Loom: Switch-based Cloud Load Balancer with Compressed States

Jiao Zhang†‡, Yuxuan Gao†, Shubo Wen†, Tian Pan†‡, Tao Huang†‡
†Beijing University of Posts and Telecommunications, Beijing, China
‡Purple Mountain Laboratories, Nanjing, China
Email: {jiaozhang, gaoyuxuan, 2013210006, pan, htao}@bupt.edu.cn

***Abstract*—Layer-4 load balancers play a critical role in large-scale data centers. Recently, load balancers implemented on programmable switches have attracted much attention since they overcome the inflexibility of dedicated load balancers and high latency of software load balancers. However, keeping per-connection state easily leads to storage exhaustion, especially under resource exhaustion attacks. Although several stateless load balancers are proposed to address this issue, the state management burden is offloaded to backend servers, causing high deployment and running costs. In this paper, a load balancer called Loom with compressed states is proposed for large-scale data centers. Firstly, we propose a novel classifier-based load balancer idea to avoid directly maintaining per-connection state. Then, a circulating Bloom filter structure is proposed that can efficiently classify connections as well as be implemented on existing programmable switches. Theoretical analysis shows that Loom can maintain $11 \sim 30x$ more concurrent connections than those directly storing the 5-tuple of connections. Loom is implemented in hardware P4 switches and experimental results indicate that $11 \sim 29x$ more concurrent connections can be maintained in Loom, which is close to the theoretical results. Besides, Loom is resistant to resource exhaustion attacks and reduces the percentage of broken connections by up to 57% with an SYN flood.**

***Index Terms*—Cloud Data Center, Load Balancer, Compressed States, Programmable Switch, Resource Exhaustion Attack**

I. Introduction

Layer-4 load balancing is an indispensable function in large-scale data centers. To achieve outward scalability, a cloud service is generally provisioned by a large number of backend servers [1], [2]. Each service exposes several public virtual IP (VIP) addresses for users to establish connections. Load balancers in data centers need to dispatch requests destined to these VIPs to one of the corresponding backend servers. Each backend server has a unique direct IP (DIP) address. A desirable layer-4 load balancer should achieve quite low latency and keep Per-Connection Consistency (PCC) in various scenarios. Per-connection consistency means that a connection will always be mapped to the same DIP server even if the DIP pool updates or the mapping function between VIPs and DIPs changes.

Existing load balancers can be mainly classified into three categories according to their implementation platforms. First, *dedicated load balancers* [3], [4]. This kind of load balancer is usually expensive and hard to scale. Second, *software load balancers* [5]–[8]. By implementing load balancers in a great number of commodity servers, software load balancers exhibit high scalability and availability. However, this kind of load balancer processes packets in software, which leads to high packet processing latency and high latency jitter when traffic load is relatively large. Third, *switch-based load balancers*. To address the drawbacks of dedicated and software load balancers, some attempts have been conducted to implement load balancers in switches [9]–[12]. This method is cost-effective and can process packets in line-rate as dedicated load balancers. However, switch-based load balancers are limited by the fixed processing logic. Recently, with the development of programmable switches, there is a trend to implement load balancers on them [11], [13]–[15].

Most load balancers record the mapping relationship between every connection and its corresponding backend server locally. That is, a load balancer generally assigns a backend server for the first packet of a new connection by using a hash function. Then the assignment result will be recorded locally. In this way, all the following packets of that connection can be guaranteed to be dispatched to the same backend servers even if the hash function changes or the backend servers pool updates. However, the hardware resource of switches is limited. Maintaining the states of all connections consumes much memory space. What's more, once a resource exhaustion attack like an SYN flood [16] happens, the memory of switches will be easily exhausted, failing to guarantee PCC.

Although several stateless load balancers have been proposed recently to solve the state management problems [13], [17], they transfer the burden of keeping per-connection consistency to backend servers. Each backend server requires to implement a module to detect whether the received packets should be processed by it. If not, it needs to forward the packets to another server that possibly has the state for the packet. This kind of method will increase packet processing latency. Besides, installing a new module leads to a deployment cost. What's more, some computation and bandwidth resources of backend servers will be taken for continuously forwarding packets.

In this paper, we propose a semi-stateful load balancer, Loom, which is implemented on programmable switches. Since most memory of stateful load balancers is taken to maintain per-connection state, *we firstly propose a novel classifier-based load balancer idea to avoid directly maintaining per-connection state*. Specifically, by using a classifier to differentiate connections as well as maintaining their corresponding hash functions, all connections can be directed to the correct backend servers. Since the classifier potentially occupies much less memory than directly maintaining per-connection state especially with a large number of connections, the load balancer can deal with more connections given a fixed memory space.

Then, *we propose a specific design of the classifier-based load balancer idea by leveraging several Bloom filters [18] and devising a circulating update scheme.* The average storage space taken up by one connection greatly decreases. *Theoretical analysis shows that Loom can support 40.4 million connections with only 50 MBytes SRAM, which indicates* $11 \sim 30x$ *improvement over those directly storing entire connection information and about 2.7 times over those with per-entry compressed solutions [11].* Besides, it greatly decreases the percentage of broken connections caused by resource exhaustion attacks like SYN flood.

We implemented Loom in an Edgecore Wedge100BF-65X switch [19] using the P4 language [20]. The parameters of Loom are set properly according to the limitations of P4 switches and the theoretical analysis results. Experimental results show that Loom significantly improves SRAM utilization. The number of connections that can be maintained is consistent with the theoretical analysis results. Compared with stateful load balancers, Loom can reduce the percentage of broken connections by up to 57% under SYN flood attacks in our experiments.

In sum, the key contributions of the work are as follows:

- We propose a novel classifier-based load balancer idea to save memory space, which uses a classifier to differentiate connections and maintains different versions of the hash functions. Thus, each connection can obtain its correct backend servers based on their corresponding hash functions.
- We propose a detailed design of the classifier-based load balancer idea by leveraging multiple Bloom filters and the proposed circulating update scheme. Then we theoretically analyze how many connections can be maintained as well as the proper parameters' values in Loom.
- Based on the analysis results, we implement Loom in a hardware P4 switch and validate that it can maintain $11 \sim 29x$ more connections than those directly storing the 5-tuple of connections and reduce the percentage of broken connections by up to 57% under SYN flood attacks.

In the remainder of this paper, we summarize the advantages and disadvantages of load balancers with different implementation platforms and state management schemes in §II. In §III and §IV, the basic idea and design details of Loom are presented. The analysis and implementation of Loom are described in §V. §VI shows experimental results. Finally, the paper is concluded in §VII.

II. Background and Previous Work

The job of a load balancer is to map connections from VIPs to DIPs evenly. In general, a load balancer is a middlebox between servers and clients. Next, we summarize the existing work and discuss their limitations from the perspective of platform and state management.

A. Platform

Dedicated load balancers: Initially, load balancers are implemented in dedicated hardware [3], [4], [21], [22]. But there are some drawbacks. Thus, dedicated load balancers are generally more expensive than commodity servers and switches in data center networks. Secondly, the features and capacity of dedicated hardware are hard to keep up with demand. At the same time, the upgrade of a dedicated load balancer usually needs to change the hardware instead of upgrading the software, which means a long upgrade period. Thus, although this kind of load balancer leads to high performance, there's an obvious lack of flexibility.

Software load balancers: To overcome the limitations of dedicated load balancers, some solutions are proposed to implement load balancers on commodity servers. Maglev [6], Ananta [5], Coucury [8], and Karan [23] are representative software load balancers. Their capacity can be easily adjusted by adding or removing load balancer servers. Meanwhile, Maglev improves the throughput of a single load balancer server through batch processing and kernel bypass techniques. Typically, software load balancers run on thousands of commodity servers. This makes them very flexible but also brings some drawbacks. Firstly, software load balancers typically need a significant number of servers (up to 3.75% of the total servers in a data center) to handle large traffic [11]. This leads to high capital expenditure [9] and high power costs. Since traffic size of a medium-sized data center is as high as 15Tbps, which requires more than 4000 SMuxes and costs more than 10 million dollars. For a programmable switch of 3.3 Tbps, it only costs 10500 dollars. For load balancing of the same traffic size, the cost of software load balancer is much higher [24]. Secondly, software load balancers introduce a high latency and jitter while processing packets in software. About 44% of the total Internet traffic is VIP traffic, and it accounts for about 30% of the total VIP traffic in a data center. The remaining 70% of the VIP traffic comes from the inter-services within a data center [5], and the traffic intra-DC is sensitive to delay (e.g., 2-5μs RTT with RDMA [25])

Switch-based load balancers: Dedicated hardware and commodity servers are two extremes, and each one has its own flaws. Thus, it is natural to think about using a device that combines the two to implement the load balancing function. Duet [9] and Rubik [26] are hybrid load balancers that combine the ECMP ability of commodity switches with software load balancers. They aim to address the performance bottleneck of software load balancers by offloading some workload to commodity switches. However, no matter using the original ECMP table in common switches or using the OpenFlow switch, the number of entries that can be stored in a switch is very limited. Besides, the fixed packet processing procedure and matching structure of common switches make it difficult to design complex logic and advanced data structures in the data plane for future upgrades.

In recent years, some programmable switches are developed, such as P4 switches [19], and Cavium's XP70 [27]. These switches have software-like flexibility and high performance. SilkRoad [11] aims to ensure PCC during frequent backend server pool updates with programmable switches. SilkRoad tracks per-flow state in programmable switches by directly storing the mapping relationship between every connection and its corresponding backend server. The relatively small SRAM size limits the amount of connections SilkRoad can maintain. This also makes SilkRoad vulnerable to resource exhaustion attacks.

B. State management

From the perspective of state management, load balancers can be divided into stateful and stateless ones. However, both of them have some limitations.

Stateful load balancers: Most existing load balancers keep per-connection state to ensure connection consistency. Once a connection is assigned to a backend server, the following packets of this connection will be routed to the same backend servers until the connection finishes. These lead to two main drawbacks. Firstly, saving the state of each connection consumes a lot of memory. Although this is not a big problem for software load balancers, for hardware load balancers with very limited resources, the memory used to record the state of each connection occupies almost all resources, even up to 91.7% [11]. Secondly, while recording per-connection state works well under normal conditions, stateful load balancers suffer from state mismatch between the load balancer and the backend servers since they can always see only one direction of flows [13]. This state mismatch may exhaust the memory of the load balancer quickly under circumstances like SYN flood or break alive connections.

Stateless load balancers: Beamer [13] and Failed [17] are two stateless load balancers. They both offload state management to application servers. Stateless load balancers keep PCC by leveraging the states maintained by backend servers instead of maintaining connection states locally. One of the most obvious benefits is that a stateless load balancer will no longer be affected by SYN Flood attacks since it does not save connection states. Furthermore, the simple logic of stateless load balancers improves performance and scalability. However, this mechanism makes backend servers involved in the core functionality of load balancing, which results in CPU and bandwidth overhead on the server-side. These extra expenses cannot be ignored in large-scale data centers. In addition, each backend server needs to be modified to fit into the load balancing system. Adaptation to different server platforms will also result in overhead. Although CHEETAH [28] can achieve stateless load balancing without expanding the backend servers, CHEETAH needs to insert a cookie in the header of all packets to assist load balancers to complete the packet forwarding. This needs to modify clients. Besides, even encoding the cookie in the packet header, such as TCP timestamp, may affect the normal operation of TCP.

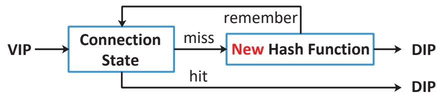


Fig. 1: Basic idea of stateful load balancers. Maintaining connection states occupies much memory space.

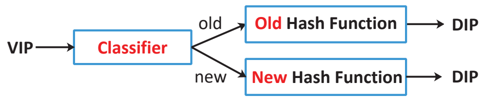


Fig. 2: Basic idea of Loom. By utilizing a classifier, much memory space could be saved.

In a word, offloading state management to the backend servers or packet header is not a proper strategy for all scenarios, but storing the states of connections directly on the load balancer also has limitations due to the resource constraints and state mismatch.

III. Design Rationale of Loom

A. Basic idea

A desirable load balancer should have the following characteristics: low latency, high flexibility, maintaining a large number of active connections, resistance to resource exhaustion attacks, and transparency to connection ends.

To design a switch-based load balancer that has the desirable characteristics, *we firstly investigate why a load balancer is required to save the states of each connection*. What a load balancer requires to accomplish is to send all the packets of a connection to the same backend server. If the number of backend servers varies before the connection ends, the relationship between a hash result and its corresponding backend server may change. Therefore, the load balancer will likely send subsequent packets of some connections to the wrong servers. To address this problem, a straightforward idea is to record the mapping between each connection and the backend server. Most stateful load balancers forward packets according to this procedure as shown in Figure 1.

However, maintaining per-connection state consumes much memory as the number of connections increases, while the hash function takes quite a small memory space. If we additionally save the last hash function and classify incoming packets into old connections and new ones. Then the packets from old connections could be hashed using the old hash function and will not be forwarded to the wrong backend server. *Since maintaining a classifier and multiple hash functions consumes much less memory compared with maintaining per-connection state, the efficiency of resource utilization can be significantly improved*. This is also the basic idea of Loom as shown in Figure 2.

Figure 3 uses a simple example to illustrate the above basic idea. Let c_1, c_2 be two connections established before a DIP pool update, and their corresponding DIP is A and B, respectively. Connection c_3 is a new connection that starts after the DIP pool update. The old hash function is used to obtain a proper DIP server for packets without local record before the DIP pool update, and the new hash function is used to obtain a proper DIP server for packets without local record after the DIP pool update. Figure 3 shows how to process packets after the DIP pool update. A stateful load balancer shown in Figure 3a queries the recorded connection states to obtain the correct DIP for each packet (for c_1 and c_2). If the connection state is not found, then it will use the new hash function to get a DIP (for c_3). However, Loom shown in Figure 3b queries whether an arrival connection belongs to the set consisting of c_1 and c_2. If it belongs, then the packet will be assigned its DIP according to the old version of `route_table` and `map_table` with the old hash function, otherwise, it uses the new version of `route_table` and `map_table` with new hash function. This method not only ensures that old and new connections are correctly forwarded, but also saves space.

In summary, by designing a state compression structure within the logic of the programmable switch, Loom maximizes the SRAM utilization and thus stores more active connections as well as becomes more resistant to SYN flood attacks.

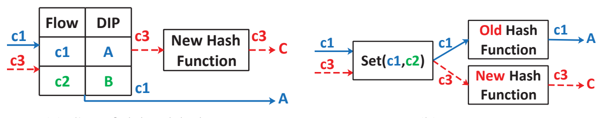
B. *Classifing structure*

Bloom filter [18] is a space-efficient probabilistic data structure designed to answer whether an element is in a set. Due to the space efficiency and quick query speed of Bloom filter, it has been widely used in databases, storage systems, networks and so on. However, it has a false positive problem and difficulty in deleting elements. Thus, a lot of attempts have been conducted to overcome these shortcomings of Bloom filters [29]–[31]. Counting Bloom filter [29], Quotient filter [30], and Cuckoo filter [31] are three typical variants of Bloom filter. Counting Bloom filter [29] solves the drawback of Bloom filters that items cannot be deleted. However, it takes up several times the storage space compared with Bloom filter. Quotient filter [30] and Cuckoo filter [31] are two structures that aim to avoid false positive events. If a collision occurs when an element is inserted in a Quotient filter, the element at the original position needs to be cyclically shifted to insert the new element. When a position conflict occurs in the cuckoo filter, the item to be inserted will take up the conflicted position while the original item will be moved to an alternate position. However, these shift or move operations are difficult to be implemented on a programmable switch [11].

In summary, although the improvements based on Bloom filter have advantages in storage space and query speed, it is hard to implement them on programmable switches. Therefore, in Loom, we propose a circulating Bloom filter data structure that can be implemented on a programmable switch as well as save storage space.

IV. Details of Loom

In this section, we will firstly describe the framework of Loom. Subsequently, the proposed circulating Bloom filter structure will be presented.



(a) Stateful load balancers (b) Loom

Fig. 3: A simple example to illustrate the basic idea.

A. *Framework*

Figure 4 shows the framework and workflow of Loom. Following the basic idea of Loom, we use multiple simple and easily implemented Bloom filters to classify connections in an SRAM-efficient way. Besides, a novel circulating update method is proposed to handle DIP pool updates and mitigate the impact of SYN flood attacks.

There are five major modules in Loom:

(1) **Conn_table** maintains the precise state of a quite small percentage of connections. It maps the connections to DIPs to maintain connection consistency when there is a false positive event (§IV-B2) caused by Bloom filters. To reduce the match field size, we store the hash value of a connection rather than its 5-tuple. One entry will be deleted if it has not been hit for a period of time.

(2) **Bloom filter** is used as a classifier for differentiating new connections and old connections. There could be several Bloom filters in our design. Figure 4) uses three Bloom Filters as an example to illustrate how Loom works. One Bloom filter (`new Bloom` maintains new connections. The other two store old connections and answer which version of `route_table` should be used. We refer them as `query Bloom` as shown in Figure 4. There is only one `new Bloom` in Loom, but the number of `query Bloom` could be larger than 1. The number of `query Bloom` means how many different old versions of `route_table` will be recorded.

(3) **Syn_table** triggers a false positive event if the processing packet is an SYN packet and one `query Bloom` filter gives a positive answer. This is because an SYN packet is the first packet of a connection, it indicates that a false positive event happens if an SYN packet matches in the `query Bloom` filters. If it is not an SYN packet, then the `route_table` with old versions can be used to forward this packet. syn_table stores entries that include [Match Key: syn_flag, version of the hit Bloom Filter; Action: resubmit, generate_digest]. Using the `syn_table` rather than syn_flag enables that we can flexibly change the entry and perform the corresponding operations according to the Match-Action results.

(4) **Route_table** maps each connection to a DIP server based on the weights which can be set by inserting different numbers of entries. The entry stored in `route_table` is [Match key:hash mod, version; Action value: DIP index]. The match fields include `hash mod` and `version` fields. `Hash mod` is obtained by `modulo` e of the 5-tuple hash result. Here e is the total number of `route_table` entries in a single version. The `version` field allows `route_table`

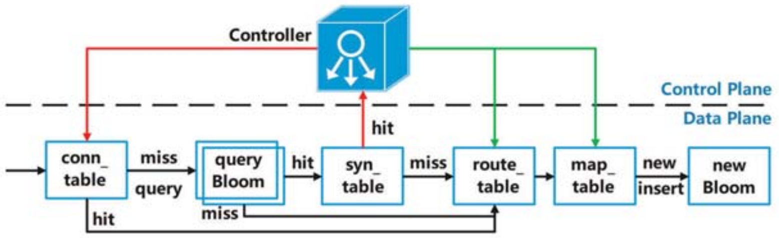


Fig. 4: Framework of Loom.

to record the route_table entries before updates. Thus, one route_table with different versions of entries can distinguish different DIP pools. The input of route_table are hash mod and version, and output is the index of DIP. With the classification ability of the query Bloom filters, the subsequent packets of old connections can be sent to the correct backend servers according to the route_table entries with an old version.

(5) **Map_table** is added to implement the mapping from DIP index to DIP. Because the action field also occupies a lot of SRAM on the hardware switch, we add a map_table at the end of all tables to compress the storage space for DIP. Using DIP index in the action fields of conn_table and route_table can effectively improve the utilization efficiency of SRAM. Adding map_table is also beneficial to simplifying the controller's handling of DIP update.

(6) **Controller** updates route_table when there is a pool update event and handles false positive events. Thus, false positive events will not affect the connection consistency. Besides, each pool update event will trigger the controller to perform a circulating update (§IV-B3).

Workflow of Loom. 1) A packet that hits conn_table can be directly sent to the corresponding backend server without passing through other tables. 2) Once a packet misses the conn_table, it queries query Bloom filters to obtain which version of route_table should be used. The two Bloom filters used for the query correspond to the past two versions of route_table. Whether a connection has already been established in the past can be determined by whether the two query Bloom filters are matched. 3) If the connection that the packet belongs to does not exist in these two query Bloom filters, we can infer that it is a new connection. Then the latest version of route_table can be used to forward the packet. This packet is then inserted into the new Bloom filter to record the new connection. 4) If this connection exists in one of the query Bloom filters, then syn_table will be used to determine whether a false positive event happens. 5) If a false positive event is detected by syn_table, the controller will get the 5-tuple of this connection. Then, the controller will insert an entry in conn_table. The following packets of this connection that may suffer false positive will never be checked in the query Bloom filters. 6) If no false positive event happens, then the packet will be sent to its backend server according to the corresponding version of route_table.

B. *Circulating Bloom filter*

1) Bloom filter: In Loom, the Bloom filter is used to answer whether a packet should be applied to an older version of route_table. It is proved that the Bloom filter can be constructed with two hash functions without any loss in the asymptotic false positive rate (e.g., gi(x)=h1(x)+ih2(x)) [32]. That is to say, new hash functions can be constructed through the addition and left shift operations in P4 [33].

2) Influence of false positive: The most obvious limitation of the Bloom filter data structure lies in possible false positive match, which means that a Bloom filter may produce a wrong result for elements that are not in the set [34].

First, we need to analyze the impact of false positive events on different kinds of connections. For an old connection, since there are already corresponding records in one query Bloom filter, all subsequent packets will hit the query Bloom filter and then be forwarded according to the route_table with the corresponding old versions. For a new connection, If the first packet, SYN packet, hits the query Bloom filters, which means that a false positive event happens. Then we can distinguish this new connection that suffers false positive by syn_table in Loom. After this connection is detected, the state of this special new connection will be added to the conn_table by the controller to prevent subsequent packets of the connection from entering the query Bloom filters for query. In order to guarantee a non-SYN packet being dispatched to the correct DIP even if it hits multiple query Bloom filters, we set the ASIC processing logic to forward only according to the first query bloom filter matched by the packet. In this way, the false positive problem could be addressed.

By controlling the number of elements in Bloom filters and the circulating update (§IV-B3), we can control the maximum False Positive Rate (FPR). The specific value of FPR can be adjusted by controlling the ratio of the storage space taken by the conn_table and the Bloom filters in the implementation. More details will be shown in §V-A.

3) Circulating update: The circulating update scheme for the Bloom filters is designed for two main goals. First, it is

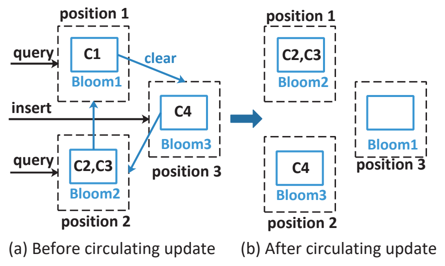


Fig. 5: Example of a circulating update.

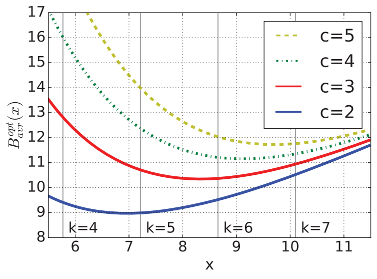


Fig. 6: Relationship between $x = \frac{m}{n}$ and the optimal number of bits occupied by each connection, $B_{avr}^{opt}(x)$.

used to handle the pool update events. After one server pool update, the new Bloom filter will become a query Bloom filter for classifying incoming connections, and the oldest Bloom filter will be cleared and reused to record the new connections with a new version value. Second, the circulating update of Bloom filters is used for maintaining normal FPR. Elements in a Bloom filter cannot be removed. However, the FPR will increase as the number of elements increases in a Bloom filter. Thus, when the number of elements in the new Bloom filter reaches a threshold, Loom will also execute the circulating update.

As shown in Figure 5, Loom uses three Bloom filters, Bloom 1, Bloom 2, and Bloom 3. They correspond to the `route_table` from the oldest version to the latest version. Before updating, Bloom 1 and Bloom 2 are used for query of connections, and Bloom 3 is used to store newly arrived connections. The entire update process is a cyclic shift of Bloom 1, Bloom 2, and Bloom 3. After a circulating update happens, Bloom 1 will clear its content and be placed at position 3, Bloom 2 will be placed at position 1, and Bloom 3 will be placed at position 2. This forms a complete circulating update. After the circulating update, an empty Bloom filter is obtained for storing new connections, while the old connections are saved in Bloom filters at position 1 and 2. As for the connections that are cleared from Bloom 1, the probability that these connections still keep alive is very low since a significant fraction of data center flows last under a few hundreds of milliseconds [35]. Besides, the probability of connections being broken can be further reduced by using the consistent hashing in `route_table`. This cyclic shift does not require multiple copy operations and only needs to change the pointer operated by each function, leading to negligible overhead. Note that since P4 does not support pointer operations, we add a flag to each Bloom filter in metadata to distinguish whether a Bloom filter is a query Bloom filter or not. The controller can change the flag to achieve circulating update.

The circulating update can effectively mitigate the impact of resource exhaustion attacks like SYN flood. When the SYN flood attacks cause a large number of new connections to be quickly inserted into the Bloom filter, the circulating update will be triggered frequently even without DIP updates. In such cases, SYN attacks without subsequent packets are removed from Bloom filters, and those normal flows with subsequent packets can continue to be recorded in the Bloom filters. Thus, the cleared normal connections that use the latest version of `route_table` can keep PCC.

V. Analysis and Implementation

In this section, we will analyze how many connections could be maintained and the optimal values of parameters in Loom. Then we describe the implementation of Loom based on the theoretical analysis results. At the same time, some additional constraints on the implementation were presented.

A. Number of connections

First, we analyze the compression efficiency of the connection states in Loom.

In the following analysis, the space occupied by syn_table, route_table, and map_table will be ignored, because they only occupy less than 0.5 MB of SRAM with 65535 DIPs. Several parameters are defined as follows:

- k represents the number of hash functions used by one Bloom filter.
- m is the number of bits taken by a Bloom filter, that is, the SRAM size occupied by a Bloom filter.
- n stands for the number of elements currently stored in a Bloom filter.
- c represents the total number of Bloom filters used in Loom.

Eq. (1) shows the theoretical value of the FPR of a single Bloom filter expressed with the variables k, m, and n [36].

$$FPR_1 = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-k\frac{n}{m}}\right)^k \quad (1)$$

There are total c Bloom filters in Loom. Thus, we can get that there are $(c-1)$ `query Bloom` filters and one `new Bloom` filter. The probability that no `query Bloom` filters generate false positive events is $(1 - FPR_1)^{c-1}$. Therefore, the total FPR can be expressed as

$$FPR_{tot} = 1 - (1 - FPR_1)^{c-1} \quad (2)$$

In Loom, the number of bits occupied by each connection, B_{avr}, can be expressed as Eq. (3)

$$B_{avr} = FPR_{tot} \times B_{conn} + (1 - FPR_{tot}) \times B_{bloom} \quad (3)$$

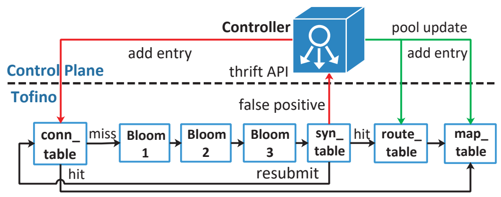


Fig. 7: Implementation framework of Loom on the programmable hardware switch.

where B_{conn} represents the bits occupied by each connection stored in conn_table, and B_{bloom} equals the ratio of m to n. Since B_{bloom} is significantly smaller than B_{conn}, B_{avr} will decrease as FPR_{tot} becomes smaller. To minimize FPR_{tot}, FPR_1 needs to be minimized. According to Eq. (1), FPR_1 has a minimum value when

$$k = ln2 \times \frac{m}{n}. \tag{4}$$

Assume that a 48-bit hash value is used in the match field of conn_table, and the action field takes up 16 bits. Then a connection takes up a total of 64 bits in the conn_table, that is $B_{conn} = 64$. Assume the value of B_{bloom} is x. Then $k = xln2$. Thus, the optimal number of bits occupied by each connection $B_{avr}^{opt}(x)$ can be expressed as Eq. (5).

$$B_{avr}^{opt}(x) = (x - 64)(1 - 0.6185^x)^{c-1} + 64 \tag{5}$$

Figure 6 depicts how $B_{avr}^{opt}(x)$ changes with different x and c. Since the derivative of this function is a transcendental function, the analytical solution of the minimum value point cannot be obtained. However, the numerical solution can be easily obtained. Furthermore, since k (the number of hash functions) is an integer and the reasonable values of x are quite limited, for a given c (the number of Bloom filters), we can find the optimal value of k and $x = \frac{m}{n}$ that minimize $B_{avr}^{opt}(x)$. As for the value of c, it represents a tradeoff between the compression efficiency and the connection consistency. Larger c causes an increase in FPR. Thus, smaller c could lead to better compression results. As shown in Figure 6, smaller c leads to smaller $B_{avr}^{opt}(x)$.

Taking $c = 3$ as an example, the optimal compression efficiency occurs when $k = 6$ or $x = 8.66$, and the corresponding average SRAM occupied by each connection is 10.37 bits. Thus, Loom can maintain about $\frac{50\text{MB}}{10.37\text{bit}} \approx 40.4$ million connections in theory. Taking $c = 2$ can have a better compression rate. However, c is too small to store different versions of connections, and thus the circulating update will clear more old connections. In order to ensure the reliability and high compression ratio of Loom, we take $c = 3$. In contrast, directly storing one IPv4 connection requires 120 bits, while the entire 5-tuple information of one IPv6 connection needs 312 bits. Note that the Bloom filter itself is a classifier that indicates which version of route_table is used, so the Bloom filters in Loom do not need an action field. In other words, Loom can store about $\frac{120}{10.37} \approx 11$ to $\frac{312}{10.37} \approx 30$ times more connections in the same SRAM space than those directly storing the entire 5-tuple of each connection. Besides, SilkRoad uses a 28 bits entry in ConnTable to store a connection [11]. Thus, Loom can achieve about $\frac{28}{10.37} \approx 2.7$x over it. The remaining SRAM can be used for other important switch functionality, such as routing and tunneling [37]. As a load balancer, SRAM is the most important and bottleneck resource that affects the number of connections that can be maintained, other resources such as hash units, ALUs are enough in our system.

B. *Implementation of Loom*

We implement a prototype system using the P4 language [20] on the Edgecore Wedge100BF-65X [19] programmable hardware switch. The implementation framework of Loom is shown in Figure 7. Due to the hardware resource and logical limitations on the hardware switch, we made some modifications to our design. The details are as follows:

(1) The **running process** is adjusted to fit into the limitations of the hardware switch. From an abstract logical point of view, there is no change in the entire process. But because of the pipeline limitations of the hardware switch, we add a resubmit operation to complete the insert operation of new connections. After adding the resubmit operation, the workflow of Loom has to be modified. Specifically, upon arrival of a packet, Loom will go through all the Bloom filters and keep all the query results. The packet that needs to be inserted into one of the Bloom filters will be resubmitted and inserted into the corresponding Bloom filter.

Note that Loom only resubmits some packets. The upper bound of the proportion of packets that need to be resubmitted is the reciprocal of the average connection length. Besides, this is done within the logical framework provided by the hardware switch. Thus, there is almost no overhead. Meanwhile, since the hardware switch is based on the pipeline architecture of P4 switches, querying all Bloom filters for every packet does not cause an increase in processing latency. Because as long as the application is adapted to the logic of the P4 pipeline, there will be no significant difference in processing latency [11], [38].

(2) The **ratio of the storage space** occupied by Bloom filters and conn_table also needs to be adjusted. Since not all SRAM in a hardware switch can be used to implement Bloom filters, some SRAM can only be used to implement conn_table. In other words, Eq. (3) needs to be subject to inequality (6), where r is a hardware-related parameter that indicates the ratio of space occupied by conn_table to Bloom filters. Thus, the space used by Bloom filters is further reduced with larger r. In our implementation, $r = \frac{2}{3}$. If $c = 3$, then according to Eq. (3) and (6), we will choose $x = 7.2$ and $k = 5$ to get an optimal B_{avr}^{opt}.

$$\frac{FPR_{tot} \times B_{conn}}{(1 - FPR_{tot}) \times B_{bloom}} > r \tag{6}$$

(3) The RPC framework, **thrift** [39], is used for communication between the controller and the Tofino data plane. The controller will get the false positive events generated by

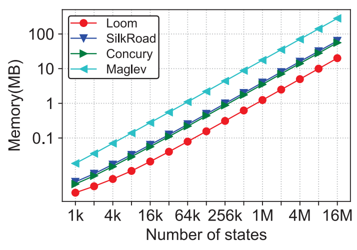


Fig. 8: Memory cost to store different number of states.

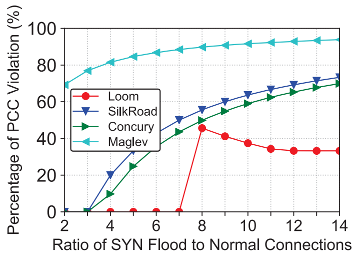


Fig. 9: Normal 5-tuples take up 25% space

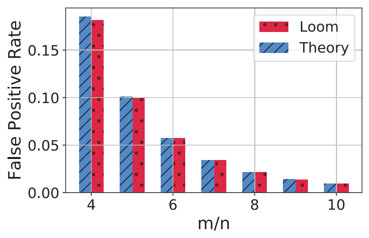


Fig. 10: FPR difference between theoretical analysis and experimental results.

`syn_table` through the thrift API, process the information carried in it, and then add the corresponding entries into `conn_table` through the thrift API. Once a DIP pool update event happens, the `route_table` and `map_table` need to be changed. This is also conducted by the controller through the thrift API. Besides, the circulating update also utilizes the thrift API to clear the oldest `query Bloom` filter and set it as the `new Bloom` filter to store new connections.

VI. Evaluation

In this section, we will evaluate the performance of Loom based on two types of evaluations:

1) *Algorithm micro-benchmark* based on the open-source and reproducible code provided by [8]. We compare Loom with three existing typical stateful load balancer algorithms: Maglev [6], SilkRoad [11], and Concury [8]. The server we used to implement Loom and the open-source code of the three compared schemes has Intel(R) Xeon(R) Gold 6230 CPU, 2.10GHz, 27.5M L3 cache shared by 8 logical cores, and 16GB RAM. Besides, we use the linear feedback shift register that can generate more than 200M states (5-tuples) per second to generate uniformly different states.

2) *Hardware-testbed evaluation* from four aspects: balance, FPR, average SRAM usage, and impact of SYN flood. Our testbed consists of several Linux containers as servers and an Edgecore Wedge100BF-65X hardware switch running our P4 code as a load balancer.

In all scenarios, $c = 3$ and $k = 5$ unless otherwise specified.

A. Algorithm micro-benchmark results

*1) **Memory usage**:* In this experiment, we send different numbers of 5-tuples and record the memory usage of these load balancers. Figure 8 shows the memory usage of SilkRoad, Maglev, Concury, and Loom to store different numbers of 5-tuples. The number of backend servers is 32. The experimental results are displayed on a logarithmic axis. As the number of states increases, we found that the storage space occupied by SilkRoad, Maglev, Concury is about 3.2, 14, 2.8 times of Loom, respectively. Loom only needs less than 10MB to store 8.39 million concurrent 5-tuples. Compared with the state-of-art stateful load balancers, Loom needs the least memory space to store the same number of 5-tuples.

*2) **PCC under SYN flood attacks**:* An SYN flood is a typical and severe resource exhaustion attack on stateful load balancers. We will show that Loom has a good defense ability against SYN flood attacks. There are 32 backend servers in this test. In the beginning, let the DIP pool consist of the first 16 servers. After sending some SYN flood 5-tuples and normal 5-tuples, the DIP pool is updated to the last 16 servers. Then, we re-sent the normal 5-tuples. A 5-tuple mapped to different backend servers is recorded as a connection that violates PCC. In this case, we can clearly obtain the total number of PCC violation connections caused by an SYN flood attack.

Figure 9 shows the number of PCC violation connections with different load balancer mechanisms. The total number of normal connections established before DIP pool updates equals 25% of the amount that can be stored in SilkRoad. The results show that Loom can reduce the PCC violation probability by up to 40%, 37%, and 60% compared to SilkRoad, Concury, and Maglev. This is because Loom has the circulating update mechanism and keeps different versions of Bloom filters and corresponding `route_table`. In contrast, Maglev uses a hash table to store connection states and consistent hashing to evenly distribute traffic. SilkRoad uses ConnTable to store connection states with different versions and Multi hash tables to distribute traffic. Concury uses OthelloMap to store connection states and weighted randomizer to distribute traffic. After the DIP pool update, the connections that are not stored in these load balancers will have to be sent to the last 16 servers and thus violate PCC.

B. Hardware-testbed evaluation results

*1) **False positive rate**:* Firstly, we show that the FPR of the Bloom filters implemented on the hardware switch is in line with the theoretical analysis results presented in §V-A. The `tcpreplay` tool is used to replay a `pcap` file containing a large number of SYN packets with different 5-tuple. In this way, a large number of connections will be injected into the load balancer to achieve a specific $\frac{m}{n}$ of the Bloom filter. Then some TCP connections will be generated and the number of false positive events reported by `syn_table` will be counted on the controller to get the FPR_{tot}^{exp} value at a specific $\frac{m}{n}$.

The results are shown in Figure 10. It can be seen that the experimental results on the hardware switch are consistent with theoretical analysis results. Furthermore, the number

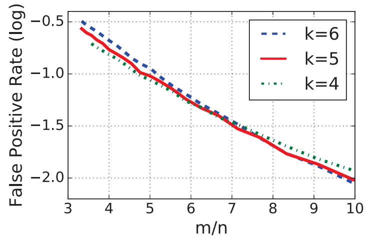


Fig. 11: Effect of k on FPR with different ratio of m to n.

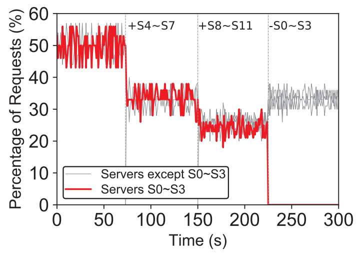


Fig. 12: Percentage of requests received by each server with DIP pool updates.

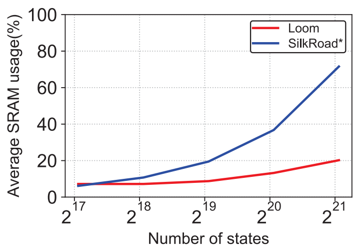


Fig. 13: Average SRAM usage to store different number of states in hardware P4 switch.

of connections that the system can store can be estimated according to Eq. (3) and FPR_{tot}^{exp}. By calculation, we obtain that the state of a connection consumes an average of 10.72 bits in our testbed. Thus, *Loom can store nearly* 39.2 *million connections using only* 50 *MB SRAM space. It achieves 11x to 29x more compression of storage space than those directly storing the entire 5-tuple information of connections and 2.6x more than those with per-entry compression [11].*

The effect of k on FPR with different $\frac{m}{n}$ values is shown in Figure 11. The experimental setup is the same as that in Figure 10. As the number of hash functions increases, FPR decreases when $\frac{m}{n}$ is smaller than 7, but increases when $\frac{m}{n}$ is larger than 7. Given a fixed $\frac{m}{n}$, the number of connections that can be supported in Loom will be maximized with the minimum FPR. From the experimental results, we can see that when $\frac{m}{n} = 7$, using $k = 5$ hash functions in each Bloom filter leads to the best performance. This is the same as the theoretical analysis result in §V-A.

2) ***Server pool update****:* In this subsection, the ability of balancing requests and handling pool update events will be evaluated. A Flask-based [40] web service runs on each DIP server. Flask [40] is a micro web framework for rapid development and deployment of services written in Python. This web service converts the format of the image file contained in the request into GIF and then sends it back to clients. Clients send post messages through python's request library to generate TCP connections and traffic.

This experiment consists of 16 backend servers. We count the number of newly received requests per second on each server. At the same time, several pool update events are performed, and the number of broken connections is counted to evaluate the ability of maintaining connection consistency when there is a pool update event under Loom.

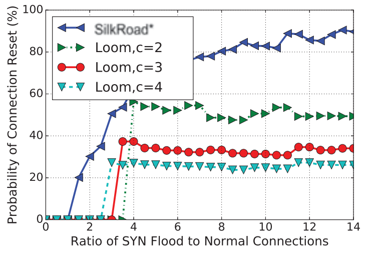
The results are shown in Figure 12, where the x-axis stands for time and the y-axis represents the percentage of new requests received by servers in one second. In the beginning, severs $S_0 - S_7$ are used to provide services. Then servers $S_8 - S_{15}$ are added into the DIP pool in sequence. Finally, servers $S_0 - S_3$ are removed from the DIP pool. Figure 12 shows the workload variations. The percentage of requests received by $S_0 - S_3$ is highlighted using red color. It can be seen that at the beginning, $S_0 - S_3$ receive about 50% of new requests per second since there are totally 8 servers $S_0 - S_7$ in the DIP pool. After $S_8 - S_{11}$ are added, the percentage received by $S_0 - S_3$ and $S_4 - S_7$ drops to about 33%. At time 150s, $S_8 - S_{15}$ is added, then the workload on each cluster decreases to about 25%. Finally, $S_0 - S_3$ are removed from the DIP pool at 250s, $S_0 - S_3$ no longer receive any new requests. New requests are evenly distributed to the remaining servers.

This shows that Loom can evenly distribute requests to all backend servers even when DIP pool updates occur. Besides, we found that no connection was broken off or reset during this experiment, which means that the consistency of each connection was maintained.

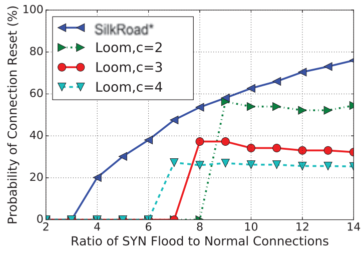
3) ***Average SRAM usage****:* In this subsection, we compare the average SRAM usage of Loom and SilkRoad* on hardware switch. Since SilkRoad is not open-source, we do our best to achieve SilkRoad's hardware logic called SilkRoad* which only uses `conn_table` without Bloom filters. The average SRAM usage is defined as the sum of percentage SRAM usage in every pipeline divided by 12 pipelines. We firstly set the entry number of conn_table in SilkRoad*, and the entry number of conn_table and Bloom filter size in Loom to support different numbers of connections. Then we use the visualization function in our hardware P4 switch to record the average SRAM usage. Figure 13 shows, the compression rate of Loom grows with the increase of states. Loom achieves 2.7x more compression of storage space than SilkRoad* when the number of states is more than 1 million, which is in accord with the theory results.

4) ***Impact of SYN flood****:* The method of this experiment is similar to the previous benchmark (§VI-A2), the difference is that we use a hardware testbed and real TCP connections.

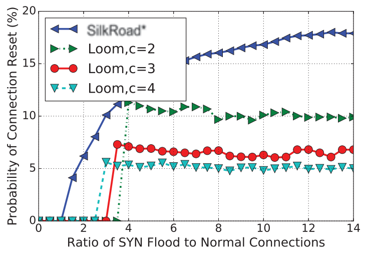
We know the number of bits a connection needs in both SilkRoad* and Loom. Thus, we inject different numbers of connections into them to make them occupy the same SRAM size. The total number of normal connections established before the DIP pool update equals 50% of the amount that can be stored in `conn_table` of the SilkRoad* in Figure 14(a) and Figure 14(c). And we change this ratio to 25% in Figure 14(b) and Figure 14(d). Besides, normal connections in Figure 14(a) and Figure 14(b) are all long flows that will not finish until the simulation ends. In Figure14(c) and Figure



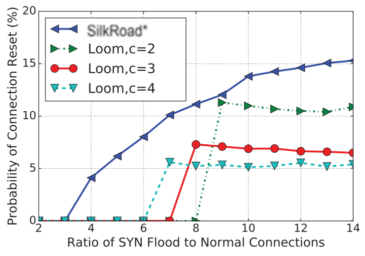
(a) Normal connections take up 50% space and all connections are long flows.



(b) Normal connections take up 25% space and all connections are long flows.



(c) Normal connections take up 50% space and connections are consisted of 80% short flows and 20% long flows.



(d) Normal connections take up 25% space and connections are consisted of 80% short flows and 20% long flows.

Fig. 14: Percentage of broken connections after DIP pool updates under an SYN flood attack.

14(d), the traffic is mixed of 80% short flows (less than 10 Kbytes) and 20% long flows to reflect more realistic traffic load in data centers [41], [42].

As shown in Figure 14(a), Loom can reduce the connection reset probability by up to 35%, 49%, and 57% when c equals 2, 3, and 4 compared to SilkRoad*, respectively. This is because Loom compresses the state of connections and can store more connections than SilkRoad*. Thus, given the same amount of SYN flood packets, fewer connections are reset in Loom. Besides, the increase of c causes Loom to reset fewer connections since more `query Bloom` filters are used to store the information of old connections. However, as c increases, Loom is more susceptible to SYN flood attacks due to reduced compression efficiency. Furthermore, for a fixed c, the percentage of broken connections is stable as the SYN flood attack becomes more severe in Loom. This is because the circulating update will make most of the normal connections be dispatched to their correct DIP servers even with the increase of SYN flood packets (§IV-B3).

When the normal connections established before the DIP pool update only take 25% `conn_table` space of SilkRoad*, connection reset happens with more SYN flood packets since more memory space can accommodate SYN flood packets. However, the connection reset probability is almost the same as that in Figure 14(a) given a fixed c. This is because the proportion of normal connections stored in each Bloom filter is almost identical with the same c. Thus, once a Bloom filter is cleared, almost the same proportion of normal connections will be reset.

At last, Figure 14(c) and Figure 14(d) illustrate that the change in flow size distribution decreases the connection reset probability by about 80% compared with Figure 14(a) and Figure 14(b), respectively. This is because short flows finish quite quickly. After the DIP pool update, only part of the 20% long flows suffer from connection reset.

VII. Conclusion

In this paper, the design and implementation of a layer-4 load balancer, Loom, are presented. By using a classifier and maintaining multiple hash functions, Loom does not need to keep per-connection state directly like existing stateful load balancers. It also does not offload state management burden to backend servers as some recently proposed stateless load balancers do. By designing a state compression structure based on the Bloom filter and a circulating update of multiple Bloom filters, Loom supports more concurrent connections and keeps the consistency of connections. Besides, it is more tolerant to SYN flood attack. Loom is implemented in a hardware P4 switch. Experimental results indicate that Loom could maintain as many connections as theoretical analysis shows and is more resistant to SYN flood attacks.

Acknowledgment

We gratefully appreciate the anonymous reviewers and our Shepherd, Timothy Wood, who helped us improve the quality of this paper. This work is supported in part by National Natural Science Foundation of China (NSFC) under Grant No. 61872401, Fok Ying Tung Education Foundation under Grant No. 171059, and Huawei collaboration 20201124006.

References

[1] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "VL2: A Scalable and Flexible Data Center Network," in *ACM SIGCOMM*, 2009.

[2] M. Al-Fares, A. Loukissas, and A. Vahdat, "A Scalable, Commodity Data Center Network Architecture," in *ACM SIGCOMM*, 2008, pp. 63–74.

[3] "A10 Networks ax Series." [Online]. Available: http://www.a10networks.com/

[4] "F5 Load Balancer." [Online]. Available: https://f5.com/

[5] 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," in *ACM SIGCOMM*, 2013.

[6] 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 *USENIX NSDI*, 2016.

[7] Y. Yu, X. Li, and C. Qian, "SDLB: A Scalable and Dynamic Software Load Balancer for Fog and Mobile Edge Computing," in *ACM SIGCOMM Workshop on Mobile Edge Communications*, 2017.

[8] S. Shi, Y. Yu, M. Xie, X. Li, X. Li, Y. Zhang, and C. Qian, "Concury: a fast and light-weight software cloud load balancer," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020, pp. 179–192.

[9] 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," in *ACM SIGCOMM*, 2015.

[10] R. Wang, D. Butnariu, J. Rexford *et al.*, "OpenFlow-Based Server Load Balancing Gone Wild," *Hot-ICE*, 2011.

[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 *ACM SIGCOMM*, 2017.

[12] N. Handigol, S. Seetharaman, M. Flajslik, N. McKeown, and R. Johari, "Plug-n-Serve: Load-balancing Web Traffic Using OpenFlow," in *ACM SIGCOMM*, 2009.

[13] Olteanu, Vladimir, A. Agache, A. Voinescu, and C. Raiciu, "Stateless Datacenter Load-Balancing with Beamer," in *USENIX NSDI*, 2018.

[14] "Barefoot's Tofino Chip and P4 Could Replace Load Balancers," https://www.sdxcentral.com/articles/news/barefoots-tofino-chip-and-p4-could-replace-load-balancers/2017/10/.

[15] "Barefoot Scores Tofino Deals with Alibaba, Baidu, and Tencent," https://www.sdxcentral.com/articles/news/barefoot-scores-tofino-deals-with-alibaba-baidu-and-tencent/2017/05/.

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

[17] A. J. Taveira, L. Saino, L. Buytenhek, and R. Landa, "Balancing on the Edge: Transport Affinity without Network State," in *USENIX NSDI*, 2018.

[18] B. H. Bloom, "Space/Time Trade-offs in Hash Coding with Allowable Errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[19] "Edgecore Wedge 100BF-65X." [Online]. Available: https://www.edge-core.com/productsList.php?cls=1&cls2=180

[20] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming Protocol-Independent Packet Processors," in *ACM SIGCOMM*, 2014, pp. 87–95.

[21] "Array Networks." [Online]. Available: https://www.arraynetworks.com/

[22] "Load Balancer.org." [Online]. Available: https://www.loadbalancer.org/

[23] "Katran: A High Performance Layer 4 Load Balancer." [Online]. Available: https://github.com/facebookincubator/katran

[24] M. Zhang, G. Li, S. Wang, C. Liu, and J. Wu, "Poseidon: Mitigating volumetric ddos attacks with programmable switches," in *Network and Distributed System Security Symposium*, 2020.

[25] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang, "Congestion Control for Large-Scale RDMA Deployments," in *ACM SIGCOMM*, 2015.

[26] R. Gandhi, Y. C. Hu, C.-K. Koh, H. H. Liu, and M. Zhang, "Rubik: Unlocking the power of locality and end-point flexibility in cloud scale load balancing." in *USENIX Annual Technical Conference*, 2015, pp. 473–485.

[27] "XPliant© CNX780XX/CNX680XX Family," https://cavium.com/xpliant-ethernet-switch-xp60-and-xp70-family.html.

[28] 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 *USENIX NSDI*, 2020, pp. 667–683.

[29] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary Cache: A Scalable Wide-area Web Cache Sharing Protocol," *IEEE/ACM Transactions on Networking (TON)*, vol. 8, no. 3, pp. 281–293, 2000.

[30] M. A. Bender, M. Farach-Colton, R. Johnson, R. Kraner, B. C. Kuszmaul, D. Medjedovic, P. Montes, P. Shetty, R. P. Spillane, and E. Zadok, "Don't Thrash: How to Cache Your Hash on Flash," *VLDB*, vol. 5, no. 11, pp. 1627–1637, 2012.

[31] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo Filter: Practically Better Than Bloom," in *ACM CoNext*. ACM, 2014, pp. 75–88.

[32] A. Kirsch and M. Mitzenmacher, *Less Hashing, Same Performance: Building a Better Bloom Filter*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, vol. 4168, pp. 456–467. [Online]. Available: http://dx.doi.org/10.1007/11841036_42

[33] "Open-source P4 implementation of features typical of an advanced L2/L3 switch," https://github.com/p4lang/switch.

[34] A. Broder and M. Mitzenmacher, "Network Applications of Bloom Filters: A Survey," *Internet Mathematics*, vol. 1, no. 4, pp. 485–509, 2004.

[35] 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.

[36] P. Bose, H. Guo, E. Kranakis, A. Maheshwari, P. Morin, J. Morrison, M. Smid, and Y. Tang, "On the false-positive rate of bloom filters," *Information Processing Letters*, vol. 108, no. 4, pp. 210–213, 2008.

[37] R. Cohen, M. Kadosh, A. Lo, and Q. Sayah, "Lb scalability: Achieving the right balance between being stateful and stateless," *arXiv preprint arXiv:2010.13385*, 2020.

[38] Z. Hang, M. Wen, Y. Shi, and C. Zhang, "Programming protocol-independent packet processors high-level programming (p4hlp): Towards unified high-level programming for a commodity programmable switch," *Electronics*, vol. 8, no. 9, p. 958, 2019.

[39] K. Rakowski, *Learning Apache Thrift*. Packt Publishing Ltd, 2015.

[40] M. Grinberg, *Flask Web Development: Developing Web Applications with Python*. " O'Reilly Media, Inc.", 2018.

[41] T. Benson, A. Akella, and D. A. Maltz, "Network Traffic Characteristics of Data Centers in the Wild," in *ACM SIGCOMM Conference on Internet Measurement*, 2010, pp. 267–280.

[42] T. A. Benson, A. Anand, A. Akella, and M. Zhang, "Understanding Data Center Traffic Characteristics," in *ACM Workshop on Research on ENterprise networking*, 2009, pp. 65–72.