Request estimation

In the QUIC protocol, a special features, called 0-RTT connection resumption, is shown in Fig. 5. Assume a client and a server had previously established a QUIC connection, then when a new connection is needed, client can send application data with the first packet of Client Hello and reuse the cached cryptographic key from previous communications. Notably this allows the client to compute the private encryption keys required to protect application data before talking to the server, thus successfully reduces the latency incurred in establishing a new connection. Thus, in the case of 0-RTT resumption, the HTTP request and response can happen before the handshake is finished, and a special detection mechanism is needed to infer the 0-RTT request packets. Given a QUIC packet with a long header, the third and fourth significant bits in the header indicate the type of this packet. If the type field shows (0x01), then the packet is a 0-RTT packet [14]. Next, to determine whether a 0-RTT packet contains HTTP request data, we propose three criteria: First, a 0-RTT request has usually a single packet; Second, the length of a 0-RTT request packet often ranges within $[100, 1000]$; Third, there is only one QUIC packet in the UDP payload. If all above requirements are satisfied, we can say that this 0-RTT packet is a 0-RTT request, otherwise, this 0-RTT packet is more likely to contain control information, other than HTTP request data. Again, these

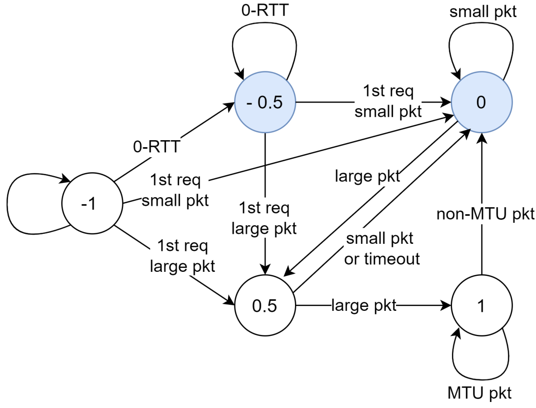

Fig. 6: State machine for request estimations, where -1 is initial state, 0 is idle state, 0.5 is waiting state, and 1 is transmission state. Once the algorithm comes to state -0.5 or state 0, a request is estimated and will be given to the match module.

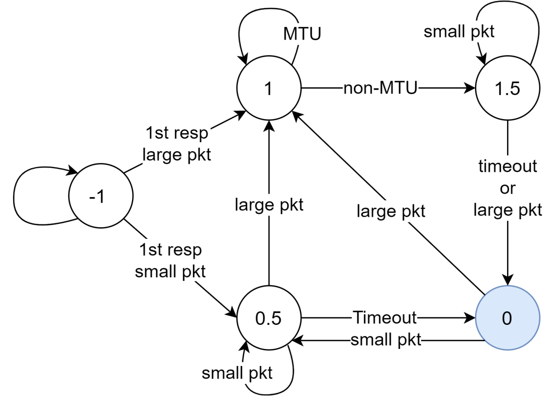

Fig. 7: State machine for response estimation, where -1 is initial state, 0 is idle state, 0.5 is waiting-to-start state, 1 is transmission state, and 1.5 is waiting-to-end state. Once the algorithm comes to state 0, a response is estimated and will be given to the match module.

criteria are empirical, which may not always be true. However, according to our observation and experience, the criteria lead to a high accuracy to estimate 0-RTT requests.

Once handshake is finished, QUIC packets will start to use short headers, which do not have a packet-type field anymore. But, similar to 0-RTT requests, request after handshake requires only one QUIC packet in the UDP payload. Meanwhile, the length of a request packet ranges between L_{req} and L_{MTU}, where L_{req} is the length threshold for request packets, and L_{MTU} is the size of MTU. In general, the MTU value L_{MTU} is network and device dependent with a value range of $[1200, 1360]$. Meanwhile, the value of L_{req} is dynamic over time. When we are inferring for the first packet of the first request, the request size threshold is set as $L_{\text{req}} = 100$ bytes. Then once the first request packet has been detected, the value of L_{req} is adjusted to 50 bytes. Later, as the HTTP request transmission continues, L_{req} will be dynamically adjusted based on the real-time traffic conditions. Details for adjusting L_{req} will be shown in Section III-D1.

Given that an HTTP request consists of either a single packet or multiple packets, in order to consolidate a sequence of request packets into a group of request objects, we design a request estimation algorithm with a state machine shown in Fig. 6. When the client sends the first Initial Hello packet to the server, a state machine is initialized for the QUIC connection with an initial state -1. During the handshake stage, if a 0-RTT request is detected, the algorithm goes to state -0.5. As long as the algorithm comes to state -0.5, a 0-RTT request will be output, and the estimated 0-RTT request will be given to the match module. On the other hand, if no 0-RTT request is found, the state will stay at -1, until handshake is finished and a new request packet is detected. If the new request packet has a length greater than $L_{\text{MTU}} - 8$, then we consider its as a large packet, and the algorithm will move to state 0.5. Otherwise, if the packet's length ranges in $[L_{\text{req}}, L_{\text{MTU}} - 8]$, we consider it a small request packet, and the algorithm comes to state 0.

State 0.5 is a waiting state, where we need the information of the next request packet to determine whether we are estimating a single-packet or multi-packet request. Therefore, at state 0.5, if we receive a large packet within one RTT, then, the current request is a multi-packet request, and more packets belonging to the same request might arrive soon, thus, the algorithm moves to the transmission state 1; otherwise, if we receive another small packet at state 0.5, the estimated request consists of two packets, and the algorithm goes to state 0, and outputs the estimated request. Meanwhile, if no new packet arrives within one RTT, then it is a single-packet request. Thus, the algorithm moves to state 0, and outputs the single-packet request. State 0 is an idle state, meaning no on-going transmission at this stage. At state 0, if a large request packet comes, the algorithm moves to state 0.5 to wait for more packets. Otherwise, the algorithm will output a single-packet request, and stays at state 0. Lastly, state 1 is a transmission state, meaning a multi-packet request is transmitting a sequence of MTU-sized packets. At state 1, if the arrived packet has a MTU-size, then transmission is on-going and the algorithm stays at state 1. If the new packet has a length smaller than MTU, then the transmission of the current request is done, so the algorithm moves to state 0, and outputs the estimated multi-packet request.

In summary, the request estimation module monitors all QUIC packets from client to server, processes the header, time, length, and order information of each packet, and outputs the estimated request to the match module.

B. *Response estimation*

Similar to the request packet, an HTTP response packet usually has only one QUIC packet in the UDP payload, and the response packet length ranges between $[L_{\text{resp}}, L_{\text{MTU}}]$, where L_{resp} is a dynamic threshold with the initial value of $L_{\text{resp}} = 35$, and the updating rule will be shown in Section III-D1. To consolidate individual packets into HTTP responses, a response estimation algorithm is designed in Fig. 7. Initially, when no request is detected, the response module stays at state -1. When at least one request and a new response packet are detected, the algorithm moves to state 1 if the response packet

size is larger than $L_{\text{MTU}} - 8$, or the algorithm moves to state 0.5 if the packet length between $[L_{\text{resp}}, L_{\text{MTU}} - 8]$.

State 0.5 is a wait-to-start state, meaning after receiving a small packet, we need to see the next packet to determine whether it is a single-packet or multi-packet response. Therefore, at state 0.5, if a large packet arrives within one RTT, the algorithm will move to state 1; if a small response packet arrives within one RTT, the algorithm stays at state 0.5, and groups the received small packets into one object. Due to different implementations, some servers may start the multi-packet response with more than one non-MTU packets. If no packet arrives during one RTT, the algorithm moves to state 0, and output an estimated response. State 0 is an idle state, meaning no transmitting response. At state 0, if a large response packet comes, the algorithm moves to state 1. Otherwise, the algorithm comes to state 0.5. State 1 is a transmission state, meaning a multi-packet response is transmitting a sequence of MTU-sized packets. At state 1, if the arrival packet has a MTU-size, then the response transmission continues and the module stays at state 1. Otherwise, the transmission finishes and the algorithm moves to state 1.5.

Lastly, state 1.5 is a wait-to-end state. Due to re-transmission, an HTTP response can end with multiple small packets. Therefore, at state 1, if the middlebox observes a small packet, it waits at state 1.5 for one RTT, during which if more small packets arrive, the algorithm consolidates these small packets with the previous MTU sequence to form one response, and stays at the state 1.5 until one-RTT timeout. If no packet arrives within one RTT, the response transmission is finished, and the module moves to state 0 to output the estimated response. However, if a large packet arrives within one RTT, then we realize that the previous response has finished, and a large packet belonging to a new response is received. In this case, the algorithm first moves to state 0, output a response consisting of all packets but not including the last one, then, the newly-arrived large packet will start another response estimation, and move the algorithm to state 1.

Thus, the response estimation module monitors all QUIC packets from server to client, processes the header, time, length, and order information of each encrypted packet, and outputs the estimated response to the match module.

C. Request-response matching

Given the estimated requests and responses, the final step is to match each request and its corresponding response to form an HTTP pair, a.k.a. HTTP object. A state machine of the matching module is given in Fig. 8, which takes the estimated HTTP requests and responses as input and outputs the object-level HTTP information. Initially, before receiving any request, the match module will ignore all response inputs and stay at state -1. After the first request is received, the algorithm comes to state 1. The definition for state 1 is that the number of requests is greater than the number of responses, meaning that some HTTP requests have been sent out, but not all of their responses are received, therefore the algorithm waits at

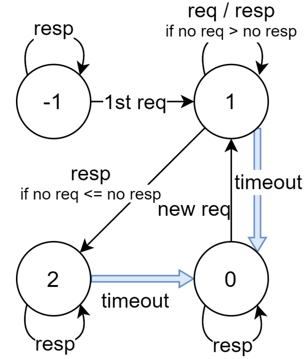

Fig. 8: State machine for match module, where -1 is initial state, 0 is idle state, 1 is waiting-for-response state, and 2 is waiting-to-output state. Once the algorithm moves over a double-line arrow, an HTTP request-response pair is estimated.

state 1 for more responses to finish the request-response match. Once the match module receives enough responses so that the number of requests becomes less than or equal to the number of responses, it moves to state 2. However, at state 1, if no request or response is received within 20 RTTs, then the match module is timeout, and the algorithm moves to state 0 to output an HTTP object consisting of all received requests and responses.

State 2 means the match module has received at least equal numbers of requests and responses, which is enough for one-to-one request-response match. However, at this moment, it is uncertain whether there will be more response objects coming, due to re-transmission and mis-estimation. Thus, the module waits at state 2 for one RTT. Any new response arrives within one RTT will be added into the current HTTP object. If no new response arrives within one RTT, the current module is timeout, and moves to state 0 to output the estimated HTTP object. If any new request is received at state 2, it will be hold until timeout, to form the next HTTP object. Lastly, state 0 is the idle state, in which all responses will be discarded, and if a request is received, the match module moves to state 1.

The match module takes all estimated requests and responses as input, and generates the HTTP request-response objects as output, while the connection-level output can be calculated, combing all estimated object-level information.

D. Supporting modules

To enable the proposed algorithm to work in different networks under various communication conditions, three supporting modules are introduced to adjust key parameters.

1) Dynamic threshold of data packet length: For the request data packet, the initial length threshold is $L_{\text{req}} = 50$, i.e., a QUIC packet from the client to server with a length smaller than 50 bytes will be considered as a non-data packet. Generally, it is easy, using L_{req}, to detect the non-data packet with a fixed or typical length, such as control frame. However, for ACK packets with variant sizes, a dynamic threshold is needed. Assume the downstream from server to client experiences packet loss, the client will inform the server with the packet missing information in the ACK packet. If the number of lost response packets keeps increasing, the ACK packet size from

TABLE III: Performance summary

	Dataset	Match accuracy	Request start time error	Request size accuracy	Response start time error	Response end time error	Response size accuracy
Youtube	Comcast, Chrome, small-scale	96%	0	99%	$\leq$ 20% RTT	50% RTT	97%
	HughesNet, Chrome, small-scale	95%	$\leq$ 1% RTT	98%	$\leq$ 15% RTT	$\leq$ 10% RTT	98%
	HughesNet, Firefox, small-scale	92%	$\leq$ 10% RTT	95%	$\leq$ 15% RTT	$\leq$ 10% RTT	99%
	HughesNet, Chrome, large-scale	94%	$\leq$ 15% RTT	96%	$\leq$ 25% RTT	$\leq$ 20% RTT	97%
Google drive login	Comcast, Chrome, small-scale	93%	$\leq$ one RTT	97%	$\leq$ 50% RTT	$\leq$ one RTT	95%
	HughesNet, Chrome, small-scale	96%	$\leq$ 10% RTT	91%	$\leq$ 10% RTT	$\leq$ 15% RTT	99%
	HughesNet, Firefox, small-scale	99%	$\leq$ 10% RTT	99%	$\leq$ 10% RTT	$\leq$ 15% RTT	91%
Google drive download	Comcast, Chrome, small-scale	87%	$\leq$ 1% RTT	85%	$\leq$ 5% RTT	$\leq$ 1% RTT	94%
	HughesNet, Chrome, small-scale	88%	$\leq$ 50% RTT	89%	$\leq$ 50% RTT	$\leq$ 50% RTT	85%
	HughesNet, Firefox, small-scale	85%	0	99%	$\leq$ 10% RTT	$\leq$ 5% RTT	99%
Google drive upload	Comcast, Chrome, small-scale	93%	10 RTTs	78%	3 RTTs	one RTT	97%
	HughesNet, Chrome, small-scale	96%	$\leq$ 50% RTT	77%	$\leq$ 20% RTT	$\leq$ 30% RTT	99%
	HughesNet, Firefox, small-scale	92%	$\leq$ 50% RTT	75%	$\leq$ 10% RTT	$\leq$ 1% RTT	99%
Facebook/ Instagram/ Google	Comcast, Chrome, small-scale	100%	0	100%	$\leq$ 5% RTT	0	99%
	HughesNet, Chrome, small-scale	100%	0	97%	$\leq$ 15% RTT	$\leq$ 20% RTT	99%
	HughesNet, Firefox, small-scale	97%	0	99%	$\leq$ 20% RTT	$\leq$ 10% RTT	94%

the client to server will become larger. Once ACK length comes to 50 bytes, the initial threshold L_{req} can no longer work.

Since the ACK packet size increases gradually, we can track the length change and adjust the threshold accordingly. For example, over a QUIC connection, once ten non-data packets have been detected, the middlebox can take the maximum length of the last ten small-sized packets as $l_{ack}^{max} = \max\{l_{ack}^{1}, \cdots, l_{ack}^{10}\}$, and adjust the request threshold by $L_{\text{req}} = l_{ack}^{max} + 10$. In the following communication, the maximum length of the latest ten non-data packets l_{ack}^{max} will be updated for every detected non-data packet, and the request threshold is updated accordingly. A similar rule applies to the response packet length, where the initial threshold is $L_{\text{resp}} = 35$; after ten non-data response packets are detected, the response threshold is updated by the maximum length of the latest ten non-data packets, plus ten bytes. Based on our analysis, the proposed scheme shows almost 100% accuracy to separate the ACK packets from the data packets for both QUIC request and response estimations.

2) Auto-detection for MTU size: The MTU size of both QUIC and UDP packets depends on the network setting, server implementation, and client device type. Therefore, MTU can take different values for different QUIC connections, or over the same connection but in different communication directions. The auto-detection algorithm for MTU size is designed as follows: The initial MTU-value for QUIC packets is set to be $L_{\text{MTU}} = 1200$. Next, for each packet, the MTU value will be updated by taking the maximum out of the length of the new packet and the current MTU value. In most cases, the MTU values for both directions over a QUIC connection can be accurately detected within the handshake stage.

3) RTT estimation: As shown both in Fig. 3 and Fig. 5, the QUIC handshake stage requires the client to starts with a Client Hello packet, and then, the server will reply with Server Hello. This round-trip pattern during the handshake stage provides a chance for RTT estimation. Especially when the QUIC connection is established without previous memory, handshake stage usually involves more than one round-trip, then the value of RTT can be calculated by averaging the time spent over these round-trips during handshake.

IV. Performance Evaluations

In this section, we evaluate the performance of the proposed algorithm, using the QUIC trace collected from various network environments. In particular, we applied Chrome and Firefox as client browsers on both Windows and Linux operation systems, over HughesNet satellite system and Comcast terrestrial system, to collect QUIC traces for video traffic, web-browsing, user login authentication, file upload, and download traffic.

In the small-scale collection, we used Wireshark to manually collect QUIC traffic, and decrypted packets by setting the SSLKEYLOGFILE environment variable. For the large-scale collection, we applied Puppeteer as a high-level API to control Chrome and play Youtube videos following some given playlists, and used tcpdump to collect packet-level QUIC trace. The large-scale dataset is limited to web browsing and video traffic from Youtube, over HughesNet satellite system, using Chrome as browser in the client-side Linux operation system. We run the large-scale data collection continuously for 11 days from Feb 9 to Feb 20, 2023, resulting in over $1,000$ times of video plays with over $11,000$ TCP connections and $18,000$ QUIC connections between our client browser with over 400 server IP addresses.

Table III shows the evaluation performance results over the small-scale Comcaset dataset, small-scale HughesNet dataset over Chrome and Firefox, and large-scale HughesNet dataset for Youtube traffic, respectively. First, our algorithm yields a high matching accuracy of over 85%, for all types of web traffic, in all environment settings. In the request estimation, other than the upload traffic, the proposed method shows an accurate estimation result, where the request start time error is

smaller than one RTT, and the request size accuracy is higher than 85%. Different from other traffic types with small-sized requests and large-sized responses, the file upload shows a reversed pattern, where the traffic from client to server is much more than the data from server to client. This uncommon pattern results in a lower accuracy of 75% in the request size estimation, and up to 10 RTTs error in the request start time estimation. In our future work, we will further refine the algorithm design, by adding more waiting states in the request state machine, to improve the request estimation for bulk upload traffic. Similarly, the response estimation shows a satisfied result of time error small than one RTT, and size accuracy of over 85%, for all web services under all settings, except for bulk upload.

Note that, compared with the terrestrial Comcast network, the satellite system has a much larger RTT due to the long propagation distance between the ground terminal/gateway and the geostationary satellite. Therefore, the evaluation results prove that our proposed algorithm can work in various networks while guaranteeing an accurate estimation result. Furthermore, due to the limited space, Table III only shows six key performance metrics from Table II, for online estimation only. Note that, the offline algorithm yields a similar estimation accuracy, and the other performance metrics also show satisfied results.

V. Limitation and Future Work

Given the empirical nature of the proposed algorithm, one limitation of our work is the performance degradation in face of excess packet loss. Massive packet loss yields a lot of data re-transmission, so that the typical transmission pattern cannot be recognized; also, the ACK packets with large sizes will be confused with the data packets, even with a dynamic length threshold. Meanwhile, the proposed algorithm only provides a coarse-grained estimation for interleaved HTTP request-response objects, since as an ISP with limited information visible in the encrypted QUIC packets, it is impossible to distinguish individual request-response pairs using interleaved timeline with length and order information only. Thus, grouping the multiplexed objects into a super HTTP object is the best estimation we could make. Furthermore, if client or server implementations apply padding as a countermeasure of traffic analysis, then the length of all QUIC packets will be MTU. In this case, our proposed algorithm might fail, given only time and order information available.

In the future work, we will apply the estimated object-level and connection-level HTTP information for network operation and management, including the traffic classification and QoE estimation. For example, different web traffics have distinct HTTP patterns, where a video connection requests content data periodically resulting in a clear and separable request-response pattern as shown in Fig. 3, while a web-browsing connection requests different types of content at the same time, inducing interleaved requests and responses. Such pattern difference enables the ISPs to classify each QUIC connection into different application categories. Moreover, the application-layer information can be applied to infer the user's QoE over the encrypted QUIC connection. For example, the download rate per object can be calculated by response size over response duration, and the time-to-first-byte can be evaluated via the estimated request start time and the response start time.

VI. Conclusion

In this work, we have analyzed the characteristics of QUIC traffic, by passively monitoring the QUIC encrypted packets to infer the application-layer attributes. To this end, we have studied the rationale of QUIC protocol design, and summarized the key pattern for HTTP request and response communications over QUIC protocol. By carefully choosing the time and size features which are still visible in the encrypted QUIC packets, we have designed a novel rule-based algorithm to estimate the attributes of HTTP requests and responses. The performance evaluation showed satisfactory results in different network systems for various web applications.

References

[1] I. Akbari, M. A. Salahuddin, L. Ven, N. Limam, R. Boutaba, B. Mathieu, S. Moteau, and S. Tuffin, "A look behind the curtain: traffic classification in an increasingly encrypted web," in *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 5, no. 1. ACM New York, NY, USA, 2021, pp. 1–26.

[2] 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 *Proceedings of the conference of the ACM special interest group on data communication*, 2017, pp. 183–196.

[3] S. Xu, S. Sen, and Z. M. Mao, "CSI: Inferring mobile ABR video adaptation behavior under HTTPS and QUIC," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–16.

[4] "Usage statistics of QUIC for websites," accessed: May, 2023. [Online]. Available: https://w3techs.com/technologies/details/ce-quic

[5] J. Border, B. Shah, C.-J. Su, and R. Torres, "Evaluating QUIC's performance against performance enhancing proxy over satellite link," in *Proceedings of the IEEE IFIP Networking Conference*, 2020, pp. 755–760.

[6] B. Anderson, A. Chi, S. Dunlop, and D. McGrew, "Limitless HTTP in an HTTPS world: Inferring the semantics of the HTTPS protocol without decryption," in *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy*, 2019, pp. 267–278.

[7] K. Jain and C.-J. Su, "Application characterization using transport protocol analysis," Oct. 22 2019, US Patent 10,454,804.

[8] P. Zhan, L. Wang, and Y. Tang, "Website fingerprinting on early QUIC traffic," *Computer Networks*, vol. 200, p. 108538, 2021.

[9] V. Tong, H. A. Tran, S. Souihi, and A. Mellouk, "A novel QUIC traffic classifier based on convolutional neural networks," in *Proceedings of the IEEE Global Communications Conference (GLOBECOM)*, 2018, pp. 1–6.

[10] T. Mangla, E. Halepovic, M. Ammar, and E. Zegura, "Using session modeling to estimate HTTP-based video QoE metrics from encrypted network traffic," *IEEE Transactions on Network and Service Management*, vol. 16, no. 3, pp. 1086–1099, 2019.

[11] M. H. Mazhar and Z. Shafiq, "Real-time video quality of experience monitoring for HTTPS and QUIC," in *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*, 2018, pp. 1331–1339.

[12] A. Bentaleb, S. Harous *et al.*, "Inferring quality of experience for adaptive video streaming over HTTPS and QUIC," in *Proceedings of the IEEE International Wireless Communications and Mobile Computing (IWCMC)*, 2020, pp. 81–87.

[13] M. Kuehlewind and B. Trammell, "Manageability of the QUIC transport protocol," *Internet Engineering Task Force, Internet-Draft draft-ietfquic-manageability-06*, 2020.

[14] J. Iyengar, M. Thomson *et al.*, "QUIC: A UDP-based multiplexed and secure transport," *Internet Engineering Task Force, Internet-Draft draft-ietf-quic-transport-27*, 2020.