TrafficGrinder: A 0-RTT-Aware QUIC Load Balancer

Robert J. Shahla
Dept. of Computer Science
Technion
Haifa, Israel
shahlarobert@campus.technion.ac.il

Reuven Cohen
Dept. of Computer Science
Technion
Haifa, Israel
rcohen@cs.technion.ac.il

Roy Friedman
Dept. of Computer Science
Technion
Haifa, Israel
roy@cs.technion.ac.il

***Abstract*—QUIC is an emerging transport protocol, offering multiple advantages over TCP. We propose a novel 0-RTT-aware load balancing scheme for QUIC. The proposed scheme is scalable, resilient to 0-RTT replay attacks, and guarantees perfect forward secrecy. It requires no modifications to the QUIC standard and it is QUIC version independent.**

0-RTT is crucial for web performance, particularly on mobile networks. Our experiments show that it reduces the time-to-first-byte by half and the server load by 40% compared to 1-RTT, regardless of the network latency.

Using both synthetic and real-world traffic traces, we show that the proposed load balancer guarantees near-optimal load balancing performance. It also guarantees faster time-to-first-byte and faster completion time compared to other load balancing schemes, such as least loaded, power-of-two-choices, and maximum session affinity.

***Index Terms*—QUIC, Load Balancing, 0-RTT**

I. Introduction

Quick UDP Internet Connections (QUIC) [1] is emerging as a challenger to TCP's dominance, aiming to resolve many of TCP's issues. With several unique features, such as multiplexing, 0-RTT, and connection migration, QUIC provides substantial advantages for latency-sensitive applications. Nevertheless, the adoption of QUIC introduces new challenges for network infrastructure. This work addresses one of these challenges: how to load balance QUIC connections between several machines while still benefiting from the most important QUIC properties.

Cloud service providers (CSPs) offer a wide range of online services. These services are provided within scalable, multi-tenant, data centers, to which millions of servers are connected. A web service is usually provided by a cluster of machines, and load balancing logic is needed to distribute user requests among these machines. Load balancers (LBs) are implemented in various layers. This work proposes a scalable Layer4 LB, called TrafficGrinder, which ensures per-connection consistency (PCC), is resilient to zero-round-trip time (0-RTT) replay attacks, and offers perfect forward secrecy (PFS) for 0-RTT packets.

This work was partially funded by the Israeli Science Foundation grant #3119/21.

PCC means that all packets belonging to a connection are always forwarded to the same machine, as long as this machine does not malfunction, even when the size of the pool changes. This property is not trivial if the LB does not keep per connection information due to scalability considerations.

QUIC 0-RTT allows a client to send application data before the connection's setup handshake is complete. 0-RTT is crucial for short connections with stringent latency requirements such as web search, advertising, and recommendation systems. To provide the 0-RTT property, the client and the server must re-use negotiated parameters from their previous connection. The client stores these parameters and provides the server with a "TLS session ticket", which allows the server to recover the 0-RTT keys that were used to encrypt the 0-RTT data [2].

A possible way to maintain 0-RTT keys across a cluster, and enable all the machines to decrypt 0-RTT packets, is by using self-contained *session tickets*. In this method, a 0-RTT key is encrypted using a shared, long-term, master key, and the encrypted key is sent to the client as a ticket. This method scales well, but does not support PFS and is susceptible to 0-RTT replay attacks. Alternatively, all the machines of the same service can employ a shared *session database*, where the 0-RTT key is stored. Then, the database index of the key is sent to the clients. This method ensures PFS and is resilient to 0-RTT replay attacks (if the 0-RTT keys are deleted upon use). It, however, is not scalable [3].

TrafficGrinder guarantees *session affinity* to short connections, which benefit the most from 0-RTT; i.e., these connections are assigned to the same machine to which the previous connection of the same client was assigned. Since session affinity is likely to impede the LB's ability to balance the load between the machines, TrafficGrinder assigns connections that are expected to be heavy according to load balancing considerations, even if this comes at the expense of 0-RTT for these connections, given that the benefit of 0-RTT to long connections is less profound than for short connections.

The contribution of this work is threefold:

1) We propose a novel load balancing scheme for QUIC. The proposed scheme guarantees 0-RTT to almost all the connections that can benefit from it. It is scalable and resilient to 0-RTT replay attacks. It ensures PCC and guarantees PFS for 0-RTT packets. All these properties

do not hurt the load balancing performance of the proposed scheme, which is similar to that of the least loaded and power-of-two-choices schemes. To our knowledge, our scheme is the first to achieve all of these features together.
2) We propose a connection ID (CID) encoding scheme that enables the LB to get up-to-date information about the real-time load of the various machines, without any communication overhead.
3) We evaluate the advantages of 0-RTT in HTTP/3 [4], using Facebook server and client implementation [5].

The rest of the work is organized as follows. Section II covers 0-RTT data transmission in QUIC, the maintenance of 0-RTT keys in a cluster, and the top-d items identification problem used by TrafficGrinder. Section III presents TrafficGrinder's design. Section IV presents the proposed CID encoding scheme. Section V discusses the performance impact of 0-RTT. Section VI evaluates TrafficGrinder. Section VII presents related work, and Section VIII concludes the work.

II. Background

A. 0-RTT Data

In QUIC [1] and TLS 1.3 [3], a client that has recently terminated a connection with a server can send this server new data during the first flight of a new connection. Such data, called 0-RTT data, is encrypted using a 0-RTT key established during the previous connection. In TLS 1.3, a server can retain a 0-RTT key generated in a previous connection with a client using one of two methods [3].

The first method is to use a self-contained session ticket (Fig. 1a). In this method, the server holds a master key K_m, which is only known to the server. The server uses K_m to encrypt the 0-RTT key k of the next connection with a specific client. The result is $Enc_{K_m}(k)$. During connection i with a specific client, the server sends this client the key k along with $Enc_{K_m}(k)$, both encrypted using the session key of connection i; these, therefore, remain secret. The client uses k to encrypt the 0-RTT data in connection $i+1$ (i.e., before a session key is determined for this connection), and attaches $Enc_{K_m}(k)$ to its Initial packet (the first QUIC packet sent for connection setup). The server decrypts $Enc_{K_m}(k)$ using its master key K_m to obtain k, and then uses k to decrypt the 0-RTT data. The client discards k when it completes the handshake of connection $i+1$ and sets up a new key for the rest of this connection. During this connection, the server sends a new pair of k and $Enc_{K_m}(k)$ to the client for connection $i+2$.

The second method is to use a session database (Fig. 1b). During connection i, the server generates a 0-RTT key k and stores it in a secure database, associated with some index j. During connection i with the client, the server sends $\langle k, j\rangle$, both encrypted with the session key of connection i. To send 0-RTT data during connection $i+1$, the client attaches j to the Initial packet, and $Enc_k(data)$ to the 0-RTT packet(s). To process the 0-RTT data of connection $i+1$, the server retrieves the 0-RTT key k from its database and decrypts the data. The key k is deleted from the database after it is used.

TABLE I
Various Modes and Methods for 0-RTT in a Cluster With an LB

Cluster Machine Operation	0-RTT Success	PFS	Replay Attack Prevention	Scalable
Individual Session DB	Low	Yes	Yes	Yes
Individual Session Ticket	Low	No	No	Yes
Collaborative Session DB	High	Yes	Yes	No
Collaborative Session Ticket	High	No	No	Yes

PFS is an essential security feature of 0-RTT data. In [6], the concept of PFS is introduced in the context of identity-based key exchange protocols. The idea is that if one endpoint of a session is compromised after the session has ended and long-term keys are stolen (K_m in our case), the attacker should not be able to decrypt the 0-RTT data sent during the session. We adhere to the definition of PFS presented in [7]: A protocol has PFS if compromised long-term keys do not compromise past session keys.

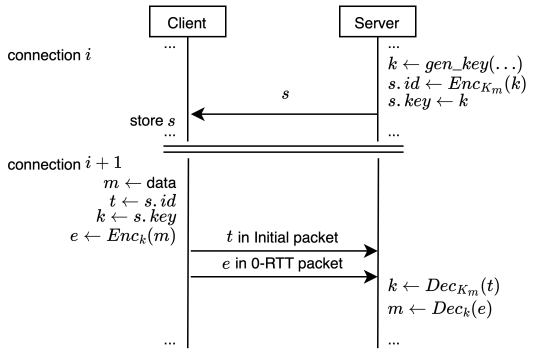
B. Ensuring 0-RTT for a Cluster of Machines with an LB

There are two main operation modes for a cluster of machines: individual and collaborative. We now discuss how 0-RTT data transmission can be supported in each mode, using a session ticket and using a session database (Table I).

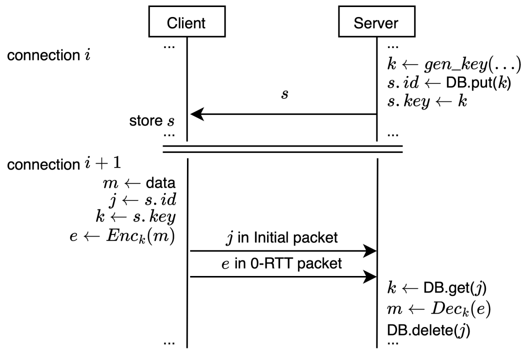
In the individual operation mode, each machine in the cluster operates separately and independently of the others. A new connection from a client is assigned to one of the machines, usually by an LB, without taking into account how past connections of the same client have been assigned. In this case, both the session database method and the session ticket method are straightforward.

Assuming there are n machines, the probability that a client is assigned to the same machine that issued the 0-RTT key in the previous connection is $\frac{1}{n}$ (in a uniform distribution). Hence, clients will not be able to take advantage of 0-RTT in most of their connections. If a session database is used, PFS and 0-RTT replay resilience are guaranteed because the 0-RTT keys are deleted after they are used. If a session ticket is naïvely used, PFS is not guaranteed since an attacker that knows the master key K_m can decrypt all the 0-RTT data whose keys were encrypted with K_m. In addition, 0-RTT replay attacks are possible because a session ticket can be used multiple times with the same master key. Note that PFS and 0-RTT replay resilience can be guaranteed using puncturable encryption [8]. Nonetheless, it is impractical due to the latter's high decryption cost [9].

In a collaborative operation, a client's connection is assigned to one of the n machines, still without considering past assignments for connections of the same client. The machines, however, can cooperate to support 0-RTT better than under individual operation. The database in the session database method and the master key K_m in the session ticket method are shared. In both methods, every client can send 0-RTT data and the machine can decrypt this data, even if the previous connection of this client was assigned to a different machine.



(a) The session ticket method



(b) The session database method

Fig. 1. The two 0-RTT key maintenance methods - Function gen_key generates a secret key, to be used for the encryption of the 0-RTT data; Enc and Dec are encryption and decryption functions; m contains the 0-RTT data; DB is the database, with functions $put(k)$ which inserts element k and returns its store index, and $get(j)$ which retrieves the element stored in index j.

When the session ticket method is used, an attacker can retransmit a 0-RTT packet, possibly to multiple machines and the machines would successfully process the packet [3]. To prevent this replay attack, [3], [10] proposed implementing a replay cache alongside the load balancer to detect and drop replayed packets. In the session database method, replay attacks are avoided if the 0-RTT key is deleted from the shared database after it has been used. This method is adopted by Cloudflare's session resumption implementation [11].

When the session ticket method is used, the machines are required to periodically change K_m to restrict the potential damage should K_m become compromised, which allows an attacker to decrypt all 0-RTT data sent by the clients using K_m-encrypted tickets. K_m change, however, prevents the machines from processing 0-RTT data encrypted with the old K_m. To address this problem, [11] suggested updating the master key every hour and having each machine retain the master keys of the previous 18 hours. This method compromises PFS to some degree, but ensures that 0-RTT can be processed in a new connection for a long time after the previous connection of the same user ends.

Note that using a shared database is an expensive solution because it requires a fast large-scale distributed key-value store. Using a shared master key is also an expensive solution because of the distributed mechanism required for changing the master keys and the maintenance of a replay cache.

C. Top-d Items Identification

TrafficGrinder uses top-d items identification over a stream of items [12]–[15]. In the context of TrafficGrinder, the items are connections and the objective is to identify the d connections that appear most frequently or those that deliver the most bytes. This is similar to the heavy hitters problem [16], whose goal is to detect items that consume more than a θ-fraction of the traffic [15], [17].

There are two main categories of algorithms for solving the top-d identification problem: sketch-based and counter-based. Sketch-based algorithms provide answers whose accuracy bound is maintained within a certain probability [13]. Counter-based algorithms maintain a counter for each item in a table of monitored items, and various algorithms use different methods for admitting and evicting monitored items [14], [15].

Space Saving [15] is a counter-based algorithm that maintains a table of monitored items as well as a counter for each monitored item. When a new item arrives there are a few options: (a) if the item is already in the table, its counter is incremented; (b) if the item is not in the table and there is a free entry in the table, the item is added with $counter = 1$; (c) if the item is not in the table and there is no free entry, the item replaces the item m whose counter is minimum and its counter is set to $c + 1$, where c is the counter of the removed item. RAP [14] improves this algorithm by employing a randomized admission filter: a new unmonitored item is admitted into a full table, and the item whose counter c_m is minimum is evicted, with probability $\frac{1}{c_m+1}$. The counter of the new item is set to $c_m + 1$. We use RAP to identify long connections in order to fine-tune the load balancing algorithm.

III. LB Design

A. High-Level Description

TrafficGrinder combines the advantages of the collaborative and the individual approaches: it provides PFS; it is resistant to replay attacks; it is scalable; and it does not require a shared database or a shared master key. The machines operate independently of each other, using the individual session database approach. Recall that this approach requires session affinity and that 0-RTT packets are assigned to the same machine that issued the 0-RTT key during the previous connection of the same user.

The connection patterns of clients vary by nature. Studies on datacenter traffic reveal that most connections are small, lasting less than a few hundred milliseconds [18]. In [18], it is shown that 80% of the connections are smaller than 10KB and that only a small fraction of them reach hundreds of MBs. It is also shown that most of the traffic is sent by the top 10% large connections.

We consider a Direct Server Return (DSR) LB, which means that the LB sees the packets sent by the clients, but

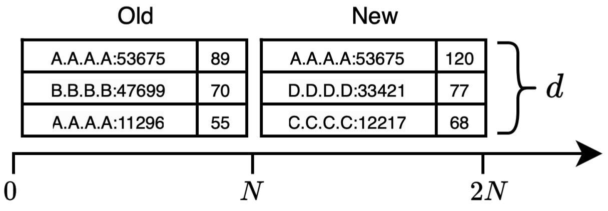


Fig. 2. An example of a sliding window RAP, with $f = 2$ RAP instances and $d = 3$ connections. The current interval is $[N, 2N]$, and the New RAP instance is updated for each received packet. To perform a query, the LB aggregates the results from both RAP instances. For interval $[2N, 3N]$, the LB drops the Old RAP instance and creates a new one.

not necessarily the ones sent by the server. Similar to prior work [19], we assume that a connection whose client sends a large number of packets over a short period of time (i.e., a "heavy hitter client") is likely to impose a higher load on the server than a connection whose client sends a small number of packets (a "mouse client"). This assumption stems from the fact that the client sends an ACK packet for every one or two data packets it receives from the server. Following [20], we further assume that the connection pattern of each client is relatively persistent over a short period of time, i.e., if connection i of a certain client imposes a light load on the server, then, with high probability, connection $i + 1$ of this client will also impose a light load on the server. Similarly, if connection i imposes a high load on the server, then, with high probability, connection $i + 1$ will also impose a high load. Thus, if all the connections of each heavy hitter client are assigned to the same machine during some period of time, the load imposed on the machines is likely to be imbalanced.

Following the above, we make the following observations:

1) Connections of mouse clients are delay-sensitive and adding an extra RTT to these connections substantially increases their latency. Hence, such connections should be serviced using 0-RTT.
2) Connections of heavy hitter clients should be assigned according to load balancing considerations, even if this increases their latency from 0-RTT to 1-RTT. This is because these connections are likely to impose a high load on the server, while, on the other hand, they last a long time so extending them by an additional RTT does not affect their clients' quality of experience.

To distinguish between clients, TrafficGrinder remembers for every machine m the recent d connections whose clients sent the largest number of packets. These connections are considered the "top-d connections for m". A client is considered a heavy hitter client as long as at least one of its connections is in the top-d set of at least one machine. Otherwise, it is considered a mouse client.

TrafficGrinder maintains a load table that stores the up-to-date load information for each machine (Table II). Each machine is allocated one line (entry) in this table, with load and freshness fields. Additionally, TrafficGrinder maintains a d-entry table for each machine, which maintains the top-d connections assigned to this machine. Each entry in such a

TABLE II
THE LOAD TABLE MAINTAINED BY TRAFFICGRINDER, WITH A PER MACHINE ENTRY

Machine	Load	Freshness
m_1		
...		
m_n		

table contains the source IP address and the source UDP port number of a connection. It also contains an estimate of the number of packets sent by the client of this connection.

To identify the top-d connections for each machine m, TrafficGrinder uses the RAP algorithm with a sliding window [14], [21]. To implement a sliding window, f RAP instances are maintained for each machine. Each instance contains d entries and is used for an interval of N packets. TrafficGrinder updates the RAP instance associated with the current interval when a packet is received. When a query is performed, TrafficGrinder aggregates the results from all RAP instances. When a RAP instance interval ends, TrafficGrinder drops the instance associated with the oldest interval and creates a new instance for a new interval. Note that for every machine m, the table may contain up to $d \cdot f$ connections. Fig. 2 depicts this process.

The value of d can be chosen based on the distribution of the traffic sent across connections. If the traffic is dominated by a few high-volume connections, a small value of d is sufficiently good. In this work, we set a threshold on the percentage of total traffic we want to capture, and then choose d such that the top-d connections contribute that percentage of the traffic.

When a client sends a QUIC Initial or 0-RTT packet, TrafficGrinder uses a source-IP hash function to determine the default machine m_i to handle this connection. It then consults the load table to decide if m_i is overloaded or not. If m_i is not overloaded, the connection is assigned to it. If m_i is overloaded, TrafficGrinder consults m_i's top-d connections table. If none of the IP addresses in this table belongs to the considered client, the connection is assigned to m_i; otherwise, the connection is assigned to the least loaded machine.

B. Load Balancing Policy

In the following discussion, when a CID is mentioned, we refer to a server-allocated CID, which appears in the source CID field of a long-header packet, such as an Initial packet or a Handshake packet, sent by a server machine. Such a CID also appears in the destination CID field of a short-header packet, sent by a client and in the destination CID field of a long-header packet sent by a client after it receives the first response from the server (a response that contains the server-allocated CID). TrafficGrinder uses the algorithm in Fig. 3 to process a client packet P and then forward P to a certain machine. The algorithm is QUIC version-independent [22], since it relies only on the Header Form bit in the packet header to distinguish between long- and short-header packets, and on the CID field.

When TrafficGrinder receives a packet P, it first checks if the packet destination CID is routable (Section IV). This requires it to decrypt the CID using K_{CID} and to validate the machine ID it contains (Section IV-B). If the CID is routable, the client has received a server-allocated CID from a machine m in a previous packet and the packet should be forwarded to m. TrafficGrinder then updates the load table and the top-d table of machine m according to the freshness mechanism (Section IV-C) and the RAP update process (Section III-A), respectively. It then forwards the packet to m.

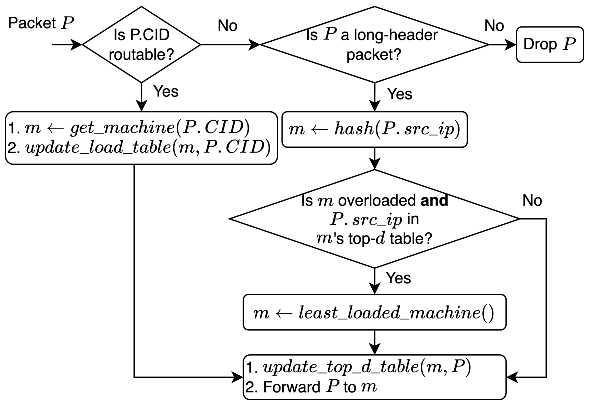
If the CID is not routable and this is a long-header packet, this means that this packet was sent by a client that has not yet received a server-allocated CID. This implies that the client has not yet received a response from the server, because the first response contains a server-allocated CID that the client must use in subsequent packets [1]. In this case, TrafficGrinder selects a machine m based on the source IP hash. We use Maglev consistent hashing [23], which ensures that, with high probability, the same client is assigned to the same machine throughout different connections, even if the machine pool changes. Then, TrafficGrinder uses the load and the top-d table of m to determine whether m is overloaded and the client is a heavy hitter for m. If this is the case, TrafficGrinder assigns the new connection to the least loaded machine m_{LL}, even if this prevents the client from sending 0-RTT data.

The least loaded machine is selected using the load table, which contains the estimated waiting time for each machine (Section IV-D). A machine m_i is considered overloaded when its estimated waiting time exceeds a certain percentile p of the estimated waiting times of all the machines and the following inequality holds:

$$\widetilde{W}_{m_i} - \widetilde{W}_{m_{LL}} > \alpha \cdot RTT^{mean}. \quad (1)$$

Here $\widetilde{W}_{m_i}$ is the estimated waiting time of machine m_i, $\widetilde{W}_{m_{LL}}$ is the estimated waiting time of the least loaded machine, α is a parameter whose default value is 1, and RTT^{mean} is the mean RTT between the clients and the cluster. If (1) holds, the client's waiting time for machine m_i is expected to be larger than $\alpha \cdot RTT$. Thus, it is better to assign this client to the least loaded machine and pay the penalty of an additional RTT. Equation 1 is evaluated only for machines whose estimated waiting time is among the top $1-p$ percentile of the estimated waiting times. This leads to the effect of load sharing rather than load balancing.

An attacker could potentially obtain a routable server-allocated CID c by establishing a connection with a machine and then use c in a subsequent Initial packet to control the selection of the machine to which its connection is assigned. To prevent this attack, TrafficGrinder uses the freshness field in c to determine if it is too old to be used. An attacker could also use c to overload a specific machine by sending a large number of packets to it. We, however, consider this to be a "standard DDoS attack", which can be prevented using mechanisms that are out the scope of this work. We note that with a high probability, a client whose IP address changes between two connections may not be able to use 0-RTT.



Packet P
Is P.CID routable?
No
Is P a long-header packet?
No
Drop P
Yes
Yes
1. $m \leftarrow get\_machine(P.CID)$
2. $update\_load\_table(m, P.CID)$
$m \leftarrow hash(P.src\_ip)$
Is m overloaded **and** $P.src\_ip$ in m's top-d table?
No
Yes
$m \leftarrow least\_loaded\_machine()$
1. $update\_top\_d\_table(m, P)$
2. Forward P to m

Fig. 3. The LB algorithm upon receiving a packet P; $get\_machine$ and $update\_load\_table$ are elaborated on in Section IV and Fig. 5 (Stage 1 and Stage 2 respectively); Function $update\_top\_d\_table(m, P)$ updates the top-d table of machine m with the IP address and port number of P.

IV. Encoding the Server Load into the Server-Allocated CID

A. Dividing the CID into Fields

A scheme for encoding the machine mapping into the CID was proposed in [24]. Its main goal is to support the QUIC connection migration feature. It divides the CID into three fields: an octet long field designated for configuration of the CID, a field for encoding the machine ID, and a field for a nonce value.

We now describe a novel idea for how to expand CID usage by incorporating more fields into it. We propose a CID encoding scheme consisting of three fields: machine ID, load, and freshness (Fig. 4). The configuration field is irrelevant to our discussion. The machine ID field is used for forwarding packets to the chosen machine, as proposed in [24]. Each machine is allocated a unique ID, which is known to the LB, and this ID is encoded in the first l_{ID} bits of the CID to guarantee PCC. The next l_l bits are used for encoding the machine load and the last l_f bits are used to indicate the CID's freshness.

The CID encodes sensitive information that should not be disclosed to clients. Attackers can use this information to estimate the number of machines in a cluster and the load distribution among the machines. In addition, this information can be used by an attacker to link multiple CIDs to the same client, thereby violating the unlinkability property of QUIC [1]. To address these concerns, the considered three fields of the CID are encrypted by the machines using AES-128-ECB, and a shared secret key K_{CID} is used by the machines and the LB [24], [25]. Sharing, maintaining, and refreshing K_{CID} is beyond the scope of this work.

B. CID Routing

One of the most important properties of QUIC compared to TCP is live connection migration. This means that a client is allowed to change its IP address and/or port number during the connection lifetime. This feature is helpful for clients that

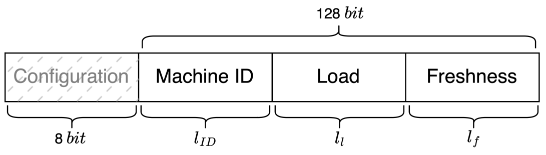


Fig. 4. CID breakdown in the proposed scheme.

change their IP address due to mobility and those that are behind a Network Address Translation (NAT) gateway.

The following description is based on [24]. The machines use the CID to identify the connection to which every packet belongs, independently of the client address. To facilitate connection migration while preserving PCC, when a machine generates a CID (a "routable CID"), it encodes its machine ID within the CID. The LB retrieves the machine ID from the destination CID of every packet sent by the clients and forwards the packet to the appropriate machine. If the machine ID retrieved from the CID does not match any of the machines in the pool and the packet is a short-header packet, the LB drops the packet (Stage 1 in Fig. 5).

C. CID Machine Load

An LB needs to estimate the load imposed on each machine in order to make load balancing decisions [26], [27]. There are two ways to measure the real-time load of a machine: (a) to actively query the machine for its load value [26] or (b) to monitor the traffic sent to the machine (and from the machine if the DSR mode is not used) [28]. Actively querying the machines for their load requires additional communication overhead between the LB and the machines. Passively monitoring the traffic sent to (or received from) every machine is less precise because even though the LB can collect statistics about the traffic sent to the machines, these statistics do not necessarily reflect the actual load imposed on each machine.

We propose that each machine informs the LB about its load value using the CIDs it allocates. In this scheme, no additional communication overhead is incurred between the LB and the machines, and the information obtained by the LB is up-to-date. The LB decrypts the CID and reads the load value upon receiving a packet. The LB should determine whether the load value is up-to-date when many connections are concurrently established, and each of them provides a different load value in its CID.

To help the LB determine when a specific CID has been allocated by the corresponding machine, we use the freshness field in the following way. Each machine maintains a freshness value, which is initialized to zero upon startup. A machine m increments the freshness value when a new CID is allocated and encodes it into the freshness field of the allocated CID. The LB updates the load of m (see Table II) with the largest observed freshness value and with the load value corresponding to that freshness value.

We note that the CID's freshness field is also used as the nonce field proposed in the IETF draft [24]. If all possible freshness values are used, then K_{CID} should be refreshed.

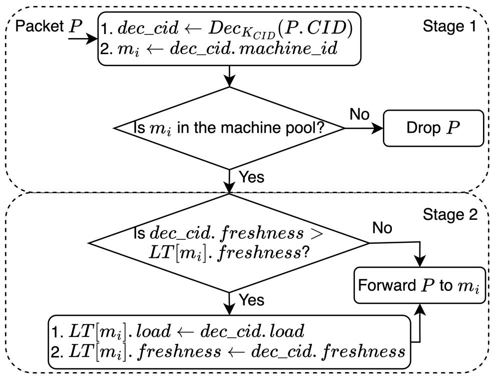

Packet P
1. $dec\_cid \leftarrow Dec_{K_{CID}}(P.CID)$
2. $m_i \leftarrow dec\_cid.machine\_id$
Stage 1
Is m_i in the machine pool?
No
Drop P
Yes
Stage 2
Is $dec\_cid.freshness >$ $LT[m_i].freshness$?
No
Forward P to m_i
Yes
1. $LT[m_i].load \leftarrow dec\_cid.load$
2. $LT[m_i].freshness \leftarrow dec\_cid.freshness$

Fig. 5. Stage 1 is the CID routing mechanism; Stage 2 is the freshness mechanism; LT is Table II; P is a short-header packet.

Stage 2 of Fig. 5 presents the freshness mechanism. The LB verifies that the freshness value is greater than the highest freshness value it has observed for the considered machine m. If this is indeed the case, the LB updates the maximum freshness value it has observed for m, as well as the load value. Otherwise, the LB ignores the load and freshness values encoded in the CID and forwards the packet to m.

D. Machine Load Estimation

Quantifying the machine load as a single number is challenging. A service that is CPU-intensive, memory-intensive, or I/O-intensive may quantify its load differently based on the type of service it provides. Several methods for quantifying the machine load have been proposed [29]. We employ a method that is agnostic to the type of service [30], [31]. To this end, a machine is characterized as a $G/G/1$ queue [32], where G indicates a general (arbitrary) distribution function. It is assumed that the requests' inter-arrival time and service time follow a general, but unknown, distribution.

Denote the mean arrival rate of requests to the machine and the mean service rate of the machine as λ and μ, respectively. Denote the mean inter-arrival time and the mean service time as $T_a = 1/\lambda$ and $T_s = 1/\mu$, respectively. Thus, $\rho = \lambda/\mu$ is the utilization of the machine. When $\rho > 1$, the queue length grows indefinitely. When $\rho < 1$, the queue is stable and statistical analysis can be applied. In this case, the machine load is defined as the mean waiting time of a request in the queue, denoted as W, estimated using Kingman's formula [33]:

$$W \approx \left(\frac{\rho}{1-\rho}\right)\left(\frac{c_a^2 + c_s^2}{2}\right) T_s,$$

where $c_a^2 = \sigma_a/T_a$ and $c_s^2 = \sigma_s/T_s$ are the coefficients of variation of the inter-arrival time and service time, respectively.

Each machine estimates the parameters of Kingman's formula by monitoring a sliding window of l requests in the following way:

$$\widetilde{T_a} = \frac{1}{l}\sum_{i=1}^{l} T_a^i, \quad \widetilde{T_s} = \frac{1}{l}\sum_{i=1}^{l} T_s^i.$$

$$\widetilde{\sigma_a} = \sqrt{\frac{1}{l-1}\sum_{i=1}^{l}(T_a^i - \widetilde{T_a})^2}.$$

$$\widetilde{\sigma_s} = \sqrt{\frac{1}{l-1}\sum_{i=1}^{l}(T_s^i - \widetilde{T_s})^2}.$$

$$\widetilde{c_a} = \frac{\widetilde{\sigma_a}}{\widetilde{T_a}}, \quad \widetilde{c_s} = \frac{\widetilde{\sigma_s}}{\widetilde{T_s}}, \quad \widetilde{\rho} = \frac{\widetilde{T_s}}{\widetilde{T_a}}.$$

T_a^i and T_s^i are the inter-arrival time and service time of the i^{th} request in the window, respectively. The machine then uses the above estimators to calculate the estimated waiting time as follows:

$$\widetilde{W} = \left(\frac{\widetilde{\rho}}{1-\widetilde{\rho}}\right)\left(\frac{\widetilde{c_a^2}+\widetilde{c_s^2}}{2}\right)\widetilde{T_s}.$$

The load is then quantified as $\widetilde{W}$. As a rule of thumb, the above Kingman's formula is used when $0.5 < \widetilde{\rho} < 1$; otherwise, the waiting time is estimated using Little's law [34]:

$$\widetilde{W} = \frac{L}{\mu} = L \cdot \widetilde{T_s},$$

where L is the number of requests in the queue.

To summarize, TrafficGrinder ensures the following security properties:

1) 0-RTT replay resistance: By using the individual session database, the scheme ensures that each 0-RTT key is available only in one machine database. This, along with requiring each machine to delete the 0-RTT key after using it, ensures that 0-RTT data is processed at most once [3].
2) PFS: Deleting the 0-RTT key after the first use ensures that if the session database of a machine is compromised, an attacker will not be able to decrypt 0-RTT data of past connections [3].

V. The Importance of 0-RTT

The goal of this section is to demonstrate the performance advantage of 0-RTT and motivate our load balancing scheme. During a 1-RTT handshake, the server must authenticate itself by sending a certificate with a relevant digital signature to the client. Client authentication is optional and is done following the server's explicit request. When 0-RTT is used on the other hand, the server is authenticated by responding to the client 0-RTT data, thereby proving its knowledge of the previously agreed upon 0-RTT key [2], [3]. Sending a certificate requires non-negligible computational resources from the server. We run several experiments and evaluate the advantage of 0-RTT over 1-RTT on the server load and the client latency. During these experiments, two handshake modes are considered: (a) 0-RTT and (b) 1-RTT without client authentication.

For these experiments we use Facebook's HTTP/3 client and server implementations [5]. Since we do not consider the network latency, both client and server are executed on the same host. The client establishes a connection with the server and requests a 10-byte message. Upon obtaining the response, the client closes the connection. The latency is measured as the time between the client sending its first packet and it receiving the first short-header (data) packet. The server runs in a Docker container, and all the clients run in a single different container. The server's container uses four CPUs and the clients' container uses eight CPUs. Each CPU is an Intel(R) Xeon(R) E5-2667 v4, 3.20GHz and each container is allocated 32GB of memory. During each experiment, $100,000$ connections are established, but the maximum number of connections existing at the same time is 200. Each experiment is repeated 21 times.

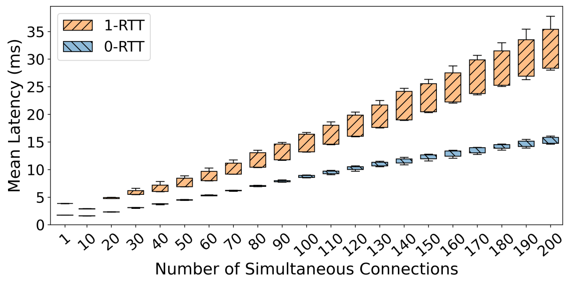
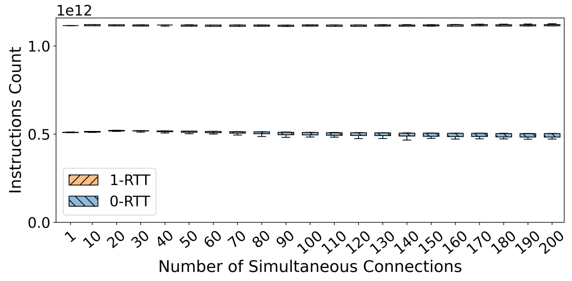


Fig. 6. 0-RTT latency vs. 1-RTT latency for $100K$ connections. The whiskers represent the 5th and 95th percentiles.

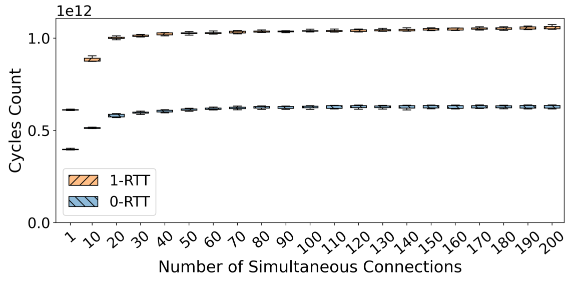
Fig. 6 depicts the mean latency for different numbers of simultaneous connections. The mean latency is calculated across $100,000$ connections for each iteration and the boxplot spans 21 iterations. Fig. 6 shows that the mean latency using 1-RTT is greater than that of 0-RTT by a factor of approximately two. In addition, it is evident that the variance of the mean values across several iterations is greater with 1-RTT than with 0-RTT and this variance grows with the number of simultaneous connections. Since the mean metric is sensitive to outliers, this suggests that 1-RTT is more sensitive to changes in the server load and leads to more outliers with high latency.

Fig. 7a shows the server instruction count and Fig. 7b shows the CPU cycles as a function of the simultaneous connections number. Each iteration, the server instruction count and CPU cycles are measured using the "perf" tool [35] and the boxplot spans 21 iterations. The graphs show over a 50% reduction in instructions count and a 40% decrease in CPU cycles with 0-RTT compared to 1-RTT. The higher 1-RTT cost is due to sending a certificate and a signature, which is not required with 0-RTT. This implies that 0-RTT brings two significant advantages: reduced client latency and lower server load.

With respect to the client latency, it is clear that 0-RTT brings a higher benefit when the RTT is large. Various studies have conducted real-world measurements to analyze the RTT in LTE cellular networks [36], [37]. Reference [36] showed a strong correlation between the RTT and the load on the cellular network. It also showed that for three different network operators in Denmark, the measured user plane RTT has an average of 51, 72, and 120 milliseconds, with a standard deviation of 22, 33 and 69 milliseconds, respectively. In [37], LTE traffic measurements were collected in a large metropolitan area in the USA. The results show that the median RTT is 70 milliseconds and the 95th percentile is 467 milliseconds.



(a)



(b)

Fig. 7. Server instruction count and CPU cycles.

VI. Evaluation

A. Algorithms and Metrics

We now evaluate the performance of TrafficGrinder using an event-based simulator. The simulator is implemented in Python and extends the one used in [28]. Each machine is represented by a single core CPU and each request r_i takes s_i CPU time to process. Requests are sent by the clients, processed by the LB, and forwarded to the chosen machines. Each experiment runs 30 times and the results are averaged.

We compare the performance of TrafficGrinder to the following two load balancing baseline policies: (a) the least loaded policy (LeastLoaded), which assigns a new connection to the least loaded machine, and (b) the power-of-two-choices policy (PowerOfTwoChoices), which assigns a new connection to the least loaded machine between two randomly chosen machines [38]. In addition, we compare the results to (c) a version of TrafficGrinder that does not support 0-RTT and (d) a SessionAffinity policy, which maximizes the number of connections that benefit from 0-RTT. This latter policy assigns every connection to the same machine to which the previous connection of the same client was assigned and is, therefore, our baseline for session affinity and 0-RTT. We compare TrafficGrinder and the above policies using the following metrics:

- Time-To-First-Byte (TTFB): This is the elapsed time between the client sending the first packet of a connection and receiving the first data packet from the server.
- Flow Completion Time (FCT): This is the elapsed time between the client sending the first packet of a connection and receiving the last packet of the connection.
- Successful 0-RTT: This is the number of connections that benefit from 0-RTT.
- Machine Load: The CPU time it takes to process all the requests in this machine's queue, including those that are being processed. We examine the mean load over the machines. We also examine the load imbalance, defined as the ratio between the maximum load and the mean load [23].

For the evaluation, we use the following datasets:

1) DoH: TCP traffic trace of DNS over HTTPS, obtained from the CESNET2 network [39]. We include only the connections established with Google DNS servers. The trace consists of $45,776$ connections and $2,910$ clients.
2) Synthetic: A synthetic trace, created using a Poisson process, consisting of $9,505$ clients, together generating $100,629$ connections. Inter-arrival time between two consecutive packets is exponentially distributed with a mean of 0.2 milliseconds. The request service time is ranges from 0 to 5 milliseconds uniformly. The trip time in each direction between a client and the cluster is uniformly distributed between 1 and 40 milliseconds. Finally, connection lengths are taken from a Zipf distribution with parameter 1.2.

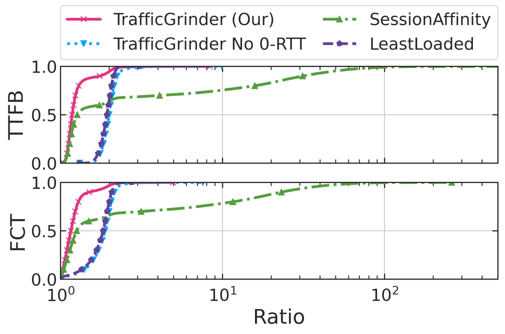
B. The Trade-Off Between 0-RTT and Load Balancing

As discussed in Section V, 0-RTT benefits both the clients and the machines by lowering their latency and load, respectively. When, however, the number of connections that benefit from 0-RTT increases, it is expected that the load balancing between the machines will decrease.

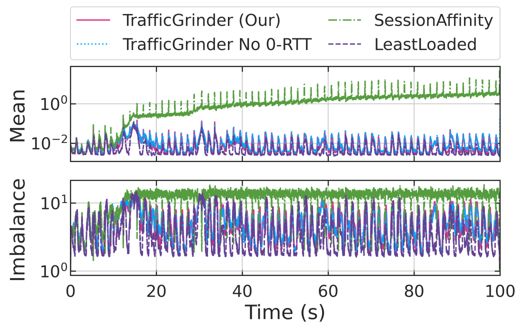
The client's quality of experience is directly affected by the TTFB, which is influenced by two factors: the connection establishment time and the machines' load variance. We normalize the TTFB of each connection by its minimum TTFB, obtained when it is serviced by an empty machine and 0-RTT.

Figures 8a and 8c show the CDF of the normalized TTFB and FCT for Synthetic and DoH, respectively. Higher values indicate better performance. In Fig. 8a, TrafficGrinder shows the best performance among the LeastLoaded and Session-Affinity policies. At the 60th percentile, the normalized TTFB of TrafficGrinder is only 1.2 (20% higher than the minimum TTFB) compared to 1.97 for LeastLoaded, a 64% improvement. For the 90th percentile, TrafficGrinder is 1.74, while LeastLoaded is 2.12. The leap in TrafficGrinder's performance at the 90th percentile is due to the fact that 10% of the connections were not able to benefit from 0-RTT, either because they were the clients' first connection or because their default machine was overloaded. SessionAffinity suffers from the imbalance between the machines and is significantly worse than both TrafficGinder and LeastLoaded. PowerOfTwoChoices demonstrated similar performance to LeastLoaded and is, therefore, not discussed anymore. In Fig. 8c, again TrafficGrinder shows the best performance, with only 15% of unsuccessful 0-RTT connections. The leap between the 80th and 85th percentiles is attributed to the load distribution across the machines.

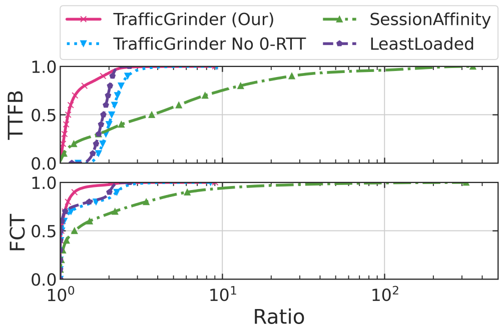
Examining the FCT, 95% of the connections in Synthetic are short (consisting of at most five client packets). Thus, the FCT is dominated by the 1-RTT connection establishment overhead. TrafficGrinder achieves a 58% FCT improvement over LeastLoaded at the 60th percentile. In Fig. 8c, the FCT



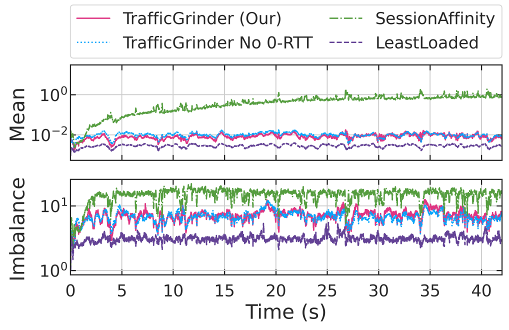
(a) Synthetic - normalized TTFB and FCT CDF



(b) Synthetic - load mean and imbalance across the machines



(c) DoH - normalized TTFB and FCT CDF



(d) DoH - load mean and imbalance across the machines

Fig. 8. Traces results for 64 machines, $d = 20$ in the top-d connections table, $\alpha = 1$ in (1), and $p = 0.8$ in the overload condition in Section III-B.

of TrafficGrinder is better than that of SessionAffinity and close to that of LeastLoaded. The notable performance surge of LeastLoaded beyond the 70th percentile is attributed to the fact that 30% of the connections in the trace are short. Consequently, the lack of 0-RTT support in LeastLoaded significantly impacts the FCT.

Table III shows the mean number of successful 0-RTT connections for the three policies. It also shows the maximum number of successful 0-RTT connections possible, i.e., all the connections that are not their clients' first connection. We distinguish between short and long connections, where short connections are those that have up to five client packets. For the Synthetic trace, TrafficGrinder guarantees 0-RTT to 99.7% of the connections that can use 0-RTT. TrafficGrinder guarantees 0-RTT for 99.9% of the short connections and for 96.6% of the long connections. SessionAffinity guarantees 0-RTT for all long connections, but not for all short connections. The reason is that a client might not be able to benefit from 0-RTT in its second connection if the second connection is established before the first connection is served. For LeastLoaded, only 1.5% of the connections benefit from 0-RTT, which is similar to the performance of uniform random assignment of connections to machines.

Figures 8b and 8d show the mean load and load imbalance metrics across the machines as a function of run time for Synthetic and DoH, respectively. Fig. 8b shows that the load imposed by TrafficGrinder is similar to that of LeastLoaded, while SessionAffinity is far behind. This happens because SessionAffinity assigns heavy hitter clients to the same machines, regardless of the load imposed on these machines. Examining the load imbalance, TrafficGrinder is close to LeastLoaded, whereas the load imbalance of SessionAffinity is considerably larger. Fig. 8d shows the results for the DoH trace. Although the mean load imposed by TrafficGrinder is slightly higher than that of LeastLoaded, the client latency is not affected.

The results of both Synthetic and DoH traces reveal that even with similar load distribution between machines (Fig. 8b, 8d), the absence of 0-RTT significantly degrades performance, particularly for short connections.

We conclude that TrafficGinder demonstrates the good properties of both SessionAffinity and LeastLoaded: the load on the machines is well balanced, as with LeastLoaded, and the percentage of connections that benefit from 0-RTT is very close to that of SessionAffinity.

C. Varying the Number of Machines

In the following experiments, we examine how changing the number of machines affects the FCT. Fig. 9a depicts the normalized FCT as a function of the number of machines. Each policy is represented by a boxplot corresponding to different numbers of machines, ranging from 24 to 88. The same Synthetic trace is used for all machine configurations.

For SessionAffinity, the FCT values decrease as the number of machines increases. This is particularly evident in the median values. The decrease in FCT can be attributed to two primary factors. First, the load per machine decreases as more machines are added, resulting in lower FCT. Second, the distribution of connections by the hash function is more uniform when the number of machines increases. Nevertheless, it is evident that the 95th percentile experiences a gradual decrease.

TABLE III
NUMBER OF SUCCESSFUL 0-RTT CONNECTIONS

Trace	Connection Length	Number of Successful 0-RTT Connections			
		Least-Loaded	Session-Affinity	Traffic-Grinder	Maximum
Synthetic	Short	1344	86099	86148	86234
	Long	77	4890	4725	4890
DoH	Short	257	16539	15533	16630
	Long	392	26010	23438	26236

This is attributed to the fact that when using SessionAffinity, the allocation of a heavy hitter client to a machine results in an increased load, irrespective of the number of machines.

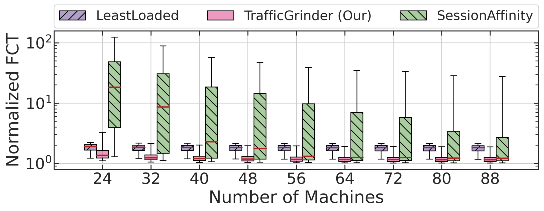
For the LeastLoaded policy, a small reduction in the FCT is observed when the number of machines increases. This is because LeastLoaded already achieves almost minimal FCT using 24 machines, and the overhead from not leveraging 0-RTT grows as the number of machines increases. Once again, TrafficGrinder combines the good properties of both LeastLoaded and SessionAffinity. As machines are added, the FCT decreases, reaching its minimum normalized value of 1. This implies that most of the connections benefit from session affinity and that the load is evenly distributed.

In Fig. 9b, we run an experiment where the load per machine remains the same while the number of machines increases. across varying numbers of machines. This is achieved by controlling the inter-arrival time between packets in the trace. For SessionAffinity, we observe a marginal increase in the median values when more machines are employed, while the 95$^{\text{th}}$ percentile exhibits a more pronounced increase. This is attributed to the fact that in SessionAffinity, increasing the load with additional machines leads to heavy hitter clients imposing a heightened load on the allocated machine. As mentioned above, increasing the number of machines in this policy does not address this issue effectively.

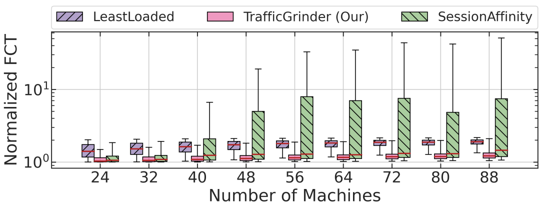
For the LeastLoaded, the FCT increases slightly when more machines are added to the pool. This is attributed to the fact that, with a constant load per machine, the primary contribution to the FCT is the overhead from not using 0-RTT. For TrafficGrinder, there is a subtle increase in the 75$^{\text{th}}$ and 95$^{\text{th}}$ percentiles. This is attributed to the increased load imposed by heavy hitter clients on the machines to which they are allocated, resulting in a slightly higher FCT for connections allocated to the same machine.

VII. RELATED WORK

Stateful LBs usually use a per-connection table for PCC, mapping each connection five-tuple (or fingerprint) to its assigned machine. SilkRoad [40] employs a hardware-based table, while Ananta [41] and Katran [42] use software tables. Duet [43] combines software and hardware with a software per-connection table. In Maglev [23], consistent hashing and a software table are used. Prism [44] uses a software per-connection table and a hardware table for connections whose five-tuple hash changes due to a change in the machine pool. It also has a version that can deal with SYN cookies [45].



(a) Normalized FCT as a function of the number of machines



(b) Normalized FCT as a function of the number of machines, maintaining the same average load per machine

Fig. 9. Synthetic trace results varying the number of machines, $d = 20$ in the top-d connections table, and $\alpha = 1$ in (1), and $p = 0.8$ in the overload condition in Section III-B.

Storing the per-connection state in the LB is not required in QUIC if the machine ID is encoded in the CID [24]. Moreover, the QUIC CONNECTION_CLOSE frame is sent within the encrypted payload [1], preventing the LB from deciding when the connection ends and its state can be removed.

Stateless LBs use hash functions and may use daisy chaining to reduce connection disruption when a machine pool change occurs [46], [47]. Stateful and stateless LB designs were proposed by Cheetah [48]. To ensure PCC, the LB encodes the machine's ID in a cookie and adds it to the packet header. Applying Cheetah to QUIC is briefly discussed in [48].

The IETF proposed a QUIC LB design that employs routable CIDs and facilitates connection migration [24]. In [49], it was shown that CIDs allocated by Facebook and Cloudflare contain repeated patterns and values in specific positions, suggesting that these CIDs are used for encoding data. In addition, it was observed that Google employs CID-aware load balancers. While Facebook QUIC implementation supports encoding routing information in the CID, their LBs still use standard five-tuple load balancing [50].

VIII. CONCLUSION

This work presented TrafficGrinder, a new load balancing scheme for QUIC. The proposed LB ensures 0-RTT, replay resistance, and PFS while maintaining near-optimal load balancing performance. The effectiveness of TrafficGrinder was evaluated through extensive simulations, using both synthetic and real-world traffic traces. We showed that TrafficGrinder improves the TTFB by up to 64%, and the FCT by up to 58%, compared to the LeastLoaded policy. This improvement is attributed to TrafficGrinder's ability to provide 0-RTT for 99.7% of the connections that are not the first between their client and the server. Furthermore, We demonstrated two significant advantages of utilizing 0-RTT over 1-RTT. It halves the time-to-first-byte and reduces the server load by 40%.

References

[1] J. Iyengar and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport," RFC 9000, May 2021. [Online]. Available: https://www.rfc-editor.org/info/rfc9000

[2] M. Thomson and S. Turner, "Using TLS to Secure QUIC," RFC 9001, May 2021. [Online]. Available: https://www.rfc-editor.org/info/rfc9001

[3] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3," RFC 8446, Aug. 2018. [Online]. Available: https://www.rfc-editor.org/info/rfc8446

[4] M. Bishop, "HTTP/3," RFC 9114, Jun. 2022. [Online]. Available: https://www.rfc-editor.org/info/rfc9114

[5] F. Inc., "Facebook/proxygen: A collection of c++ http libraries including an easy to use http server." accessed: September 17, 2023. [Online]. Available: https://github.com/facebook/proxygen

[6] C. G. Günther, "An identity-based key-exchange protocol," in *Advances in Cryptology EUROCRYPT: Workshop on the Theory and Application of Cryptographic Techniques, 1989. Proceedings*, vol. 434. Springer, 2003, pp. 29–37.

[7] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone, *Handbook of applied cryptography*. CRC press, 2018.

[8] F. Günther, B. Hale, T. Jager, and S. Lauer, "0-rtt key exchange with full forward secrecy," in *Advances in Cryptology EUROCRYPT: 36th*. Springer, 2017, pp. 519–548.

[9] F. Dallmeier, J. P. Drees, K. Gellert, T. Handirk, T. Jager, J. Klauke, S. Nachtigall, T. Renzelmann, and R. Wolf, "Forward-secure 0-rtt goes live: implementation and performance analysis in quic," in *CANS Proceedings 19*. Springer, 2020, pp. 211–231.

[10] F. Inc., "Fizz," 2018, accessed: April 10, 2023. [Online]. Available: https://github.com/facebookincubator/fizz

[11] Cloudflare, "cf-tls," 2015, accessed: August 26, 2023. [Online]. Available: https://github.com/cloudflare/cf-tls

[12] M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," in *International Colloquium on Automata, Languages, and Programming*. Springer, 2002, pp. 693–703.

[13] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[14] R. B. Basat, G. Einziger, R. Friedman, and Y. Kassner, "Randomized admission policy for efficient top-k and frequency estimation," in *IEEE INFOCOM*. IEEE, 2017, pp. 1–9.

[15] A. Metwally, D. Agrawal, and A. El Abbadi, "Efficient computation of frequent and top-k elements in data streams," in *International conference on database theory*. Springer, 2005, pp. 398–412.

[16] J. Misra and D. Gries, "Finding repeated elements," *Science of computer programming*, vol. 2, no. 2, pp. 143–152, 1982.

[17] R. Berinde, P. Indyk, G. Cormode, and M. J. Strauss, "Space-optimal heavy hitters with strong error bounds," *ACM TODS*, vol. 35, no. 4, pp. 1–28, 2010.

[18] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, 2010, pp. 267–280.

[19] J. Lu, T. Pan, S. He, M. Miao, G. Zhou, Y. Qi, S. Zhang, E. Song, X. Sun, H. Zhao *et al.*, "Cloudsentry: Two-stage heavy hitter detection for cloud-scale gateway overload protection," *IEEE TPDS*, 2023.

[20] J. Wallerich, H. Dreger, A. Feldmann, B. Krishnamurthy, and W. Willinger, "A methodology for studying persistency aspects of internet flows," *ACM SIGCOMM Computer Communication Review*, vol. 35, no. 2, pp. 23–36, 2005.

[21] B. Turkovic, J. Oostenbrink, F. Kuipers, I. Keslassy, and A. Orda, "Sequential zeroing: Online heavy-hitter detection on programmable hardware," in *IFIP*. IEEE, 2020, pp. 422–430.

[22] M. Thomson, "Version-Independent Properties of QUIC," RFC 8999, May 2021. [Online]. Available: https://www.rfc-editor.org/info/rfc8999

[23] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein, "Maglev: A fast and reliable software network load balancer," in *13th USENIX Symposium NSDI 16*, 2016, pp. 523–535.

[24] M. Duke, N. Banks, and C. Huitema, "QUIC-LB: Generating Routable QUIC Connection IDs," Internet Engineering Task Force, Internet-Draft draft-ietf-quic-load-balancers-17, Aug. 2023, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-quic-load-balancers/17/

[25] J. Daemen and V. Rijmen, "Aes proposal: Rijndael," 1999.

[26] A. Aghdai, C.-Y. Chu, Y. Xu, D. H. Dai, J. Xu, and H. J. Chao, "Spotlight: Scalable transport layer load balancing for data center networks," *IEEE TCC*, vol. 10, no. 3, pp. 2131–2145, 2020.

[27] W. Zhang *et al.*, "Linux virtual server for scalable network services," in *Ottawa Linux Symposium*, vol. 2000, 2000.

[28] Z. Yao, Y. Desmouceaux, J.-A. Cordero-Fuertes, M. Townsley, and T. Clausen, "Hlb: toward load-aware load balancing," *IEEE/ACM Transactions on Networking*, vol. 30, no. 6, pp. 2658–2673, 2022.

[29] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A review of auto-scaling techniques for elastic applications in cloud environments," *Journal of grid computing*, vol. 12, pp. 559–592, 2014.

[30] T. De Matteis and G. Mencagli, "Proactive elasticity and energy awareness in data stream processing," *Journal of Systems and Software*, vol. 127, pp. 302–319, 2017.

[31] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood, "Agile dynamic provisioning of multi-tier internet applications," *ACM TAAS*, vol. 3, no. 1, pp. 1–39, 2008.

[32] D. G. Kendall, "Stochastic processes occurring in the theory of queues and their analysis by the method of the imbedded markov chain," *The Annals of Mathematical Statistics*, pp. 338–354, 1953.

[33] J. F. Kingman, "On queues in heavy traffic," *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 24, no. 2, pp. 383–392, 1962.

[34] J. Little, "A proof of the theorem l= lambdaw," *Operations Research*, vol. 9, no. 3, 1961.

[35] P. Wiki, "Perf wiki." [Online]. Available: https://perf.wiki.kernel.org/index.php/Main_Page

[36] M. Lauridsen, L. C. Gimenez, I. Rodriguez, T. B. Sorensen, and P. Mogensen, "From lte to 5g for connected mobility," *IEEE Communications Magazine*, vol. 55, no. 3, pp. 156–162, 2017.

[37] J. Huang, F. Qian, Y. Guo, Y. Zhou, Q. Xu, Z. M. Mao, S. Sen, and O. Spatscheck, "An in-depth study of lte: Effect of network protocol and application behavior on performance," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 363–374, 2013.

[38] M. Mitzenmacher, "The power of two choices in randomized load balancing," *IEEE TPDS*, vol. 12, no. 10, pp. 1094–1104, 2001.

[39] K. Jeřábek, K. Hynek, T. Čejka, and O. Ryšavý, "Collection of datasets with dns over https traffic," *Data in Brief*, vol. 42, p. 108310, 2022.

[40] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics," in *ACM SIGCOMM*, 2017, pp. 15–28.

[41] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu *et al.*, "Ananta: Cloud scale load balancing," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 207–218, 2013.

[42] F. Inc., "Katran: A high performance layer 4 load balancer," 2019, accessed August 4, 2023. [Online]. Available: https://github.com/facebookincubator/katran

[43] R. Gandhi, H. H. Liu, Y. C. Hu, G. Lu, J. Padhye, L. Yuan, and M. Zhang, "Duet: Cloud scale load balancing with hardware and software," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 27–38, 2014.

[44] R. Cohen, M. Kadosh, A. Lo, and Q. Sayah, "Lb scalability: Achieving the right balance between being stateful and stateless," *IEEE/ACM Transactions on Networking*, vol. 30, no. 1, pp. 382–393, 2021.

[45] ——, "Hardware syn attack protection for high performance load balancers," in *2021 IEEE Symposium on High-Performance Interconnects (HOTI)*. IEEE, 2021, pp. 9–16.

[46] C. Hopps, "Analysis of an Equal-Cost Multi-Path Algorithm," RFC 2992, Nov. 2000. [Online]. Available: https://www.rfc-editor.org/info/rfc2992

[47] J. Zhou, M. Tewari, M. Zhu, A. Kabbani, L. Poutievski, A. Singh, and A. Vahdat, "Wcmp: Weighted cost multipathing for improved fairness in data centers," in *EuroSys*, 2014, pp. 1–14.

[48] T. Barbette, C. Tang, H. Yao, D. Kostić, G. Q. Maguire Jr, P. Papadimitratos, and M. Chiesa, "A high-speed load-balancer design with guaranteed per-connection-consistency," in *17th USENIX Symposium NSDI 20*, 2020, pp. 667–683.

[49] J. Mücke, M. Nawrocki, R. Hiesgen, P. Sattler, J. Zirngibl, G. Carle, T. C. Schmidt, and M. Wählisch, "Waiting for quic: On the opportunities of passive measurements to understand quic deployments," *arXiv preprint arXiv:2209.00965*, 2022.

[50] F. Inc., "mvfst," 2021, accessed: April 10, 2023. [Online]. Available: https://github.com/facebook/mvfst