Cheetah: A High-Speed Programmable Load-Balancer Framework With Guaranteed Per-Connection-Consistency

Tom Barbette, Erfan Wu, Dejan Kostić, Gerald Q. Maguire, Jr., *Life Fellow, IEEE*, Panagiotis Papadimitratos, and Marco Chiesa, *Fellow, IEEE*

***Abstract*— Large service providers use load balancers to dispatch millions of incoming connections per second towards thousands of servers. There are two basic yet critical requirements for a load balancer: *uniform load distribution* of the incoming connections across the servers, which requires to support advanced load balancing mechanisms, and *per-connection-consistency* (PCC), i.e, the ability to map packets belonging to the same connection to the same server even in the presence of changes in the number of active servers and load balancers. Yet, simultaneously meeting these requirements has been an elusive goal. Today's load balancers minimize PCC violations at the price of non-uniform load distribution. This paper presents CHEETAH, a load balancer that supports advanced load balancing mechanisms *and* PCC while being scalable, memory efficient, fast at processing packets, and offers comparable resilience to clogging attacks as with today's load balancers. The CHEETAH LB design guarantees PCC for *any* realizable server selection load balancing mechanism and can be deployed in both stateless and stateful manners, depending on operational needs. We implemented CHEETAH on both a software and a Tofino-based hardware switch. Our evaluation shows that a stateless version of CHEETAH guarantees PCC, has negligible packet processing overheads, and can support load balancing mechanisms that reduce the flow completion time by a factor of $2-3\times$.**

***Index Terms*— Cloud networks, layer 4 load balancing, P4, programmable networks, stateful classification, stateless load balancing, per-connection-consistency, TCP, QUIC.**

I. Introduction

THE vast majority of services deployed in a datacenter need load balancers (LBs) to spread incoming connection requests over the set of servers running these services. As almost half of the traffic in a datacenter must be handled by a LB [2], the inability to uniformly distribute connections across servers has expensive consequences for datacenter and service operators. The most common, yet cost-ineffective, way of dealing with imbalances while meeting stringent Service-Level-Agreements (SLAs) is to over-provision [3].

Existing LBs rely on a simple hash computation on the connection identifier to distribute incoming traffic among the servers [2]–[8]. Recent measurements on Google's production traffic showed that hash-based LBs may suffer from load imbalances of up to 30% [3].

A natural question is: Why do existing LBs not use more sophisticated load balancing mechanisms, e.g, weighted round robin [9], "power of two choices" [10], or least loaded server? The answer lies in the extreme dynamicity of cloud environments. Services and LBs "*must be designed to gracefully withstand traffic surges of hundreds of times their usual loads, as well as DDoS attacks*" [7]. This means that the number of servers and LBs used to provide a service can quickly change over time. Guaranteeing that packets belonging to existing connections are routed to the correct server despite dynamic reconfigurations requires *per-connection-consistency* (PCC) [11] and has been the focus of many previous works [2]–[7], [11] given its challenging design. When only the number of LBs change, hash-based load balancing mechanisms guarantee PCC as packets reach the correct server even when sent to a different LB [5], [7]. To deal with changes in the numbers of servers, existing LBs either store the "connection-to-server" mapping [2]–[4], [11] or let the servers reroute packets that were misrouted [5], [7]. In both cases, a hash function helps mitigate PCC violations, though it cannot completely avoid them (more details in Sect. II). To summarize, existing LBs cannot uniformly distribute connections across the servers as they rely on hash functions to mitigate (but not avoid) PCC violations.

This paper presents the design and evaluation of CHEETAH, a LB with the following properties:

- ***dynamicity***, the number of LBs and servers can increase or decrease depending on the actual load;
- ***per-connection-consistency*** (PCC), packets belonging to the same connection are forwarded to the same server;
- ***uniform load distribution***, by supporting advanced load balancing mechanisms that efficiently utilize the servers;
- ***efficient packet processing***, the LB should have minimal impact on communication latency; and

Manuscript received November 15, 2020; revised June 12, 2021 and August 1, 2021; accepted September 6, 2021; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor P. Giaccone. This work was supported by the European Research Council (ERC) under the European Union's Horizon 2020 Research and Innovation Program under Grant 770889. The work of Panagiotis Papadimitratos was supported in part by KAW Academy Fellowship and in part by SSF Framework Grant (Project SURPRISE). The work of Marco Chiesa was supported by KTH Digital Futures. This paper is an extended version of the work that first appeared with title "A High-Speed Load-Balancer Design with Guaranteed Per-Connection-Consistency" at Usenix NSDI 2020. *(Corresponding author: Marco Chiesa.)*

The authors are with the Department of Computer Science, KTH Royal Institute of Technology, 11428 Stockholm, Sweden (e-mail: barbette@kth.se; mchiesa@kth.se).

This article has supplementary downloadable material available at https://doi.org/10.1109/TNET.2021.3113370, provided by the authors.

Digital Object Identifier 10.1109/TNET.2021.3113370

- ***resilience***, it should be hard for a client to "clog" the LB and the servers with spurious traffic.

CHEETAH takes a different approach than existing LBs, as CHEETAH stores information about the connection mappings into the connections themselves. More specifically, when a CHEETAH LB receives the first packet of a connection, it encodes the selected server's identifier into a *cookie* that is permanently added to all the packet headers exchanged within this connection. Unlike prior work, which relies on hash computations to mitigate PCC violations, the design of CHEETAH completely *decouples* the load balancing logic from PCC support. This in turn allows an operator to guarantee PCC regardless of the "connection-to-server" mapping produced by the chosen load balancing logic. The goal of this paper is not the design of a novel load balancing mechanism for uniformly spreading the load but rather the design of CHEETAH as a building block to support PCC for *any* realizable load balancing mechanisms *without* violating PCC. There are *two challenges* that need to be addressed to leverage cookies in an LB. First, we cannot expose the identity of a server to the external users. To make CHEETAH resilient to attacks, we generate "opaque" cookies that can be processed fast and can only be interpreted by the LB; therefore, server identifiers are never exposed to the user, thus thwarting resource exhaustion and selective targeting of servers. The second challenge is how to support the semantic of stateful LBs, which may still be needed for storing statistics about the incoming connections, in an efficient way that is amenable to implementation on high-speed networking switches (e.g, Tofino switch [12]).

We present two different implementations of CHEETAH, a stateless and a stateful version. Our stateless and stateful CHEETAH LBs carefully encode the connection-to-servers mappings into the packet headers so as to guarantee levels of resilience that are no worse (and in some cases even stronger) than existing stateless and stateful LBs, respectively. For instance, our stateful LB increases resilience by utilizing a novel and fast stack-based mechanism that simplifies the operation of today's cuckoo-hash-based stateful LBs, which suffer from slow insertion times.

In summary, our contributions are:

- We quantify limitations of existing stateless and stateful LBs through large-scale simulations. We show that the quality of the load distribution of existing LBs is 40 times worse than that of an ideal LB. We also show stateless LBs (such as Beamer and Faild) can reduce such imbalances at the price of increasing PCC violations.
- We introduce CHEETAH, an LB that guarantees PCC for any realizable load balancing mechanisms. We present a stateless and a stateful design of CHEETAH, which make different trade-offs in terms of resilience and performance.
- We implement our stateless and stateful CHEETAH LBs in FastClick [13] and compare their performance with state-of-the-art stateless and stateful LBs, respectively. We also implement both versions of CHEETAH with a weighted round-robin LB on a Tofino-based switch [12].
- In our experiments, we show the potential benefits of CHEETAH with a non-hash-based load balancing mechanism. The number of processor cycles per packet for *both* our stateless and stateful implementation of CHEETAH is comparable to existing stateless implementations and 3.5x fewer cycles per packet than existing stateful LBs.

II. BACKGROUND AND MOTIVATION

Internet organizations deploy large-scale applications using clusters of servers located within one or more datacenters (DCs). We provide a brief background on DC LBs, discuss related work, and show limitations of the existing schemes. However, we do not discuss geo-distributed load balancing across DCs or network-level DC load balancers [14]–[21], whose goal is to load balance the traffic within the DC network and do not deal with per-connection-consistency problems. Further, we distinguish between *stateless* LBs, which do not store per-connection state, and *stateful* LBs, which store information about ongoing connections.

Multi-tier load balancing architectures. Datacenter operators assign a *Virtual IP* (VIP) address to each operated service. Each VIP in a DC is associated with a set of servers providing that service. Each server has a *Direct IP* (DIP) address that uniquely identifies the server within the DC.

A LB inside the DC receives incoming connections for a certain VIP and selects a server to provide the requested service. Each connection is a Layer 4 connection (typically TCP or QUIC). For each VIP, a LB partitions the space of the connection identifiers (e.g, TCP 5-tuples) across all the servers (i.e, DIPs) associated with that VIP. The partitioning function is stored in the LB and is used to retrieve the correct DIP for each incoming packet.

A large-scale DC may have tens of thousands of servers and hundreds of LBs [3], [6], [11]. These LBs are often arranged into different tiers (see Fig. 1). The 1st-tier of LBs are faster and less complex than those in subsequent tiers. For example, a typical DC would use BGP routers using ECMP forwarding at the 1st-tier, followed by Layer 4 LBs, in turn followed by Layer 7 LBs and applications [4]. Similar to prior work on DC load balancing, we consider Layer 7 LBs to be at the same level as the services [2]–[4]. Any 1st tier LB receiving a packet directed to a VIP, performs a look up to fetch the *set* of 2nd tier LBs responsible for that VIP. It then forwards the packet towards any of these LBs. The main goal of the 1st-tier is demultiplexing the incoming traffic at the per VIP level towards their dedicated 2nd tier LBs. The 2nd-tier LBs perform two crucial operations: *(i)* guaranteeing (PCC) [11] and *(ii) load balancing* the incoming connections.

A. Limits of Stateless Load Balancers

Traditional stateless LBs cannot guarantee PCC. A stateless LB partitions the space of connection identifiers among the set of servers. The partitioning function is stored in the LB and does not depend on the number of active connections. Most stateless LBs, e.g, ECMP [22], [23] & WCMP [8], store this partitioning in the form of an *indirection table*, which maps the output of a hash function modulo the size of the table to a specific server [5], [7], [8], [22]. A *uniform hash* scheme maps each server to an equal number of entries in the indirection table. When a LB receives a packet, it extracts the connection identifier from the packet and feeds

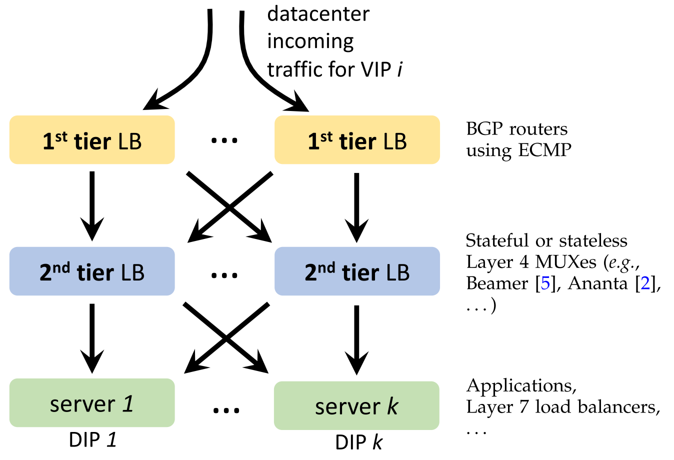

datacenter incoming traffic for VIP *i*
1st tier LB
...
1st tier LB
BGP routers using ECMP
2nd tier LB
...
2nd tier LB
Stateful or stateless Layer 4 MUXes (*e.g.*, Beamer [5], Ananta [2], ...)
server *1*
...
server *k*
DIP *1*
DIP *k*
Applications, Layer 7 load balancers, ...

Fig. 1. Traditional datacenter load balancing architecture.

it as input to a hash function. The output of the hash modulo the size of the indirection table determines the index of the entry in the table where the LB can find to which server the packet should be forwarded. If the number of servers changes, the indirection table must be updated, which may cause some *existing* connections to be rerouted to the new (and incorrect) server that is currently associated with an entry in the table, resulting in a PCC violation.

Advanced stateless LBs cannot always guarantee PCC. Beamer [5] and Faild [7] introduced *daisy-chaining* to tackle PCC. They encapsulate in the header of the packet the address of a "backup" server to which a packet should be sent when the LB hits the wrong server. This backup server is selected as the last server that was assigned to a given entry in the indirection table *before* the entry was remapped. PCC violations are prevented as long as one does not perform two reconfigurations that change the same entry in the table twice (as only one backup server can be stored in the packet) **and** one can *simultaneously* reconfigure all the LBs (see [5] for an example).

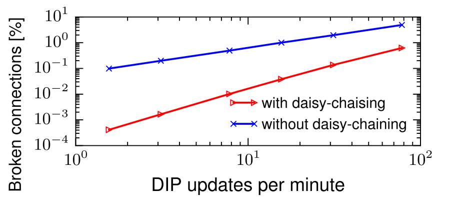
Fig. 2a shows the percentage of broken connections (i.e, PCC violations) with and without daisy chaining in our large-scale simulations. We used the same parameters, traffic workloads, and cluster reconfiguration events derived from previous work on real-world DC load balancing, i.e, SilkRoad [11]. Namely, we simulated a cluster of 468 servers and we generated a workload using the same traffic distribution of a large web server service. We performed *DIP updates*, i.eremoval or additional of servers from the cluster, using different frequency distributions. SilkRoad reports that 95% of their clusters experience between 1.5 and 80 DIP updates/minute and provide distributions for the update time. We define the number of *broken connections* as the number of connections that have been mapped to *at least two* different servers during their starting and ending times. Fig. 2a shows that Beamer and Faild (plotted using the same line) still break almost 1% of the connections at the highest DIP update frequency, which may lead to an unacceptable level of service level agreement (SLA) violations [11].

Hash-based LBs cannot uniformly spread the load. We now investigate the ability of different load balancing mechanisms to uniformly spread the load across the servers for a single VIP. Similar to the Google Maglev work [3], we define the *imbalance* of a server as the ratio between the number of connections active on that server and the average number of active connections across all servers. We also define the *system imbalance* as the maximum imbalance of any server. The imbalance of a simulation run is the *average* imbalance of the system during the entire duration of the simulation. We discuss different load metrics in Sect. IV. Using the same simulation settings as described above, we compare *(i)* Beamer [5]/ Faild [7], which use a uniform hash, *(ii)* Round-Robin [24], which assigns each new connection to the next server in a list, *(iii)* Power-Of-Two [10], which picks the least loaded among two random servers, and *(iv)* Least-Loaded [24], which assigns each new connection to the server with the fewest active connections. We note that Round-Robin, Power-Of-Two, and Least-Loaded require storing the connection-to-server mapping, hence they cannot be supported by Beamer/Faild. In this simulation, we do not change the size of the cluster but rather vary only the number of connections that are active at the same time in the cluster between 20K and 200K. We choose this range of active connections to induce the same imbalances (15%-30%) observed for uniform hashes in Google Maglev [3]. Fig. 2b shows the results of our simulations. Round-Robin outperforms a Beamer-like LB by a factor of $1.2\times$. When comparing these schemes with Power-Of-Two and Least-Loaded, we observe a reduction in imbalance by a factor of $10\times$ and $40\times$, respectively. These results show that a more uniform load distribution can be achieved by storing the mapping between connections and servers, though one still has to support PCC when the LB pool size changes. We note that today's stateful LBs [2]–[4], [11] rely on different variations of uniform-hash, thus suffer from imbalances similarly to Beamer.

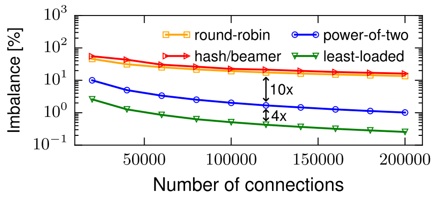
Beamer can reduce imbalance at the cost of a greater number of PCC violations. We tried to reduce the imbalances in Beamer by monitoring the server load imbalances and modify the entries in the indirection table accordingly. We extended Beamer with a dynamic mechanism that gets as input an imbalance threshold and remove a server from the indirection table whenever its load is above this threshold. The server is re-added to the table when its number of active connections drops below the average. Note that, if an entry in the indirection table changes its server mapping twice, Beamer will break those existing connections that were relying on the initial state of the indirection table. Fig. 2c shows the percentage of broken connections for increasing imbalance thresholds. We set the number of active connections to 70K (corresponding to an *average* 30% imbalance in Fig. 2b). We note that guaranteeing an imbalance of at most 10% would cause 3% of all connections to break. Even with an imbalance threshold of 40% one would still observe 0.1% broken connections because of micro-bursts. Hence, even this extended Beamer cannot guarantee PCC and uniform load balancing at the same time.

B. Limits of Stateful Load Balancers

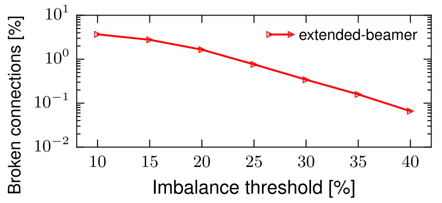
Stateful LBs store the connection-to-server mapping in a so-called `ConnTable` for two main reasons: *(i)* to preserve



(a) Impact of daisy-chained during DIP updates.



(b) Load imbalance.



(c) Dynamic Beamer imbalance.

Fig. 2. Analysis of PCC-violations and load imbalances of state-of-the-art load balancers. To ease visibility, points are connected with straight lines along the x-axis.

PCC when the number of servers changes and *(ii)* to enable fine-grain visibility into the flows.

Today's stateful LBs cannot guarantee PCC. Consider Fig. 1 and the case in which we add an additional stateful LB for a certain VIP. The BGP routers, which rely on ECMP, will reroute some connections to a LB than does not have the state for that connection. Thus, this LB does not know to which server the packet should be forwarded unless all LBs use an identical hash-based mechanism (and therefore experience imbalances). Therefore, existing LBs (including Facebook Katran [4], Google Maglev [3], and Microsoft Ananta [2]) rely on hashing mechanisms to mitigate PCC violations. However, this is not enough if the number of servers also changes, then some existing connections will be routed to an LB without state, hence it will hash the connection to the wrong server, thus breaking PCC.

Today's stateful LBs rely on complex and slow data structures. State-of-the-art LBs rely on cuckoo-hash tables [25] to keep per-connection mappings. These data structures guarantee constant time lookups but may require non-constant insertion time [26]. These slow insertions may severely impact the LB's throughput, e.g, a throughput loss by 2x has been observed on OpenFlow switches when performing ~ 60 updates/second [27]. Finally, SYN-flood attacks are more effective with slow insertions at the LB.

We refer the reader to our supplementary online-only material (Appendix C) for an extensive discussion of these limitations.

C. Service Resilience and Load Balancers

Load balancers are an indispensable component against Distributed Denial of Service (DDoS) attacks, e.g, bandwidth depletion at the server and memory exhaustion at the LB. Dealing with such attacks is a multi-faceted problem involving multiple entities of the network infrastructure [28], e.g, firewalls, intrusion detection, application gateways. This paper does not focus on how the LB fits into this picture but rather studies the resilience of the LB itself and the resilience that its design provides to the service operation.

LBs shield servers from targeted bandwidth depletion attacks. An LB system should be able to absorb sudden bursts due to DDoS attacks with minimal impact on a service's operation. Today's LB mechanisms rely on hash-based load balancing mechanisms to provide a first pro-active level of defense, which consists in spreading connections across all servers. As long as an attacker does not reverse engineer the hash function, multiple malicious connections will be spread over the servers. A system should not allow clients to target specific servers with spurious traffic.

Stateful LBs support per-connection view at lower resilience. Stateful LBs provide fine-grained visibility into the active connections, providing resilience to the service operation, e.g, by selectively rerouting DDoS flows. At the same time, stateful LBs are a trivial target of resource depletion clogging DDoS attacks: incoming spurious connections add to the connection table rapidly exhaust the limited LB memory (e.g, [5]) or grow the connection table aggressively, rapidly degrading performance even with ample memory [27]. Stateless LBs can inherently withstand clogging DDoS, sustaining much higher throughput, but can only offer per-server statistics visibility to the service operation.

Having analyzed the above limitations of today's load balancers, we conclude by asking the following question: *"Can we design a load balancing system that guarantees PCC, supports any realizable load balancing mechanism, and achieves similar levels of resiliency of today's state-of-the-art LBs?"*

III. The Cheetah Load Balancer

In this section, we present Cheetah, a load balancing system that supports arbitrary load balancing mechanisms and guarantees PCC without sacrificing performance. Cheetah solves many of today's load balancing problems by encoding information about the connection into a *cookie* that is added to all the packets of a connection. Cheetah sets the cookie according to any chosen and realizable load balancing mechanism and relies on that cookie to *(i)* guarantee future packets belonging to the same connection are forwarded to the same server and *(ii)* speed up the forwarding process in a stateful LB, which increases the resilience of the LB. Understanding what information should be encoded into the cookie, how to encode it, and how to use this information inside a stateless or stateful LB is the goal of this section. We first introduce the stateless Cheetah LB, which guarantees PCC and preserves the same resilience and packet processing performance of existing stateless LBs. We then introduce the stateful Cheetah LB, which improves the packet processing performance of today's stateful LBs, and present an LB architecture that strikes different tradeoffs in terms of performance and resilience. We stress the fact that Cheetah is not a new LB mechanism but rather a building block for deploying arbitrary LB mechanisms without breaking PCC (we show several implemented LB mechanisms in Sect. IV).

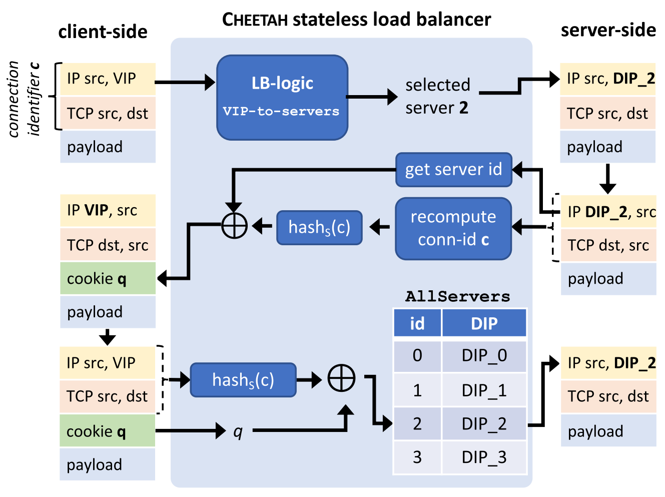


Fig. 3. CHEETAH stateless LB operations.

A naïve approach. We first discuss a straightforward approach to guarantee PCC that would not work in practice because of its poor resiliency. It entails storing the identifier of a server (i.e, the DIP) in the cookie of a connection. In this way, an LB can easily preserve PCC by extracting the cookie from each subsequent incoming packet. We note that such naïve approaches are reminiscent of several previous proposals on multi path transport protocols [29], [30], where the identifiers of the servers are explicitly communicated to the clients when establishing multiple subflows within a connection. There is at least one critical resiliency issue with this approach. Some clients can wait to establish many connections to the same server and then suddenly increase their load. This is highly undesired as it leads to cascade-effect imbalances and service disruptions [31].

A. Stateless CHEETAH *LB*

The stateless CHEETAH LB: encoding an opaque offset into the cookie. We now discuss how we overcome the above issues in CHEETAH. We aim to achieve the same resiliency levels of today's production-ready stateless LBs (e.g, Faild [7]/ Beamer [5], [31]) while supporting arbitrary load balancing mechanisms and guaranteeing PCC. We assume a single tier LB architecture and defer the discussion of multi-tier architectures to later in this section. In this subsection, we assume the LB both adds and removes a cookie from a packet without any help from the servers.

The CHEETAH stateless LB keeps two different types of tables (see Fig. 3): an `AllServers` table that maps a server identifier to the DIP of the server and a `VIPToServers` table that maps each VIP to the set of servers running that VIP. The `AllServers` table is mostly static as it contains an entry for each server in the DC network. Only when servers are deployed in/removed from the DC is the `AllServers` table updated. The `VIPToServers` table is modified when the number of servers running a certain service increases/ decreases, a more common operation to deal with changes in the VIP current demands.

When the LB receives the first packet of a connection (top part of Fig. 3), it extracts the set of servers running the service (i.e, with a given VIP) from the `VIPToServers` table, selects one of the servers according to any pre-configured load balancing mechanism (e.ground-robin), and forwards the packet. For every packet received from a server (middle part of Fig. 3), the LB encodes an "opaque" identifier of the server mapping into the cookie for this connection. To do so, CHEETAH computes the hash of the connection identifier with a salt S (unknown to the clients), XORs it with the identifier of the server, and adds the output of the XOR to the packet header as the cookie. The salt S is the same for all connections. When the LB receives any subsequent packet belonging to this connection (bottom part of Fig. 3), it extracts the cookie from the packet header, computes the hash of the connection identifier with the salt S, XORs the output of the hash with the cookie, uses the output of the XOR as the identifier of the server, and retrieves the DIP of the server from the `AllServers` table.

Stateless CHEETAH guarantees PCC. CHEETAH relies on two main design ideas to avoid breaking connections: *(i)* moving the state needed to preserve the mapping between a connection and its server into the packet header of the connection and *(ii)* using the more dynamic `VIPToServers` table only for the 1st packet of a connection. Subsequently, the static `AllServers` table is used to forward packets belonging to any existing connection. This trivially guarantees PCC.

Compared to existing stateless LBs Stateless CHEETAH achieves comparable resiliency. Binding the cookie with the hash of the connection identifier brings one main advantage compared to the earlier naïve scheme, as an attacker must first reverse engineer the hash function of the LB in order to launch an attack targeting a specific server. This is similar to existing stateless LBs, which rely on non-cryptographically secure (yet efficient) hash functions such as CRC. This makes CHEETAH as resilient as other production-ready stateless LBs. We note that CHEETAH is orthogonal to DDoS mitigation defence mechanisms, especially when deployed in reactive mode. We further discuss CHEETAH resilience, including support for multi path protocols, in Sect. VI.

Stateless CHEETAH supports arbitrary load balancing mechanisms. All the reviewed state-of-the-art LBs (even stateful ones) are restricted to uniform hashing when it comes to load balancing mechanisms — as any other mechanism would break an unacceptable number of connections when the number of servers/LBs changes. In contrast, whenever a new connection arrives at a stateless CHEETAH LB, CHEETAH selects a server among those returned from a lookup in the `VIPToServers` table. The selected server may depend upon the specific load balancing mechanism configured by the service's operator. We note that the selection of the server may or may not be implementable in the data-plane. The CHEETAH LB guarantees that once the mapping connection-to-server has been established by the LB logic (not necessarily at the data-plane speed), all the subsequent packets belonging that that connection will be routed to the selected server. Since the binding of the connection to the server is stored in the packet header, CHEETAH can support LB mechanisms that go well beyond uniform hashing. For instance, an operator may decide to rely on "power of two choices" [10], which is renowned to decrease load imbalances. Another service operator may prefer a weighted round-robin load balancing mechanism that uses some periodically reported metrics (e.g, CPU utilization) to spread the load among the servers.

Lower bounds on the size of the cookie. In CHEETAH, the size of the cookie has to be at least $\log_2 k$ bits, where k is the maximum number of servers stored in the `AllServers` table. Therefore, the size of the cookie grows logarithmically in the size of the number of servers. One question is whether PCC can be guaranteed using a cookie whose size is smaller than $\log_2(k)$ and the memory size of the LB is constant.

Theorem 1: Given an arbitrarily large number of connections, any load balancer using $O(1)$ memory requires cookies of size $\Omega(\log(k))$ to guarantee PCC under any possible change in the number of active servers, where k is the overall number of servers in the DC that can be assigned to the service with a given VIP.

Proof Sketch: We prove the statement of the theorem in the widely adopted Kolmogorov descriptive complexity model [32]. We leverage similar techniques used in the past to demonstrate a variety of memory-related lower bounds for shortest-path routing problems [33].

Let R be the set of all the possible connection identifiers. Let C be the set of all possible cookies. Let $S = \{s_1, \ldots, s_k\}$ be the set of servers. We assume $|R| \gg k$, which is the most interesting case in real-world datacenters. Suppose, by contradiction, that there exists an LB which uses $O(1)$ memory with cookies of size smaller than $\log(k)$ bits that guarantees PCC under any arbitrary number of changes in the subset of active servers $A \subseteq S$. For any possible set of active servers A, the LB maps a new incoming connection identifier $r \in R$ to a server $s \in S$ using an arbitrarily function $f : R \times 2^S \rightarrow S$.

Let us now restrict our focus to the k distinct sets of active servers in which only a single server is active, i.e, $A_i{=}\{s_i\}$, for $i{=}\{1, \ldots, k\}$. At any time instant, only one of these servers is active. The non-active servers continue serving the previously established connections, which should always be routed to the correct server. Depending on the time instant when a connection $r \in R$ arrives at the LB, the connection may be mapped to one of these currently active servers. Therefore, the LB must be able to distinguish among $|R| \times k$ distinct possible mappings between connections and servers. Consider a cookie with l bits, where $l < \log(k)$. This information allows us to distinguish among $|R| \times l$ possible mappings, leaving $|R| \times (\log k - l) > |R|$ mappings to the LB memory. This is a contradiction as we assumed the LB uses a memory of $O(1)$. □

It is trivial to verify that the above theorem holds even if one wants to implement an advanced LB mechanism, e.ground-robin, least-loaded, even in the absence of changes in the set of active servers.

While the above results close the doors to any sublogarithmic overhead in the packet header; in practice, operators may decide to trade some PCC violations and load imbalances for a smaller sized cookie. We leave the design of such LBs as future work.

B. Stateful CHEETAH LB

We also designed a stateful version of CHEETAH to support a finer level of visibility into the flows than that offered by

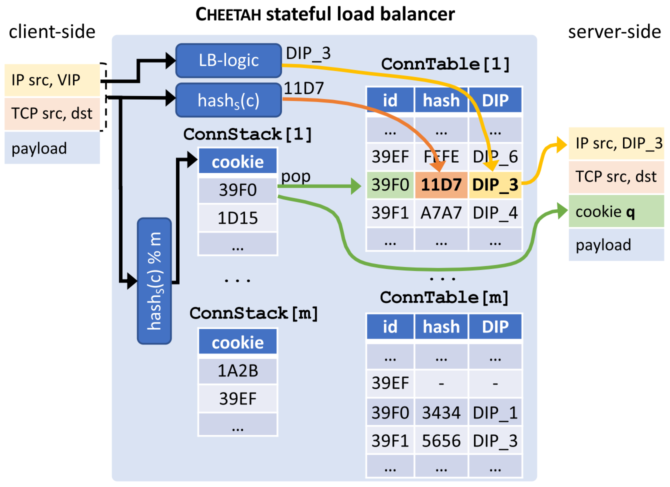


Fig. 4. CHEETAH stateful LB operations for the 1st packet of a connection. We omit both the `VIP-to-servers`, which is part of the LB-logic, and the stateless cookie for identifying the stateful LB. The response packet does not traverse the load balancers. Subsequent packets from the user access their index in the correspoding `ConnTable`.

stateless LBs. A stateful LB can keep track of the behaviour of each individual connection and support complex network functions, such as rate limiters, NATs, detection of heavy-hitters, and rerouting to dedicated scrubbing devices (as in the case of Microsoft Ananta [2] and CloudFlare [28]). In contrast to existing LBs, our stateful LB guarantees PCC (inherited from the stateless design) and uses a more performant `ConnTable` that supports *constant time* insertions/deletions and is amenable to fast data plane implementations. In the following text we say that PCC is guaranteed if a packet is routed to the correct server as long as an LB having state for its connection exists. We also assume that the LB adds the cookie into the packets received by the users but it does not remove it, i.ethe servers must be aware of the cookie. It is however possible to also handle the cookies entirely at the LB as shown in our GitHub repository.

The stateful CHEETAH load balancer: encoding table indices in the packet header. As discussed in Sect. II, today's stateful LBs rely on advanced hash tables, e.g, cuckoo-hashing [25], to store per-connection state at the LB [11]. Such data structures offer constant-time data-plane lookups but insertion/modification of any entry in the table requires intervention of the slower control plane or complex & workload-dependant data structures (e.g, Bloom filters [11], Stash-based data structures [26]), which are both complex and hard to tune for a specific workload.

We make a simple yet powerful observation about stateful tables that any insertion, modification, or deletion of an entry in a table can be greatly simplified if a packet carries information about the index of the entry in the table where its connection is stored. Since datacenters may have tens of billions of active connections, we need to devise a stateful approach where the size of the cookie is explicitly given as input. In a stateful CHEETAH LB (see Fig. 4), we store a set of m `ConnTable` tables that keep per-connection statistics and DIP mappings. We also use an equal number of `ConnStack` stacks of indices, each storing the unused entries in its corresponding `ConnTable`.

For the sake of simplicity, we first assume there is only one LB and one `ConnTable` with its associated `ConnStack`, i.e, $m = 1$. Whenever a new connection state needs to be installed, CHEETAH pops an index from `ConnStack` and incorporates it as part of the cookie in the packet's header. It also stores the selected server and the hash of the connection identifier with a salt S into the corresponding table entry. This hash value allows the LB to filter out malicious attempts to interfere with legitimate traffic flows, similarly to SilkRoad [11]. Whenever a packet belonging to an existing connection arrives at the LB, CHEETAH extracts the index from the cookie and uses it to quickly perform a lookup only in the `ConnTable`. Note that insertion, modification, and deletion of connections can be performed in constant time entirely in the data plane. We explain details of the implementation in Sect. IV.

The number of connections that we can store within a single `ConnTable` is equal to 2^{r}, where r is the size of the cookie. In practice, the size of the cookie may limit the number of connections that can be stored in the LB. We therefore present a hybrid approach that uses a hash function to partition the space of the connection identifiers into m partitions. As for any stateful table, m should be chosen high enough so the total number of entries $m * 2^{\mathrm{r}}$ is suitable. The same cookie can be re-used among connections belonging to distinct partitions.

A hybrid datacenter architecture. Stateful LBs are typically not deployed at the edge of the datacenter for two reasons: they are more complex and slower compared to stateless LBs. As such, they are a weak point that could compromise the entire LB availability. Therefore, we propose a 2-tier DC architecture where the first tier consists of stateless CHEETAH LBs and the second tier consists of stateful CHEETAH LBs. The stateless LB uses the first bytes of the cookie to encode the identifier of a stateful load balancer, thus guaranteeing a connection always reaches the same LB regardless of the LB pool size. The stateful load balancer uses the last bytes of the cookie to encode per-connection information as described above.

C. Deployment Scenarios

There are three main dimensions of the implementation space: (i) server modifications vs no modifications, (ii) stateless vs stateful, and (iii) hardware vs software implementation. By allowing modifications to the servers, we can support Direct Server Return (DSR), which means that the servers insert the cookie and packets do not have to traverse any load balancer on their way back to the users. If we cannot modify the servers, we only need to consider stateless CHEETAH. In this case traffic from the servers can hit any of the load balancers, which will then add the cookie.

We now discuss bandwidth requirements for the different design scenarios with hardware and software LBs. A software LB sees traffic in both directions without DSR and in one direction with DSR. A hardware LB could instead replace the ingress switch/router of the datacenter. In this case, traffic from/to the users has anyway to traverse the switch in both directions. Therefore, a hardware implementation that is deployed "on-path" removes any bandwidth overheads due to software LB implementations. We note that some server selection mechanisms may not be *realizable* in hardware. It depends on the capabilities of the device. For instance, while it is easy to implement Weighted-Round-Robin in P4, sorting an array of server loads is more cumbersome. With software implementations, such a problem does not exist.

IV. IMPLEMENTATION

The simplicity of our design makes CHEETAH amenable to highly efficient implementations in the data-plane. We implemented stateful and stateless CHEETAH LBs on FastClick [13], a faster version of the Click Modular Router [34] that supports DPDK [35] and multi-processing. Previous stateless systems, such as Beamer [5], have also relied on FastClick for their software-based implementation. We also implemented stateless and stateful versions of the CHEETAH LB with a weighted round-robin LB on a Tofino-based switch using P4 [12]. Both implementations are available at [36]. We first discuss the critical question of where to actually store the cookie in today's protocols and then describe the FastClick and Tofino implementations.

Preserving legacy-compatibility. Our goal is to limit the amount of modifications needed to deploy CHEETAH on existing devices. Ideally, we would like to use a dedicated TCP option for storing the CHEETAH cookie into the packet header of all packets in a connection. However, this would require modifications to the clients, which would be infeasible in practice. We therefore identified three possible ways to implement cookies within existing transport protocols *without* requiring any modifications to the clients' machines: *(i)* incorporate the cookie into the `connection-id` of QUIC connections, *(ii)* encode the cookie into the least significant bits of IPv6 addresses and use IPv6 mobility support to rebind the host's address (the LB acts as a home agent), and *(iii)* embed the cookie into part of the bits of the TCP timestamp options. In this paper, we implemented a proof-of-concept CHEETAH using the TCP timestamp option[1] as explained in Appendix A (see online-only supplementary material) and a proof-of-concept CHEETAH using QUIC connection IDs (Sect. IV-D). We note that a QUIC implementation is easier and more performant since parsing TCP options is an expensive operation in both software and hardware LBs. As for TCP timestamps, we note that similar encodings of information into the TCP timestamp have been proposed in the past but require modifications to the servers [30]. The stateless CHEETAH LB can transparently translates the server timestamps with the encoded timestamps without interfering with TCP timestamp related mechanisms (i.e, RTT estimation and protections against wrapped sequences [37]). Therefore, no modifications are required to the servers for stateless mode unless the datacenter operator wants to guarantee Direct Server Return (DSR), i.e, packets from the servers to the client do not traverse any load balancer. In that case, the server must encode the cookie into the timestamp itself. The cookie must also be sent back by the server for stateful mode, as the load balancer would not be able to find the stack index for returning traffic. Server modifications are described

[1]We verified in Appendix A in the online-only supplementary material that the latest Android, iOS, Ubuntu, and MacOS operating systems support TCP timestamp options but not Windows.

in Appendix A.2 (see online-only supplementary material). We leave the implementation of CHEETAH on IPv6 as future work. Finally, we also explored Segment Routing (SR) [38] as a source routing protocol for adding both the VIP and an encoded cookie. One of the problems with SR is that we could not find a way for the server to force the client to add the CHEETAH cookie back to the server.

A. Analysis of TCP Timestamp Usage

Today's servers use TCP timestamp granularity $\geq$ 1ms. We ran a comprehensive set of measurements to determine the granularity of the TCP timestamp unit utilized by the largest service providers according to the Alexa Top100 ranking [39]. We downloaded large files from each the top 15 ranked web sites and extracted both the TCP timestamp TS_{val} options and the client side timestamp. We then computed the difference between the TCP last and first timestamps and divided this amount by the difference between the client measured last and first (non-TCP) timestamps. The result is the granularity of the server-side TCP timestamp unit. We report the results in Table II. All the service providers using TCP timestamps have a granularity of at least 1ms so the timestamp would wrap every $2^{16} \approx 65$ seconds when using CHEETAH to support these services. This means a connection handled by CHEETAH should be sending packets that are not spaced apart more than 65 seconds, e.g, using a keep-alive.

TCP timestamps are mostly supported in today's OSes. We ran a small experiment to verify whether today's client devices support the echoing of TCP timestamp options back to the servers. We tested the latest OSes available in both recent smartphones and desktop PCs: Google Android 9, iOS 13, Ubuntu 18.04, Microsoft Windows 10, and MacOS 10.14. We observed that all except Microsoft Windows correctly negotiate and echo the TCP timestamp option when the server requires to use it. Based on some recent measurements, more than 98% of the smartphone and tablet devices are either using Android or iOS [40]. Smartphone devices are the most common type of devices, representing 53% of all devices [41]. For desktop devices, Windows is the predominant OS with over 75% of the desktop share whereas MacOS represent a 16% of this share [42]. For Windows desktop devices, a cloud operator can either encode the cookie in the QUIC header (69% of the Windows users use Google Chrome, compatible with QUIC [43]), IPv6 address, or use a traditional stateful LB for these devices.

B. FastClick Implementation Using TCP Timestamps

The FastClick implementation is a fully-fledged implementation of CHEETAH that supports L2 & L3 load balancing and multiple load balancing mechanisms (e.g, round-robin, power-of-two choices, least-loaded server). The LB supports different load metrics including number of active connection and CPU utilization. The LB decodes cookies for both stateless and stateful modes using the TCP timestamp as described above, and can optionally fix the timestamp in-place if the server is not modified to do it.

Parsing TCP options. Each TS option has a 1-byte identifier, 1-byte length, and then the content value. Options may appear in any order. This makes extracting a specific option a non-trivial operation [29]. We focus on extracting the timestamp option TS_{ecr} from a packet. To accelerate this parsing operation, we performed a statistical study over 798M packets headers from traffic captured on our campus.

TABLE I
TCP OPTIONS PATTERN

SYN packets	
MSS SAckOK Timestamp [NOP WScale]	49.86%
MSS NOP WScale NOP NOP Timestamp [SAckOK EOL]	44.49%
MSS NOP WScale SAckOK Timestamp	4.53%
Slow path	1,12%
SYN-ACK packets	
MSS SAckOK Timestamp [NOP WScale]	76.85%
MSS NOP WScale SAckOK Timestamp	18.79%
MSS NOP NOP Timestamp [SAckOK EOL]	1.69%
MSS NOP WScale NOP NOP Timestamp [SAckOK EOL]	1.55%
Slow path	1,12%
Other packets	
NOP NOP Timestamp	98.46%
NOP NOP Timestamp [NOP NOP SAck]	1.49%
Slow path	0,05%

Table I shows the most common patterns observed across the entire trace for packets containing the timestamp option. The Linux Kernel already implements a similar fast parsing technique for non-SYN(/ACK) packets. We first consider non-SYN packets (i.e, "Other packets" in the table). Our study shows that 99.95% of the packets have the following pattern: NOP (1B) + NOP (1B) + TimeStamp (10B) possibly followed by other fields. When a packet arrives, we can easily determine whether it matches this pattern by performing a simple 32-bit comparison and checking that the first two bytes are NOP identiers and the third one is the Timestamp id. We process the remaining 0.05% of the traffic in the slow path. We now look at SYN packets. Consider the first row in the table, i.e, MSS (4B) + SAckOK (2B) + TimeStamp (10B) + SAck + EOL. To verify if a packet matches this pattern, we perform a 64-bit wildcard comparison and check that the first byte is the MSS id, the fifth byte is the SAckOK id, and the seventh byte is the TimeStamp id. We can apply similar techniques for the remaining patterns matchable with 64 bits. Some types of hosts generate packets whose patterns are wider than 64 bits, which is the limit of our x86_64 machine. We then rely on one SSE 128bit integer wildcard comparison to verify such patterns. The remaining 2.24% of patterns are handled through a standard hop-by-hop parsing following the TCP options Type-Length-Value chain. Finally, we note that we can completely avoid the more complex parsing operations for SYNs and SYN/ACKs if servers use TCP SYN cookies [44] (see Appendix A in the online-only supplementary material for more details).

Load balancing mechanisms. CHEETAH supports any realizable LB mechanisms while guaranteeing PCC. We implemented several load balancing mechanisms that will be evaluated using multiple workloads in Sect. V-B. Among the load-aware LB mechanisms, we distinguish between metrics that can be tracked with or without coordination. Without any coordination, the LB can keep track of the number of packets/bytes sent per server and an estimate of the number of open connections based on a simple SYN/FIN counting

TABLE II
MEASURED TCP TIMESTAMP GRANULARITY FOR DIFFERENT WEBSITES. N.A. MEANS TCP TIMESTAMPS WERE DISABLED

Web site	TS granularity	Method
drive.google.com	1ms	gdown
dropbox.com	1ms	wget
twitch.tv	1ms	watch video
weobo.com	1ms	watch video
bilibili.com	N.A.	-
pan.baidu.com	N.A.	-
reddit.com	4ms	watch video
qq.com	4ms	watch video
instagram.com	4ms	watch video
onedrive.live.com	N.A.	-
facebook.com	1ms	watch video
twitter.com	N.A.	-
imdb.com	10ms	watch video

mechanism.[2] For LB approaches that require coordination with the servers, our implementation supports load distribution based on the CPU utilization of the servers. Note that using a least-loaded server for coordination-based approaches is a bad idea as a single server will receive all the incoming connections until its load metric increases and is reported to the LB, ultimately leading to instabilities in the system. Therefore, we decided to implement the following two load-aware balancing mechanisms, which we introduced in Sect. II: *(i)* power-of-two choices and *(ii)* a weighted round robin (WRR). For WRR, we devised a system where the weights of the servers change according to their relative (CPU) loads. We increase the weights for servers that are underutilized depending on the difference between their load and the average server load. More formally, the number of buckets N_i assigned to server i is computed as $N_i = round(10\frac{L_{avg}}{(1-\alpha)*L_i+\alpha*L_{avg}})$ where L_i is the load of a server, and α is a factor that tunes the speed of the convergence, which we set to 0.5. A perfectly balanced system would give $N = 10$ buckets to each server. An underutilized server gets more than N buckets (in practice limited to $3N$) while an overloaded server gets less than N buckets (lower bounded by 2).

C. P4 Implementation Using TCP Timestamps

The stateless CHEETAH LB follows exactly the description from Sect. III-A. We store the `all-servers` and the `VIP-to-servers` tables using exact-match tables. We rely on registers, which provide per-packet transactional memories, to store a counter that implements the weighted-round-robin LB. We note that implementing other types of LB mechanisms such as least-loaded in the data-plane is non-trivial in P4 since one would need to extract a minimum from an array in $O(1)$. This operation will likely requires to process the packet on the CPU of the switch. The insertion/deletion of the cookie on any subsequent non-SYN packet can be performed in the data-plane. The stateful CHEETAH LB adheres to the description in Sect. III-B. We use P4 registers to enable the insertion of connections into the `ConnTable` at the speed of the data-plane. We store the elements of the `ConnStack` stack in an array of registers, the `ConnTable` into an array of registers, and the index in the array of registers that stores the head of the stack. We implemented CHEETAH on both the `Tofino` [12] and `bmv2` [45] targets. The source code of both implementations is publicly available at our repository [36]. There is a tricky aspect to be considered when implementing the stateful version of CHEETAH. All registers in a PISA pipeline must be accessed in the same order in the different "if-else" branches. In stateful CHEETAH, we however need to handle *two* operations that require registers to be processed in the opposite order. First, when CHEETAH receives a TCP SYN from a client, it pops an empty table index from the `ConnStack` and it stores the hash of the connection identifier in the `ConnTable`. The second operation considers the reception of a TCP FIN packet, which requires to extract the cookie from the packet, check the hash, and pull the cookie onto the `ConnStack`. Since the stacks and tables are implemented in the registers, these two operations cannot be supported at the same time. The solution consists of handling the FIN only when it is received by the server and skip the hash check, assuming the server is trusted. Finally, we expect the `VIPToServers` and `AllServers` tables to occupy limited memory on the switch. A 40K-server datacenter would need roughly $2 \cdot (2\text{B}+4\text{B}) \cdot 40\text{K} = 480\text{KB}$ of memory whereas programmable switches have tens of MBs of SRAM memory [46]. Stateless CHEETAH can scale up to millions of servers. Stateful CHEETAH is limited by the amount of memory available to store connections (similarly to SilkRoad [11]). More specifically, with our current implementation we can use seven stages of the pipeline for `ConnTable`, yet further optimizations may be possible. The `ConnStack` data structure occupies 2 bytes per entry in the `ConnTable`.

D. P4/Picoquic Implementation Using QUIC

To show the feasibility of realizing CHEETAH without relying on TCP timestamps, we also implemented a prototype of stateless CHEETAH on top of the QUIC transport protocol [47]. We used the `picoquic` [48] implementation of QUIC. We extended `picoquic` by adding the encoding of the CHEETAH cookie in the second and third bytes of both the short and long connection IDs. We then implemented a P4 load balancer that can be compiled to the `bmv2` target. The LB extracts the cookie from the connection ID and redirects the packets to the correct server. We do not currently support QUIC migrations: when the source IP address and UDP port changes, the LB will not be able to forward the packet to the correct server. We leave the problem of guaranteeing PCC during QUIC connection migrations and the implementation of the Cheetah QUIC-based LB on the Tofino as future work. The source code of both implementations is publicly available at our repository [36].

V. EVALUATION

The CHEETAH LB design allows datacenter operators to unleash the power of arbitrary load balancing mechanisms while guaranteeing PCC, i.e, the ability to grow/shrink the LB and DIP pools without disrupting existing connections. In this section, we perform a set of experiments to assess the performance achievable through our stateless and stateful LBs. All experiments are based on the FastClick implementation

[2]We envision an ad-hoc reliable mechanism to signal closed connection between the LB and the server.

using TCP timestamps unless stated differently. All experiments scripts, including documentation for full reproducibility are available on GitHub [36].

We pose four main questions in this evaluation:

- *"How does the **cost of packet processing** in* CHEETAH *compare with existing LBs?"* (Sect. V-A)
- *"Can we reduce **load imbalances** by implementing more advanced LB mechanisms in* CHEETAH*?"* (Sect. V-B)
- *"How does the **PCC support** in* CHEETAH *compare with existing stateless LBs?"* (Sect. V-C)
- *"How does* CHEETAH *performs on a Tofino?"* (Sect. V-D)

Experimental setting. The LB runs on a dual-socket, 18-core Intel®Xeon®Gold 6140 CPU @ 2.30GHz, though only 8 cores are used from the socket attached to the NIC. Our testbed is wired with 100G Mellanox Connect-X 5 NICs [49] connected to a 32×100G NoviFlow WB5000 switch [50]. All CPUs are fixed at their nominal frequency.

Workload generation. To generate load, we use 4 machines with a single 8-core Intel®Xeon®Gold 5217 CPU @ 3.00GHz with hyper-threading enabled using an enhanced version of WRK [51] to generate load towards the LB. We also use four machines to run up to 64 NGINX web servers (one per hyper-thread), isolated using Linux network namespaces. Each NGINX server has a dedicated virtual NIC using SRIOV, allowing packets to be switched in hardware and directly received on the correct CPU core. We generate requests from the clients using uniform and bimodal distributions, as well as the large web server service distributions already used in the simulations of Sect. II.

Metrics. We evaluate the imbalance among servers using both the variance of the server loads and the 99th percentile flow completion times (FCTs). We refer to the FCT as the time elapsed between the time the last ACK received at the sender (the web server) and the time the sender initiates the connection. We measure the LB packet processing time in CPU cycles per second. Each point is the average of 10 runs of 15 seconds unless specified otherwise.

A. Packet Processing Analysis

We first investigate the cost in terms of packet processing time for using stateless CHEETAH. We compare it against stateful CHEETAH, a stateful LB based on per-core DPDK cuckoo-hash tables, and two hash-based LBs, one using hashes computed in hardware by the NIC for RSS [52], and one using hashes computed in software with DPDK. We also compare with a streamlined version of Beamer [5], without support for bucket synchronization, UDP, and MPTCP, thus representing a lower-bound on the Beamer performance.

Stateless CHEETAH incurs minimal packet processing costs. Fig. 5 shows the number of CPU cycles consumed by different LBs divided by the number of forwarded packets for increasing number of requests per second. We tune the request generation for a file of 8KB so that none of the machines were overloaded. The main result from this experiment is that stateless CHEETAH consumes almost the same number of CPU cycles per packet as the most optimized hardware assisted hash-based mechanism and significantly fewer cycles than stateful approaches. Beamer consumes more cycles than

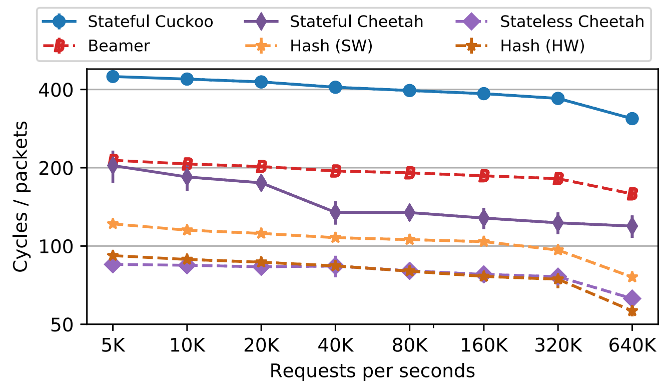


Fig. 5. CPU cycles/packet for various methods. CHEETAH achieves 5x fewer cycles than stateful LBs.

both CHEETAH LBs, still without bringing PCC guarantee (see Sect. V-C). This is mainly due to the operation of encapsulating the backup server into the packet header and the more compute-intensive operations needed by Beamer to lookup into a bigger "stable hashing" table. Finally, we note that each methods only need 4 CPU cores to saturate the 100Gbps link.

We further evaluate the packet processing latency of our Tofino implementation using in-built timestamp primitives. Our findings show that the QUIC and TCP timestamp implementations results in 0.7% and 30% higher processing times compared to a hash-based LB, respectively. One can notice the more complex parsing of TCP timestamps results in higher overheads. We note the packet processing latency does not affect the overall throughput achievable on a Tofino and that all processing latencies are below $1\mu\text{s}$.

Stateful CHEETAH outperforms cuckoo-hash based LBs. We also note in Fig. 5 the improvements in packet processing time of stateful CHEETAH (which uses a stack-based `ConnTable` table) compared to the more expensive stateful LBs using a cuckoo-hash table. Stateful CHEETAH achieves performance close to a stateless LB and a factor of $2-3$x better that cuckoo-hash based LBs.

Dissecting stateless CHEETAH performance. The key insight into the extreme performance of CHEETAH is that the operation of obfuscating the cookie only adds less than a 4-cycle hit. We in fact rely on the network interface card hardware to produce a symmetric hash (*i.e.*, using RSS). We expect the advent of SmartNICs as well as QUIC and IPv6 implementations, which have easier-to-parse headers, to perform even better. We note that our stateless CHEETAH implementation uses server-side TCP timestamp correction (see Sect. IV), which only imposes a 0.2% performance hit over the server processing time. If we were to use LB-side timestamp correction, we observe that the stateless CHEETAH modifies the timestamp MSB on the LB in just 30 cycles per packet performance hit. To summarize, stateless CHEETAH brings the same benefits as stateful LBs (in terms of load balancing capabilities) in addition to PCC guarantees at basically the same cost (and resilience) of stateless LBs.

B. Load Imbalance Analysis

We now assess the benefits of running CHEETAH using a non-hash-based load balancing mechanism and compare

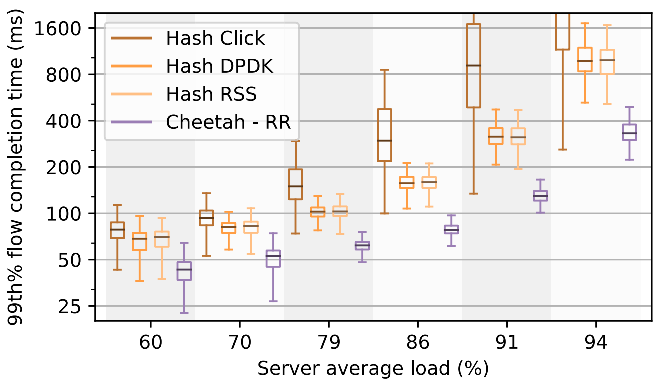


Fig. 6. 99th-perc. FCT for the increasing average server load. CHEETAH achieves 2x−3x lower FCT than Hash RSS.

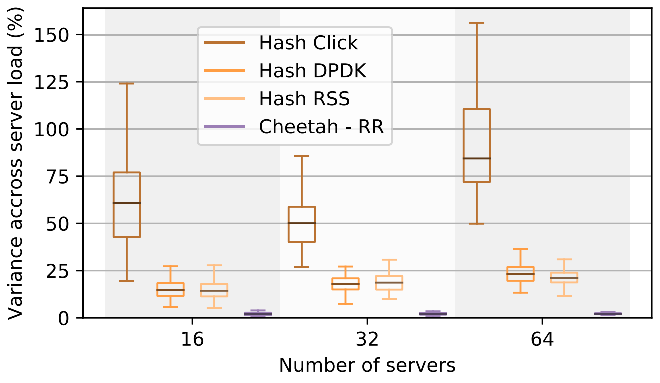


Fig. 7. Variance among servers' load of various methods for an increasing number of servers. CHEETAH, though stateless, allows a near-perfect load spreading.

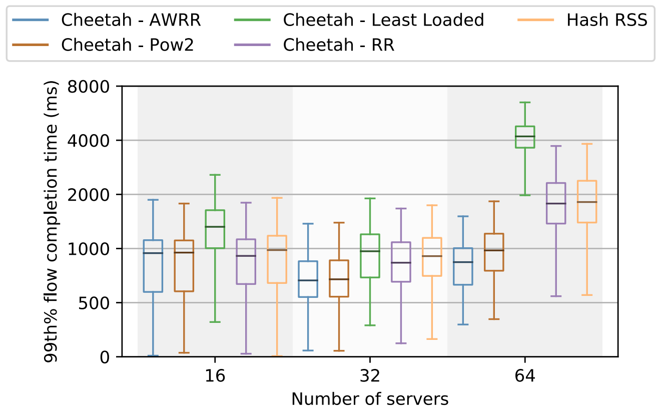


Fig. 8. Evaluation of multiple load balancing methods for a bimodal workload. Both AWRR and Pow2 outperform Hash RSS by a factor of 2.2 and 1.9 resp. with 64 servers.

it to different uniform hash functions (similarly to those implemented in Microsoft Ananta [2], Google Maglex [3], Beamer [5], and Faild [7]). We stress that we do not propose novel load balancing mechanisms but rather showcase the benefits of a load balancer design that supports any realizable load balancing mechanisms. We only evaluate stateless CHEETAH as the load imbalance does not depend on the stored state (and would result in similar performance).

In this experiment, each server performs a constant amount of CPU-intensive work to dispatch a 8KB file. The generator makes between 100 and 200 requests per server per second on average depending on our targeted system load. Given this workload, we expect an operator to choose a uniform round-robin LB mechanism to distribute the load.

CHEETAH significantly improves flow completion time. Fig. 6 compares CHEETAH with round-robin and hash-based LB mechanisms with 64 servers. We consider three hash functions: Click [34], DPDK [35], and the hardware hash from RSS. We stress the fact that these hash-based functions represent the quality of load balancing achievable by existing stateless (e.g, Beamer) and stateful (e.g, Ananta) LBs. We measure the 99th percentile flow completion time (FCT) tail latency for the increasing average server load. CHEETAH reduces the 99th percentile FCT by a factor of 2-3x compared to any hash-based mechanism, e.g, Hash RSS.

CHEETAH spreads the load uniformly. To understand why CHEETAH achieves better FCTs, we measure the variance of the servers' load over the experiment for an average server load of 60% and 16, 32, and 64 servers. Fig. 7 shows that the variance of RR is smaller than hash-based methods. This is because the load balancer iteratively spreads the incoming requests over the servers instead randomly spreading them. In this specific scenario, CHEETAH allows operators to leverage RR, which would otherwise be impossible with today's load balancers. Fig. 7 also shows that the quality of the hash function is important as the default function provided in Click does not perform well. In contrast, the CRC hash function used by DPDK is comparable to the Toeplitz based function used in RSS [53]. Moreover, the RSS function has the advantage of being performed in hardware.

CHEETAH improves FCT even with non-uniform workloads. Fig. 8 shows the tail FCT for a bimodal workload, where 10% of requests take 500ms to be ready for dispatching and the remaining ones take a few hundred microseconds. In this scenario, some servers will be loaded in an unpredictable way thus creating a skew that requires direct feedback from the servers to solve. We can immediately see that RR with 64 servers leads to very high FCTs. We evaluate three ways to distribute the incoming requests according to the current load (see Sect. IV-B): automatic weighted round robin (AWRR), power of two choices (Pow2), and the least loaded server. Each server piggybacks its load using a monitoring Python agent on the server that reports its load through an HTTP channel to the LB at a frequency of 100Hz, though experimental results showed similar performance at 10Hz. Least loaded performs poorly since it sends all the incoming requests to the same server for 10ms, overloading a single server. Pow2 and AWRR spread the load more uniformly as the LB penalizes those servers that are more overloaded. Consequently, both methods reduce the FCT by a factor of two compared to Hash RSS with 64 servers. These experiments show the potential of deploying advanced load balancing mechanisms to spread the service load.

C. PCC Violations Analysis

We now demonstrate the key feature of the stateless CHEETAH LB, i.e, avoid breaking connections while changing the server and/or LB pool sizes. We compare CHEETAH against Hash RSS, consistent hashing, and Beamer. In Beamer, a connection breaks whenever the bucket to which the connection

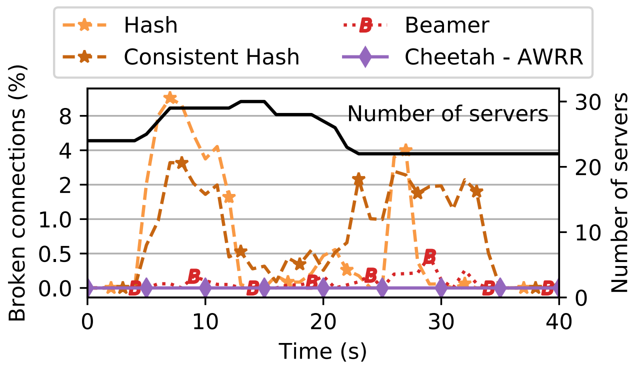


Fig. 9. Percentage of broken requests while scaling the number of servers. Cheetah guarantees PCC whereas hashing breaks up to 11% of the connections, consistent hashing 3% and Beamer up to 0.5%.

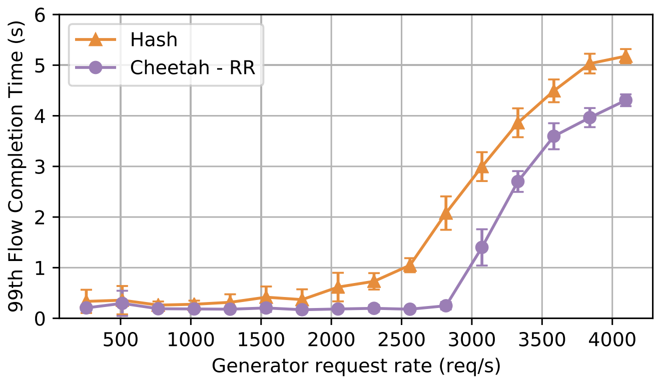


Fig. 10. Tail latency for QUIC experiment.

is mapped is updated twice before the connection ends. When a new server is added to the pool, Beamer re-assigns 1/N of the buckets from existing servers to the new server. To avoid as much disruption as possible, the oldest buckets are taken from existing servers to assign to a new server. When a server is removed, its buckets are assigned equally to remaining servers.

We start our experiment with a cluster consisting of 24 servers. We tune a python generator [36] to create 1500 requests/s, increasing following a sinusoidal load to 2500 requests/s and descending back to 1500 over the 40 seconds of the experiment. The workload follows the web server distribution. We iteratively add 7 servers to the pool as the load increases. We then drain 8 servers when the rate decreases. Fig. 9 shows the percentage of broken requests over completed requests every second over time. Some connections gets accounted as broken dozen of seconds later as clients send retransmissions before raising an error. Compared to Beamer, Cheetah achieves better load balancing with AWRR without breaking connections.

D. QUIC & Tofino Implementation

We devise a new experiment to showcase CHEETAH working with the QUIC protocol using a P4 program compiled on a Tofino switch. We use this experiment to simply show that the QUIC implementation on a Tofino works with different LB mechanisms. We do not aim to quantify the possible gains for different QUIC workloads and different LB mechanisms and we do not aim to benchmark the Tofino, which is *not* a bottleneck in any of our experiments. We integrate picoquic, the reference IETF QUIC implementation with the

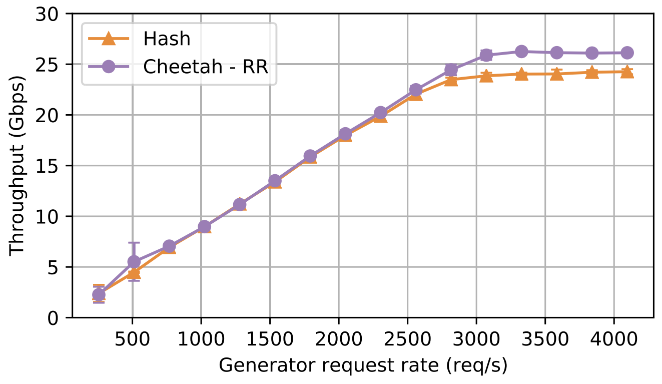


Fig. 11. Throughput for QUIC experiment.

WRK HTTP request generator [51] and use the embedded QUIC/HTTP 0.9 server of picoquic on one server per core, using 2 machines of 16 cores. We modify the picoquic server to generate the cookie in the connection ID as mentioned in Sect. IV-D. We compare a simple hash-based LB implemented in P4, and the stateless CHEETAH using Round Robin (RR) in Fig. 10 under an increasing request rate of 1MB files. At 2.5K req/s, we see that both LB mechanisms achieve roughly the same throughput of roughly 22.5Gbps. The tail latency of the hash-based LB is however 4x higher than the RR one, i.e, 1 second vs. 250ms, because of a worse load balance of the workload. At loads above 3K req/s, we observe in Fig. 11 an 8% improvement of the RR LB supported with CHEETAH over the hash-based LB. The reason why the two lines do not converge to the same throughput at higher loads is that the WRK load generator keeps at most *2048* simultaneous open connections, which effectively self-limit the generated load.

VI. FREQUENTLY ASKED QUESTIONS

Does CHEETAH preserve service resilience compared to existing LBs? Yes. We first discuss whether a client can clog a server. A client generating huge amounts of traffic using the same connection identifier can be detected and filtered out using heavy-hitter detectors [2]. This holds for any stateless LBs, e.g, Beamer [5]. A more clever attack entails reverse engineering the salted hash function and deriving a large number of connection identifiers that the LB routes to the same (specific) server, possibly with spoofed IP addresses. To do so, an attacker needs to build the $(conn.id, cookie) \mapsto server$ mapping. This requires performing complex measurements to verify whether two connection IDs map to the same server. Given that CHEETAH uses the same hash function of any existing LB (which is not cryptographic due to their complexity [7]), reverse engineering this mapping will be as hard as reverse engineering the hash of the existing LBs. As for the resilience to resource depletion, we note from Fig. 5 that stateless (stateful) CHEETAH has similar (better) packet processing times of today's stateless (stateful) LBs. Moreover, stateful CHEETAH supports line-rate insertions, which mitigate SYN flood attack. Thus, we argue that CHEETAH achieves the same levels of resilience of today's existing LB systems.

Does CHEETAH make it easy to infer the number of servers? Not necessarily. A 16-bit cookie permits at least an order of magnitude more servers than the number of servers used to operate the largest services [11]. If this is still a concern, one can hide the number of servers by reducing the size of the cookie and partitioning the connection identifier space similarly to our stateful design of CHEETAH.

Does CHEETAH support multipath transport protocols? Yes. In multipath protocols, different sub-connection identifiers must be routed to the same DIP. Previous approaches exposed the server's id to the client [29], [30]; however, this decreases the resilience of the system decreases. CHEETAH can use a different permutation of `AllServers` for each additional i'th sub-connection. Clients inform the server of the new sub-connection identifier to be added to an existing connection. The server replies with the cookie to be used using the i'th `AllServers` table. This keeps the resilience of the system unchanged compared to the single path case.

VII. RELATED WORK

There exists a rich body of literature on datacenter LBs [2]–[5], [8], [11], [14]–[22], [54], [55]. We do not discuss network-level DC load balancers [14]–[21], whose goal is to load balance the traffic within the DC network and do not deal with per-connection-consistency problems.

Stateless LBs. Existing stateless LBs rely on hash functions and/or "daisy chaining" techniques to mitigate PCC violations (Sect. II), e.g, ECMP [22], WCMP [8], consistent hashing [54], Beamer [5], and Faild [7]. The main limitation of such schemes is the suboptimal balancing of the server loads achieved by the hash function, which is known to grow exponentially in the number of servers [56]. Shell [57] uses the timestamp option as a reference to an history of indirection tables, which comes at both the expense of memory and low-frequency load rebalancing. Encoding the connection-to-server mapping has been discussed in an editorial note without discussing LB resilience, stateful LBs, or implementing and evaluating such a solution [58].

QUIC-LB [55] is a high-level design proposal at the IETF for a stateless LB that leverages the `connection-id` of the QUIC protocol for routing purposes. While sharing some similarities to our approach, QUIC-LB *(i)* does not present a design of a stateful LB that would solve cuckoo-hash insertion time issues, *(ii)* does not evaluate the performance obtainable on the latest generation of general-purpose machines, *(iii)* relies on the modulo operation with an odd number to hide the server from the client, an operation that is not supported in P4, and *(iv)* does not discuss multi path protocols. Stateless load balancers that support multipath transport protocols have been proposed in the past. Such load balancers guarantee all the subflows of a connection are routed to the same server by explicitly communicating an identifier of the server to the client [29], [30]. These approaches may be externally exploited by malicious users to cause targeted imbalances in the system, which is prevented in CHEETAH thanks to using distinct hashes for the subflows (see Sect. VI).

Stateful LBs. Existing stateful LBs store the connection-to-server mapping in a cuckoo-hash table [2]–[4], [6], [11] (see Sect. II). These LBs still rely on hash-based LB mechanisms — as these lead to fewer PCC violations when changing the number of LBs. In contrast, CHEETAH decouples PCC support from the LB logic, thus allowing operators to choose any realizable LB mechanism. Moreover, hash-based tables suffer from slow (non-constant) insertion time. FlowBlaze [26] and SilkRoad [11] tackled this problem using a stash-based and bloom-filter-based implementations, respectively. Yet, both solutions cannot guarantee insertions in constant-time: FlowBlaze relies on a stash that may be easily filled by an adversary while SilkRoad is limited by both the size of the bloom filter (which is quickly filled under SYN-flood attacks) and the complexity of the implementation. Other stateful LBs such as LBAS [59], Spotlight [60] Concury [61], and the work of Hunt *et al.* [62] replace the hash server selection mechanism with a more advanced scheme yet insertions into the connection table cannot be performed in constant time. CRAB [63] is an LB that only participates in the connection establishment and then migrates a TCP connection to the selected server. CRAB supports arbitrary server selection mechanisms but (i) it requires modifications to the TCP stack at both clients and servers, (ii) does not obfuscate the server IDs, and (iii) does not design a fast data-plane for stateful LBs. CHEETAH differs from these works by using a constant-time stack that is amenable to fast implementation in the dataplane. Existing stateful LBs also suffer from the fact that the 1$^{\text{st}}$-tier of stateless ECMP LBs reshuffle connections to the wrong stateful LB when the number of LBs changes. In contrast, 1$^{\text{st}}$-tier stateless CHEETAH guarantees connections reach the correct stateful LB regardless of changes in the LB pool size.

VIII. CONCLUSION

We introduced CHEETAH, a novel building block for load balancers that guarantees PCC and supports any realizable LB mechanisms. We implemented CHEETAH on both software switches and programmable ASIC Tofino switches. We consider this paper as a first step towards unleashing the power of load balancing mechanisms in a resilient manner. We leave the question of whether one can design novel load balancing mechanisms tailored for Layer 4 LBs as well as deployability with existing middleboxes as future work.

ACKNOWLEDGMENT

The authors would like to thank all the anonymous reviewers for their value feedback.

REFERENCES

[1] T. Barbette *et al.*, "A high-speed load-balancer design with guaranteed per-connection-consistency," in *Proc. USENIX NSDI*, 2020, pp. 667–683.

[2] P. Patel *et al.*, "Ananta: Cloud scale load balancing," in *Proc. ACM SIGCOMM*, 2013, pp. 207–218.

[3] D. E. Eisenbud *et al.*, "Maglev: A fast and reliable software network load balancer," in *Proc. USENIX NSDI*, 2016, pp. 523–535.

[4] C. Hopps. (Sep. 2019). *Katran: A High Performance Layer 4 Load Balancer*. [Online]. Available: https://github.com/facebookincubator/katran

[5] V. Olteanu, A. Agache, A. Voinescu, and C. Raiciu, "Stateless datacenter load-balancing with beamer," in *Proc. USENIX NSDI*, 2018, pp. 125–139.

[6] R. Gandhi *et al.*, "Duet: Cloud scale load balancing with hardware and software," in *Proc. ACM SIGCOMM*, 2014, pp. 27–38.

[7] J. T. Araujo, L. Saino, L. Buytenhek, and R. Landa, "Balancing on the edge: Transport affinity without network state," in *Proc. USENIX NSDI*, 2018, pp. 111–124.

[8] J. Zhou *et al.*, "WCMP: Weighted cost multipathing for improved fairness in data centers," in *Proc. EuroSys*, 2014, pp. 1–14.

[9] W. Wang and G. Casale, "Evaluating weighted round Robin load balancing for cloud web services," in *Proc. 16th Int. Symp. Symbolic Numeric Algorithms Sci. Comput.*, Sep. 2014, pp. 393–400.

[10] M. D. Mitzenmacher, "The power of two choices in randomized load balancing," Ph.D. dissertation, Dept. Comput. Sci., Univ. California, Berkeley, CA, USA, 1996.

[11] 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 *Proc. ACM SIGCOMM*, 2017, pp 15–28.

[12] Barefoot. (Sep. 2019). *Tofino: World's Fastest P4 Programmable Ethernet Switch ASIC*. [Online]. Available: https://www.barefootnetworks.com/products/brief-tofino/

[13] T. Barbette. (2015). *Github–FastClick*. [Online]. Available: https://github.com/tbarbette/fastclick

[14] N. Kang, M. Ghobadi, J. Reumann, A. Shraer, and J. Rexford, "Efficient traffic splitting on commodity switches," in *Proc. 11th ACM Conf. Emerg. Netw. Exp. Technol.*, Dec. 2015, pp. 1–13.

[15] S. Ghorbani, Z. Yang, P. B. Godfrey, Y. Ganjali, and A. Firoozshahian, "DRILL: Micro load balancing for low-latency data center networks," in *Proc. SIGCOMM*, 2017, pp. 225–238.

[16] T. Benson, A. Anand, A. Akella, and M. Zhang, "MicroTE: Fine grained traffic engineering for data centers," in *Proc. 7th Conf. Emerg. Netw. EXperiments Technol. (CoNEXT)*, New York, NY, USA, 2011, p. 8, doi: 10.1145/2079296.2079304.

[17] H. Zhang, J. Zhang, W. Bai, K. Chen, and M. Chowdhury, "Resilient datacenter load balancing in the wild," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2017, pp. 253–266.

[18] N. Katta, M. Hira, A. Ghag, C. Kim, I. Keslassy, and J. Rexford, "CLOVE: How I learned to stop worrying about the core and love the edge," in *Proc. ACM HotNets*, 2016, pp. 155–161.

[19] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford, "HULA: Scalable load balancing using programmable data planes," in *Proc. ACM SOSR*, 2016, pp. 1–12.

[20] K. He, E. Rozner, K. Agarwal, W. Felter, J. Carter, and A. Akella, "Presto: Edge-based load balancing for fast datacenter networks," in *ACM SIGCOMM*, 2015, pp. 465–478.

[21] M. Alizadeh *et al.*, "CONGA: Distributed congestion-aware load balancing for datacenters," in *ACM SIGCOMM*, 2014, pp. 503–514.

[22] C. Hopps, *Analysis of an Equal-Cost Multi-Path Algorithm*, document RFC 2992, Nov. 2000.

[23] M. Chiesa, G. Kindler, and M. Schapira, "Traffic engineering with equal-cost-multipath: An algorithmic perspective," *IEEE/ACM Trans. Netw.*, vol. 25, no. 2, pp. 779–792, Apr. 2017.

[24] J. C. Villanueva. (Jun. 2015). *Comparing Load Balancing Algorithms*. [Online]. Available: https://www.jscape.com/blog/load-balancing-algorithms

[25] R. Pagh and F. F. Rodler, "Cuckoo hashing," *J. Algorithms*, vol. 51, no. 2, pp. 122–144, May 2004.

[26] S. Pontarelli *et al.*, "Flowblaze: Stateful packet processing in hardware," in *Proc. USENIX NSDI*, 2019, pp. 531–548.

[27] F. Németh, M. Chiesa, and G. Rétvári, "Normal forms for match-action programs," in *Proc. 15th Int. Conf. Emerg. Netw. Exp. Technol.*, Dec. 2019, pp. 44–50.

[28] M. Majkowski. (Jan. 2018). *SYN Packet Handling in the Wild*. [Online]. Available: https://blog.cloudflare.com/syn-packet-handling-in-the-wild/

[29] F. Duchene and O. Bonaventure, "Making multipath TCP friendlier to load balancers and anycast," in *Proc. IEEE 25th Int. Conf. Netw. Protocols (ICNP)*, Oct. 2017, pp. 1–10.

[30] V. Olteanu and C. Raiciu, "Datacenter scale load balancing for multipath transport," in *Proc. Workshop Hot topics Middleboxes Netw. Function Virtualization*, Aug. 2016, pp. 20–25.

[31] H. Tabunshchyk. (Dec. 2017). *Super Fast Packet Filtering With eBPF and XDP*. [Online]. Available: https://bit.ly/2mpoIyO

[32] A. N. Kolmogorov, "Three approaches to the quantitative definition of information," *Int. J. Comput. Math.*, vol. 2, nos. 1–4, pp. 157–168, 1968.

[33] P. Fraigniaud and C. Gavoille, "Memory requirement for universal routing schemes," in *Proc. 14th Annu. ACM Symp. Princ. Distrib. Comput. (PODC)*, 1995, pp. 223–230.

[34] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Trans. Comp. Syst.*, vol. 18, no. 3, pp. 263–297, Aug. 2000. [Online]. Available: http://doi.acm.org/10.1145/354871.354874

[35] Linux Foundation. (2015). *Data Plane Development Kit*. [Online]. Available: http://www.dpdk.org

[36] Cheetah Authors. (2020). *Github–Cheetah Source Code*. [Online]. Available: https://github.com/cheetahlb/

[37] D. Borman, R. T. Braden, V. Jacobson, and R. Scheffenegger, *TCP Extensions for High Performance*, document RFC 7323, Sep. 2014. [Online]. Available: https://rfc-editor.org/rfc/rfc7323.txt

[38] C. Filsfils, S. Previdi, L. Ginsberg, B. Decraene, S. Litkowski, and R. Shakir, *Segment Routing Architecture*, document RFC 8402, Jul. 2018. [Online]. Available: https://rfc-editor.org/rfc/rfc8402.txt

[39] Alexa. *The Top 500 Sites on the Web*. Accessed: Nov. 10, 2020. [Online]. Available: https://www.alexa.com/topsites

[40] Statcounter. *Mobile Operating System Market Share Worldwide*. Accessed: Nov. 10, 2020. [Online]. Available: https://gs.statcounter.com/os-market-share/mobile/worldwide

[41] *Desktop vs Mobile vs Tablet Market Share Worldwide*. Accessed: Nov. 10, 2020. [Online]. Available: https://gs.statcounter.com/platform-market-share/desktop-mobile-tablet

[42] *Desktop Operating System Market Share Worldwide*. Accessed: Nov. 10, 2020. [Online]. Available: https://gs.statcounter.com/os-market-share/desktop/worldwide

[43] NetApplications. *Market Share for Mobile, Browsers, Operating Systems and Search Engines | Netmarketshare*. Accessed: Nov. 10, 2020. [Online]. Available: https://bit.ly/37iRNOK

[44] W. Eddy, *TCP SYN Flooding Attacks and Common Mitigations*, document RFC 4987, Aug. 2007. [Online]. Available: https://rfc-editor.org/rfc/rfc4987.txt

[45] Community Authors. (2020). *Github–Behavioral Model (BMV2) Source Code*. [Online]. Available: https://github.com/p4lang/behavioral-model

[46] K. Qian *et al.*, "FlexGate: High-performance heterogeneous gateway in data centers," in *Proc. ACM APNet*, 2019, pp. 36–42.

[47] J. Iyengar and M. Thomson. *QUIC: A UDP-Based Multiplexed and Secure Transport*. Accessed: Nov. 10, 2020. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-quic-transport/

[48] C. Huitema. (2020). *Picoquic QUIC Implementation*. [Online]. Available: https://github.com/private-octopus/picoquic

[49] M. Technologies. *ConnectX-5 EN Single/Dual-Port Adapter Supporting 100Gb/s Ethernet*. Accessed: Nov. 10, 2020. [Online]. Available: https://www.mellanox.com/page/products_dyn?product_family=260&mtag=connectx_5_en_card

[50] NoviFlow. (Sep. 2019). *Katran: A High Performance Layer 4 Load Balancer*. [Online]. Available: https://noviflow.com/noviswitch/

[51] W. Glozer. *WRK*. Accessed: Nov. 10, 2020. [Online]. Available: https://github.com/wg/wrk

[52] Intel. (2016). *Receive-Side Scaling (RSS)*. [Online]. Available: http://www.intel.com/content/dam/support/us/en/documents/network/sb/318 483001us2.pdf

[53] H. Krawczyk, "New hash functions for message authentication," in *Proc. EUROCRYPT*, 1995, pp. 301–310.

[54] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in *Proc. ACM STOC*, 1997, pp. 654–663.

[55] M. Duke, "QUIC-LB: Generating routable QUIC connection IDs," IETF Internet-Draft draft-duke-quic-load-balancers-04, May 2019. Accessed: Nov. 10, 2020. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-quic-load-balancers

[56] V. F. Kolchin, B. A. Sevastyanov, and V. P. Chistyakov, *Random Allocations*. Washington, DC, USA: Winston, 1978.

[57] B. Pit-Claudel, Y. Desmouceaux, P. Pfister, M. Townsley, and T. Clausen, "Stateless load-aware load balancing in p4," in *Proc. IEEE 26th Int. Conf. Netw. Protocols (ICNP)*, Sep. 2018, pp. 418–423.

[58] J. McCauley, A. Panda, A. Krishnamurthy, and S. Shenker, "Thoughts on load distribution and the role of programmable switches," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 49, no. 1, pp. 18–23, Feb. 2019.

[59] J. Zhang *et al.*, "Fast switch-based load balancer considering application server states," *IEEE/ACM Trans. Netw.*, vol. 28, no. 3, pp. 1391–1404, Jun. 2020.

[60] A. Aghdai, C.-Y. Chu, Y. Xu, D. Dai, J. Xu, and J. Chao, "Spotlight: Scalable transport layer load balancing for data center networks," *IEEE Trans. Cloud Comput.*, early access, Sep. 18, 2020, doi: 10.1109/TCC.2020.3024834.

[61] S. Shi *et al.*, "Concury: A fast and light-weight software cloud load balancer," in *Proc. ACM SoCC*, 2020, pp. 179–192.

[62] G. Hunt, E. Nahum, and J. Tracey, "Enabling content-based load distribution for scalable services," IBM, 1997. Accessed: May 20, 2020. [Online]. Available: http://www.cs.columbia.edu/~nahum/papers/ibm-tr97-cluster.pdf

[63] M. Kogias, R. Iyer, and E. Bugnion, "Bypassing the load balancer without regrets," in *Proc. 11th ACM Symp. Cloud Comput.*, Oct. 2020, pp. 193–207.