Maat: A Fair and Scalable Layer-4 Load Balancer With Per-Connection-Consistency
Paper id 216

***Abstract*—Layer-4 load balancers play a pivotal role in data centers by efficiently distributing incoming traffic across thousands of servers. These load balancers need to meet two key requirements: load balancing fairness, which ensures a uniform distribution of incoming traffic across servers, and per-connection-consistency (PCC), which guarantees that all packets belonging to the same connection are forwarded to the same server. Existing load balancers face a dilemma: they either sacrifice fairness to significantly reduce PCC violations, failing to meet both requirements, or they adopt complex scheduling mechanisms, which increase deployment and operational costs. In this paper, we introduce Maat, a load balancer designed to achieve fair load distribution and PCC while seamlessly integrating into existing data center infrastructures. Maat introduces a novel scheduling method called the power of one random choice. This method improves the utilization of all available servers, greatly enhancing load distribution fairness while reducing PCC violations. Furthermore, Maat can fully guarantee PCC by utilizing counting Bloom filters with negligible memory overhead. We implement Maat on a Tofino switch, and our experimental results show that the packet processing overhead of Maat is acceptable. Compared to other Layer-4 load balancers, Maat improves load balancing fairness by up to 74.42% and reduces flow completion time by 10.58%, while fully guaranteeing PCC at the cost of minimal memory consumption.**

***Index Terms*—layer-4 load balancer, fairness, per-connection-consistency, programmable switch**

I. Introduction

In recent years, there has been a noticeable trend for businesses and enterprises to migrate services such as online searches and online games to cloud environments [1]. This shift has driven the deployment of massive data centers to meet the growing demand for cloud services. However, simply providing the abundant link and server resources [2] is no longer sufficient to meet the rapidly growing user needs. It can even be counterproductive if incoming traffic is not allocated resources adequately for processing. Recognizing the need for more reliable and scalable services, operators have begun to deploy load balancing mechanisms within data centers. For instance, a Layer-4 load balancer (L4 LB) ensures that incoming traffic is distributed fairly across all reachable backend servers (DIPs). The goal of L4 LB is to maximize resource utilization to meet the growing needs of the business.

Approximately half of data center traffic requires L4 LB for processing [3]. Inefficient handling of this traffic by L4 LB can have severe consequences for the data center. Potential outcomes include some servers being underutilized, resulting in resource wastage, while others become overloaded, leading to downtime or even service interruptions. These issues can further cause a sharp decline in user experience [4]. Therefore, in Layer-4 load balancing, fairness should be treated as a first-class citizen.

Effective management of incoming traffic by L4 LB encounters two significant challenges. The first challenge arises from skewed traffic distribution. Applications commonly utilize short connections for synchronization and long connections for massive data transfer, resulting in a long-tail distribution where a few flows carry the majority of the traffic [5]. Improper handling of L4 LB in such scenarios can easily lead to uneven load distribution among servers [6]. The second challenge stems from the limitations imposed by L4 LB when dealing with stateful protocols, such as TCP and QUIC [7]. These protocols are usually connection-oriented to improve communication reliability. Consequently, all packets of a flow must be processed by the same server to maintain *per-connection-consistency* (PCC) [4]. PCC violations can cause connection interruptions and increase the end-to-end delay during connection re-establishment, significantly impacting service quality. To address these challenges, previous works have focused on maintaining PCC [8]–[11]. However, some studies have revealed that these schemes often reduce PCC violations at the expense of load distribution fairness [12], contradicting the fundamental purpose of L4 LB.

In general, LB primarily enhances fairness in two main ways. The first method involves adjusting the scheduling granularity, commonly utilized in L3 LB [6]. Operators often use flowlet-level [13]–[15] or packet-level [16] granularity to improve fairness. However, in L4 LB, since all packets of the same connection must be forwarded to the same server to maintain PCC, the granularity is limited to the flow-level. The second approach is to choose different scheduling methods, such as hashing [4], [10], round-robin [17], [18], the least congestion [13], [19], and so on. However, because of PCC limitations, L4 LB must maintain the mapping between flows and servers in dynamic network environments [11]. Consequently, existing solutions typically rely on hashing, which is usually difficult to achieve good fairness, or requires modifications to the TCP/IP protocol stack in terminal devices to select other advanced scheduling methods. Although the latter can bring better fairness, it often has poor scalability and is difficult to deploy directly in existing data centers.

Based on the above discussion, we believe that L4 LB should meet the following requirements: 1) **Fairness**: fully

utilize the available resources of all servers; 2) **PCC**: packets belonging to the same connection are forwarded to the same server; 3) **Scalability**: no changes to the end host or protocol stack during deployment. To achieve these goals, this paper proposes Maat, an L4 LB implemented on a programmable switch. Maat proposes a novel scheduling method termed the power of one random choice. Specifically, to enhance fairness, Maat processes each flow by selecting one of two servers for forwarding, with one server selected through hashing to maintain PCC. To further improve fairness, Maat also employs a finer-grained approach to capture and analyze the status information of incoming traffic directly, making the selection more accurate.

Using the power of one random choice while always maintaining PCC on Maat is challenging. To address PCC violations caused by DIP pool updates (i.e., removal or additional of servers), we adopt Othello hashing, a minimum perfect hash [20], as our hashing function for server selection. It is typically used for load balancing with low memory overhead and no false hits [21]. Additionally, the power of one random choice also cause PCC violations due to random selection. We transform this problem into a membership set problem and introduce a counting Bloom filter [22] to ensure PCC. By cleverly combining Othello hashing with counting Bloom filters, Maat can fully guarantee PCC with negligible memory even under heavy load.

To summarize, the key contributions of Maat are as follows:

- We show that existing scheduling methods often face a dilemma: they either sacrifice fairness to guarantee PCC or cannot be directly deployed in contemporary data centers. Consequently, we propose a novel scheduling method named power of one random choice, capable of satisfying the requirements of L4 LB for both fairness and PCC simultaneously.
- We design and implement Maat, an LB that meets the two key requirements of L4 LB. Maat is easy to implement in programmable hardware (e.g., Tofino switch [23]) and can be deployed without any changes to the end host or TCP/IP protocol stack.
- We evaluate Maat extensively in hardware testbed and conduct simulations in scenarios with realistic traffic patterns. Our experimental results show that Maat achieves significant improvement in fairness by 70-80% compared to existing L4 LB under heavy load.

In §II of this paper, we first summarize the advantages and disadvantages of previous L4 LB related work, then introduce the design motivation behind Maat, and finally, we present some background algorithms. Subsequently, we introduce the overall architecture and design details of Maat in §III. Next, the implementation and experimental results of Maat are described in §IV, and finally, we conclude Maat in §V.

II. BACKGROUND AND MOTIVATION

Datacenter operators assign a virtual IP (VIP) address to each service they operate, and this virtual IP address is generally assigned to the L4 LB. Each VIP is associated with a set of servers (DIP pool) providing that service. The purpose of this paper is to design a simple and practical L4 LB scheme that evenly distributes incoming traffic across all available DIPs. In this section, we begin by presenting the advantages and disadvantages of both load-agnostic and load-aware L4 LB. Subsequently, we discuss the design motivation of Maat. Finally we introduce some background algorithms.

A. Previous work

Load-agnostic L4 LB: Load-agnostic load balancers are characterized by relying on simple hash calculations[1] to distribute incoming traffic among backend servers. Since these LBs do not consider the current load status of the servers, they often result in load imbalances. Research has shown that hash-based methods can suffer up to 30% load imbalance [8].

Ananta [3] is a L4 LB that relies on ECMP to forward traffic. Due to its slow packet processing speed, Duet [10] offloads part of the workload to the switch to alleviate the performance bottleneck that exists in Ananta. Silkroad [4] use programmable switches to further improve throughput, but these solutions all use ECMP to forward traffic, and PCC violations are prone to occur when the DIP pool updates. Google proposes Maglev [8] to effectively alleviate this problem through consistent hashing. Later, Faild [11] and Beamer [9] also adopt consistent hashing as their load balancing scheduling method.

However, consistent hashing may introduce false hits during connection lookups due to the use of digests rather than complete state information. The occurrence of false hits may cause the L4 LB to forward packets to the wrong DIP, resulting in PCC violations. To address these problems, some works proposed applying Othello hashing to L4 LB [21]. For instance, Concury [24] and SDLB [25] are designed for cloud data centers and mobile edge computing, respectively. Due to the false-hit freedom characteristics of this hash algorithm, L4 LB can fully maintain PCC. Although load-agnostic LBs have made great progress in maintaining PCC. However, fairness as a first-class citizen in L4 LB has not received enough attention, and some schemes even ensure PCC at the expense of fairness.

Load-aware L4 LB: Research in recent years has also realized that it is wrong for load-agnostic load balancers to sacrifice fairness to maintain PCC [1], [12], [26]. As a result, some works have begun to strive to enhance fairness while maintaining PCC. The mainstream method is to adjust the weight of different servers through real-time network status (e.g., server status, traffic status), so that incoming traffic has a greater probability of selecting a more suitable server, thereby alleviating the load imbalance based on the hash scheme.

Server status-based methods collect information such as CPU and memory usage to guide forwarding strategies. However, they face three main challenges: 1) This type of solution is difficult to cope with the challenge of device heterogeneity and can easily cause L4 LB to make wrong forwarding

[1]Hash algorithms refer to algorithms such as ECMP, consistent hashing, and minimum perfect hashing.

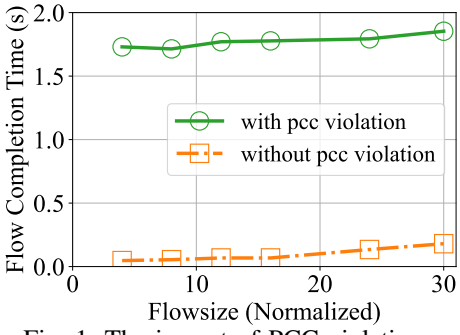


Fig. 1: The impact of PCC violations.

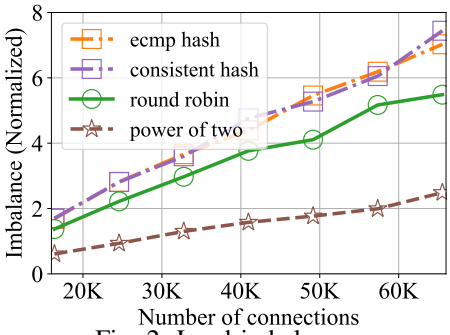


Fig. 2: Load imbalance.

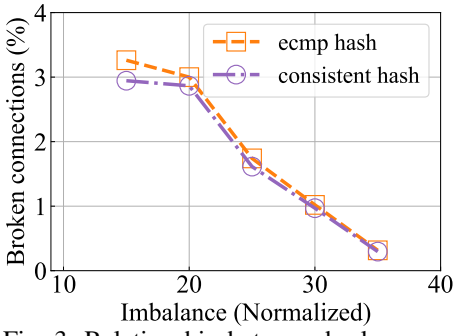


Fig. 3: Relationship between broken connections and load imbalance.

decisions, thus damaging fairness [26]. Here, device heterogeneity [27] refers to the presence of multiple different devices within the same service. For server operators, a single service may be provided by multiple machines of various models [28]. Moreover, with the rise of cloud computing [29] and network function virtualization (NFV) [30], the device heterogeneity faced by contemporary cloud data centers may be more complex. 2) Server status-based solutions usually require additional high-performance equipment to implement complex interactions between LB and servers. For example, Spotlight [6] requires specifying an additional server (Intel i9 7900x with 64 GB RAM) as a dedicated controller for traffic management. 3) These methods also introduce security risks due to the frequent interactions required between the L4 LBs and servers to obtain accurate, real-time status updates. Third-party attackers can easily use sniffing attacks [31] to obtain server status information from frequent interactions, thereby conducting more targeted DDoS attacks [32], causing some servers to go down or even the entire service to collapse. In summary, server state-based solutions not only face huge scalability challenges but also have difficulty handling potential security issues.

Another approach involves the load balancer directly collecting traffic status, which avoids complex interactions with servers, addresses potential security issues, and provides good scalability. However, there are also some problems with traffic state-based methods. For example, in LBAS [26], the scheduling policy relies on the number of active connections on the current server, ignoring the different effects of flow size on server load. Therefore, this approach runs the risk of forwarding incoming traffic to inappropriate servers. Although some methods [33] are aware of this problem, they still use a coarse-grained approach to collect traffic status information, which also leaves room for improper traffic allocation. To solve these problems, Maat introduces a more fine-grained traffic statistics method. With packet-level traffic statistics, incoming traffic can be assigned more accurate servers for better fairness.

B. Motivation

PCC violations significantly increase the tail latency. Maintaining PCC in L4 LB is crucial, as our experiments demonstrate that it has a significant impact on performance. We deploy HTTP service on three servers (1 client, 2 servers) and use a Tofino switch as the L4 LB. During this experiment, we generate workloads with increasing flow sizes and perform DIP pool updates. The experimental results in Fig. 1 show that the FCTs with PCC violations are 10.28 times higher than those without PCC violations. This indicates that PCC violations severely affect the processing of incoming traffic, especially latency-sensitive traffic, potentially leading to unacceptable service level agreement (SLA) violations. From the experiment, it is clear that maintaining PCC is a prerequisite for L4 LB to improve fairness. However, we must ensure that load balance is not sacrificed to maintain PCC [34].

Hash-based L4 LB cannot spread incoming traffic evenly. We compare ecmp/consistent hashing and other scheduling strategies for their ability to evenly distribute incoming traffic. The workload is generated based on the traffic distribution of Web search [35]. The experimental results are shown in Fig. 2. Among the four commonly used strategies, the power of two choices exhibits significant advantages. Specifically, it performs 2.83 times better than ecmp/consistent hashing and 2.21 times better than round-robin. However, the limitations of maintaining PCC make it challenging to directly implement methods like the power of two choices or round-robin in L4 LB environments. Cheetah proposes a method that requires modifying the TCP/IP protocol stack by encoding the selected server's identifier into a cookie added to all the packet headers.

Hash-based L4 LB cannot guarantee PCC and load balance at the same time. We conduct a simple experiment to evaluate whether these hash-based L4 LB can simultaneously achieve the two goals of enhancing fairness and maintaining PCC. In this experiment, we use an imbalance threshold as input and removed/added the server from the DIP pool when the server load is above/below the threshold [12]. We use the traffic distribution of Web search to generate 100K connections. Fig. 3 illustrates the relationship between fairness and the ratio of broken connections. It is worth noting that even when the load imbalance (normalized) reaches 30, the ratio of broken connections remains around 1%. This demonstrates that L4 LB faces a conflict between enhancing fairness and ensuring PCC. In existing hash-based schemes, a common approach is to sacrifice load balance [8], [9], which goes

against the original purpose of load balancing.

In brief, these experiments demonstrate that existing schemes cannot meet the three requirements of L4 LB (fairness, PCC, scalability). This motivates us to propose a new L4 LB scheme.

C. Background algorithms

Power of two choices is a commonly used scheduling method in load balancing, and in Figure 3 we can see that it can achieve better fairness. This algorithm, originally proposed by David G. Andersen and Michael Kaminsky [36], significantly enhances load balancing performance in supermarket models with a single input queue and multiple output queues using a limited number of choices (e.g., d = 2) [37].

However, in L4 LB, two random choices make maintaining the mapping between connections and servers more complex [34], [38]. Therefore, most existing solutions still use hash functions to select servers because the hash function helps mitigate PCC violations. Unlike the existing L4 LB, Maat uses a novel scheduling method inspired by the power of two choices termed the power of one random choice. The basic idea is to compare the load of one hash selected server and one randomly selected server and choose the one with less load to forward traffic.

Othello hashing is a minimal perfect hashing algorithm [39], [40] that supports forwarding information base and load balancing in programmable networks, including two arrays as its data plane, and a construction program in its control plane, as the interaction protocols of the two planes [24]. Since the Othello hashing has the characteristics of fast query speed and no false hits [25], which can help us maintain PCC. So we use it as the hash function of the power of one random choice. We use the example in Fig. 4 to demonstrate the operation of constructing and querying Othello.

Construction in the Othello control plane. In Fig. 4, two arrays A and B are constructed with m_a and m_b elements, respectively. Each array element is an l-bit value. In this example, $l = 2$ and assume $m_a = m_b = 5$ for better illustration. For each value i in A we place a vertex a_i, and for each value j in B we place a vertex b_j. Two hash functions h_a and h_b are used to compute the hash values of all f_i. Then, for each f_i, we place an edge between the two vertices corresponding to its hash value. For example, $h_a(f_2) = 3$ and $h_b(f_2) = 1$, so place an edge connecting a_3 and b_1. For a f_i and its corresponding *index*, Othello hashing requires that two connected elements $A[h_a(f_i)] \oplus B[h_b(f_i)] = index$, where $\oplus$ is a bitwise exclusive OR (XOR). For f_2 in this example, $a_3 \oplus b_1 = 11_2 = 3$. Note that after placing the edges of all f_i, the bipartite graph formed by all vertexes must be acyclic. If a cycle is found, the construction must find another pair of hash functions to reconstruct the bipartite graph. It has been proved that the expected time cost of constructing a bipartite graph with n f_i is O(n) [21].

Query in the Othello data plane. The Othello query structure contains two arrays A and B, as shown in Fig. 4. To query the value of f_1, we need to compute h_a and h_b, which are mapped to a_1 of A and b_0 of B. We then compute the bitwise XOR of two vertexes and get the *index* is 00_2. For each packet, only two memory accesses are required, that is, reading a value from arrays A and B respectively, which can achieve fast queries.

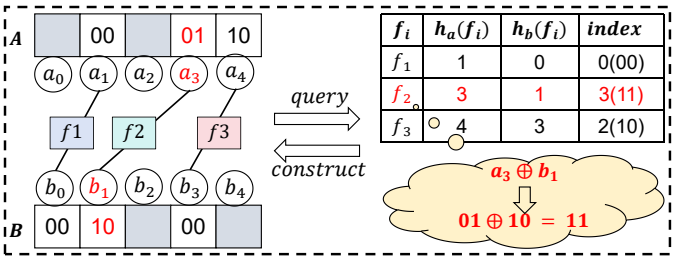


Fig. 4: The construction and query of Othello structure. h_a and h_b are the hash functions corresponding to the arrays A and B respectively.

III. Details of Maat

In this section, we first outline the design rationale of Maat (§III-A). Then the overall framework and workflow of Maat are described in detail (§III-B & §III-C). Finally, the mechanism for Maat to fully maintain PCC is introduced (§III-D). The main symbols used in Maat are shown in Table I.

TABLE I: Symbols frequently used in Maat.

Symbol	Description
Δ	Flow size threshold used to identify load imbalance among servers
S_i	The i^{th} server (DIP)
$T[S_i]$	The total flow size of the i^{th} server
$B[S_i]$	The backup server of the i^{th} server
$F[S_i]$	Boolean value used to identify whether the i^{th} server is acting as a backup server

A. Rationale of Maat

Maat is designed with two dimensions in mind: 1) Unlike existing L4 LBs that only use hash functions to select servers, Maat selects the less loaded server from two choices: one selected by hash and one selected randomly, which improves fairness. 2) Maat divides the incoming traffic into two categories, one is sent to hash-selected servers, and the other is sent to randomly selected servers. This approach helps resolve potential PCC violations with negligible memory overhead. By utilizing these two dimensions, Maat successfully meets both L4 LB requirements—maintaining PCC and improving fairness—without requiring modifications to the existing network infrastructure.

B. Frameworks of Maat

Fig. 5 shows the framework and workflow of Maat. In the data plane, we propose a scheduling method called the power of one random choice for enhancing fairness. Meanwhile, we also integrate Othello hashing and counting Bloom filters into Maat to mitigate potential PCC violations. In the control plane, our main concern is to update the data plane in real-time in response to changes in the DIP pool.

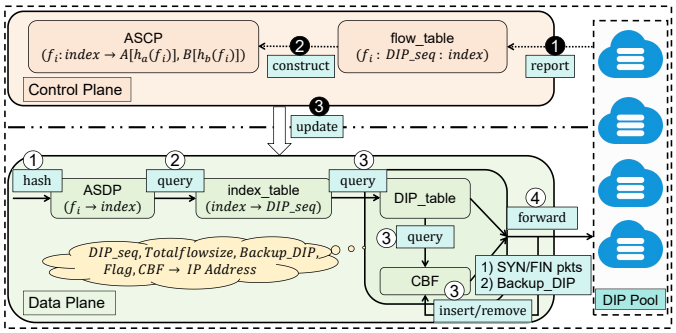


Fig. 5: Framework of Maat.

Maat is mainly composed of six major components:

(1) ASDP (Array Sets on Data Plane) is implemented by Othello hashing structure [21], which consists of two arrays, A and B. Its main function is to query the corresponding *index* for each flow. For example, with f_3 in Fig. 4, we first use h_a and h_b to calculate the hash values of f_3, then use XOR to calculate $A[h_a(f_3)]$ (a_4=10_2) and $B[h_b(f_3)]$ (b_3=00_2). Finally, we can get that the *index* corresponding to f_3 is 2.

(2) index_table establishes a mapping relationship between *index* and *DIP_seq*. This is a many-to-one mapping [9], where multiple different *index* values may map to the same *DIP_seq*. *DIP_seq* has a static mapping relationship with the IP address of the DIP. The *index* is introduced to reduce memory consumption. Because the server's IP address is 32 bits, while the *index* is usually much smaller, such as 8 bits [24]. Additionally, the number of *index* mapped by each *DIP_seq* can also be used to illustrate the weight (load status) of the server.

(3) DIP_table is the core component of Maat and is responsible for forwarding packets to the appropriate server. It describes the status information of all servers (DIPs) running the same service. As shown in Fig. 7, it includes the following five fields: 1) *DIP_seq*: A one-to-one mapping with the DIP's actual IP address. 2) *IP Address*: The actual IP address of the DIP, is used to modify the destination IP address in the packet header. 3) *Totalflowsize (T)*: Maat directly counts the number of packets of all flows processed on each server, rather than distinguishing which flows are large flows and which flows are small flows like the previous solution [33]. This method can more accurately reflect the impact of traffic on the server load (see Fig. 9). 4) *Backup_DIP (B)*: This field is used to record the backup server of S_i, The backup server $B[S_i]$ is randomly selected to ensure load fairness among servers. As shown in Fig. 7(b), the backup server of S_1 is S_2. 5) *Flag (F)*: A boolean field, defaulting to false. When S_1 is selected as the backup for S_2, the *Flag* of S_1 is set to true to prevent other servers (e.g., S_3) from selecting S_1 as their backup server. This field is introduced to achieve better load balancing between servers (see §III-C1 for details).

(4) counting Bloom filter (CBF) is one of the variants of Bloom filter (BF) [41]. BF is a highly space-saving probabilistic data structure that enables efficient insertion and querying. CBF supports deletion operations based on BF, making CBF more suitable for dynamically changing data sets such as network traffic. CBF is primarily used to avoid potential PCC violations.

Time: t0

S_i	$T[S_i]$	$B[S_i]$
S_1	1485	1
S_2	150	null
S_3	1237	1
S_4	1645	1

Time: t1

S_i	$T[S_i]$	$B[S_i]$
S_1	1958	1
S_2	2642	3
S_3	1539	1
S_4	2083	1

Fig. 6: Reason for introducing *Flag* field in DIP_table (**t1** > **t0**, $\Delta = 1000(packets)$).

(5) flow_table maintains the mapping relationship between f_i and the server (DIP) on the control plane, recording the latest flows processed by the servers. It works by using a lightweight application-layer program to track the flow status of each server, which has been deployed to LB [10] in current data centers without changing the network stack. Whenever the flow status on the servers changes (e.g., flow arrival or termination), the servers report to LB, and then LB uses these reports to update flow_table.

(6) ASCP (Array Sets on Control Plane) is also implemented by the Othello hashing structure but serves a different role than ASDP. Maat uses the latest mapping relationship in flow_table to construct ASCP. Whenever the DIP pool updates, ASCP can quickly (microsecond level) update the data plane [24] to ensure that PCC violations not occur in Maat when the DIP pool changes, except in the case of server failure.

C. Workflow of Maat

(1) Data plane: The data plane of Maat is the key to achieving the two requirements (fairness & PCC). For all incoming traffic, the LB selects an appropriate server among those running the corresponding service, rewrites the destination IP field in the packet header, and forwards it to the selected server. Maat's processing of packets is divided into two situations [24]: 1) Stateless packets. The flow to which the packet belongs has not selected any server. For example, for the first packet of each flow, LB needs to select a server for forwarding according to the pre-specified scheduling method; 2) Stateful packets. Unlike stateless packets, the flow to which the packet belongs has been processed. The LB needs to forward the packet to a specific server to ensure that all packets of the same connection are forwarded to the same server (PCC). Next, we describe in detail how the data plane handles these two types of packets.

1) Stateless packets: When Maat receives a stateless packet, such as the first packet (SYN=1) of a certain TCP flow (f_i). We first query the *index* corresponding to the flow in the ASDP (①). Next, we find the *DIP_seq* corresponding to the *index* in the index_table (②), and finally, we find the hash-selected server in DIP_table (assume S_1). We then process the stateless packets based on the power of one random choice (③). We handle the following two situations according to Algorithm 1.

Case 1. $B[S_1]$ is *null*. In this case, S_1 may have a load imbalance with other servers, so we randomly select a server

(assume S_2). If $T[S_1]-T[S_2] \geq \Delta$ and $F[S_2] = false$, where Δ is a predefined threshold indicating whether the flow should be forwarded to the backup server (Lines 1-8). The former condition shows that the load of S_1 is much greater than the load of S_2, there is a load imbalance between S_1 and S_2, and the latter indicates S_2 can serve as the backup server of S_1. Thus, we set $F[S_2]$ is *true* and $B[S_1]$ is S_2, insert the flow into CBF (③) to prevent PCC violation (see §III-D for details), and finally forward the flow to S_2 (④).

If either of the above two conditions is not met, i.e., $T[S_1]-T[S_2] \leq \Delta$ or $F[S_2] = true$, it indicates that S_1 and S_2 may load balance or the load of S_1 is smaller than S_2, or S_2 is already a backup server for other servers (Lines 9-11). In these cases, we should forward the packet to S_1 (④) to achieve better fairness. We introduce the *Flag* field in DIP_table to avoid scenarios like Fig. 6, where at **t0**, the load of S_2 is significantly lower than S_1, S_3 and S_4. Without the *Flag* field, S_2 could be selected as a backup server by the other three servers. S_2 may handle too much load and become unbalanced with other servers at **t1**. This is caused by S_2 being selected as a backup server by multiple servers. In our tests, not introducing *Flag* field lead to a 28.12% decrease in fairness, and as the workload increases, this imbalance becomes more and more serious, especially in the case of long-tail distributions.

Case 2. $B[S_1]$ is not *null*. There may be a load imbalance between S_1 and $B[S_1]$. We compare the loads of S_1 and $B[S_1]$. If $T[S_1]$ is much larger than $T[B[S_1]]$, and the difference exceeds Δ, which means there is a serious load imbalance between S_1 and $B[S_1]$ (Lines 12-16). Maat insert this flow into CBF (③), forwarding the packet to $B[S_1]$ (④). On the contrary, it means that there is load balancing between S_1 and $B[S_1]$, and the stateless packet is still forwarded to S_1 (Lines 17-19).

2) Stateful packets: When Maat receives a stateful packet (belonging to f_i), our goal is no longer to select the appropriate server based on the scheduling method, but to ensure that the connection where the packet is located not be broken. Similarly, we find the correct *index* and *DIP_seq* in ASDP (①) and index_table (②) respectively. We can find the server selected by the hash function (assume S_1). Next, we use the power of one random choice in the DIP_table (③) to handle the following two situations:

Case 1. $B[S_1]$ is *null* (Lines 20-22). In this case, S_1 has not selected any server as a backup server. We only need to forward the packet to S_1 (④), which ensures that all packets belonging to the same connection are forwarded to the same server, that is, no PCC violation occurs.

Case 2. $B[S_1]$ is not *null* (Lines 23-29). To maintain PCC, we need to confirm that the currently processed flow is forwarded to S_1 or $B[S_1]$. Here, we use CBF (see §III-D for details). If the flow record is found in CBF (③), the flow is forwarded to $B[S_1]$ (④). If there is no related record, the flow is forwarded to S_1 (④).

As shown in Fig. 7(a), the first packet of f_1 is hashed to select S_2. Then, by using the power of one random choice, Maat randomly selects S_1, and finally, forwards the first packet of f_1 to S_2 according to the judgment condition. Other situations can also be processed according to Algorithm 1, and the incoming packets are forwarded to the appropriate server to meet the requirements of L4 LB for fairness and PCC.

Algorithm 1: power of one random choice

Input : A stateless/stateful packet of flow f_i,
Hash selected server: S_1,
Threshold: Δ

Output: The server to handle the flow f_i

```
// A stateless packet of flow f_i
1:  if B[S_1] is null then
2:      Random select server (S_2);
3:      if T[S_1] − T[S_2] ≥ Δ and F[S_2] = false then
4:          B[S_1] = S_2;
5:          F[S_2] = true;
6:          T[S_2]++;
7:          CBF.insert(f_i);
8:          return S_2;
9:      else
10:         T[S_1]++;
11:         return S_1;
12: else
13:     if T[S_1] − T[B[S_1]] ≥ Δ then
14:         T[B[S_1]]++;
15:         CBF.insert(f_i);
16:         return B[S_1];
17:     else
18:         T[S_1]++;
19:         return S_1;
    // A stateful packet of flow f_i
20: if B[S_1] is null then
21:     T[S_1]++;
22:     return S_1;
23: else
24:     if CBF.query(f_i) = true then
25:         T[B[S_1]]++;
26:         return B[S_1];
27:     else
28:         T[S_1]++;
29:         return S_1;
```

As seen from the workflow of the Maat data plane, the power of one random choice combines the advantages of the hash algorithm and the power of two choices cleverly. Initially, hash selection plays a key role in significantly reducing PCC violations. At the same time, through one random choice, the utilization of all server resources is improved, thereby greatly enhancing load fairness among servers. Subsequent experiments demonstrate that this algorithm not only effectively preserves PCC but also enhances fairness.

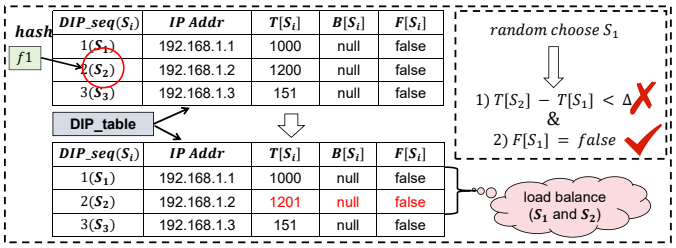
(2) Control plane: The main task of the control plane is to

promptly update the latest mapping relationship between flow and DIP to the data plane when the network changes. This function is similar to the control plane in most previous works [4], [8], [38]. The control plane interacts with the lightweight programs running on each server [10]. Whenever the control plane receives a report from the server, flow_table is updated (❶). Afterward, Maat(❷) constructs the latest ASCP through addition/deletion operations. The time complexity of related operations is $O(1)$ [21]. Whenever the DIP pool changes, the control plane can use ASCP to construct the latest ASDP for the data plane, clears the CBF, index_table and DIP_table is updated to forward traffic to the new DIP pool, where *T*,*B* and *F* of DIP_table are reset (❸).

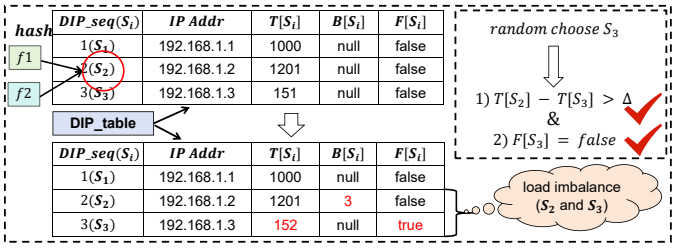
D. Keep PCC

Related work [24] has mentioned that the Othello hashing structure can guarantee PCC regardless of network changes (e.g., DIP pool updates). However, when Maat applies this structure to hash selection in the power of one random choice mechanism, PCC violations may occur. In Fig. 7(a), the hash selection of f_1 is S_2, but the random selection is S_1. It is observed that there is a load balance between S_2 and S_1, therefore, the packet is forwarded to S_2. In Fig. 7(b), the hash selection of f_2 is S_2, and the random selection is S_3. However, there is a load imbalance between S_2 and S_3, so Maat forwards f_2 to S_3 to alleviate the load imbalance. Although the hash selections for f_1 and f_2 are both S_2, the final server selection is different: f_1 is forwarded to S_2, while f_2 is forwarded to S_3. Each flow may have two choices: 1) a hash-selected server; or 2) a randomly selected server. If the selection of each flow cannot be distinguished, a PCC violation is likely to occur.

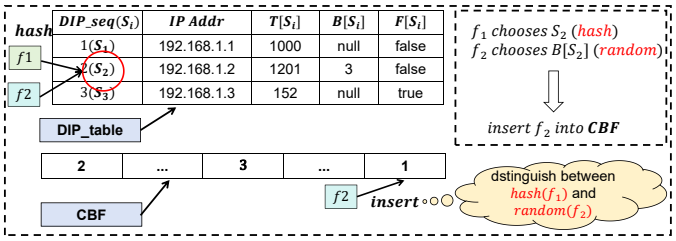
The basic approach involves utilizing an additional table to store the relationship between the connection and the backup server. However, to minimize memory usage, we can elegantly transform this key-value store challenge into a more simplified membership set problem [10]. For this problem, we adopt the counting Bloom filter (CBF) as a solution, CBF is a space-efficient random data structure that can efficiently determine whether an element belongs to a set. Specifically, when a flow is forwarded to the backup server, for SYN packets, we insert this flow into CBF, and for FIN packets, we remove this flow from CBF, and for other packets, we query CBF to confirm whether to forward them to the backup server. In Fig. 7(c), considering f_1 and f_2, f_2 needs to be inserted into CBF. Then, for subsequent packets from f_1 and f_2, a simple query in the CBF can determine whether to forward the packet to S_2 or S_3, thus helping us avoid potential PCC violations. Furthermore, whenever the control plane updates the data plane, the data plane can maintain the latest mapping relationship, so we only need to clear CBF when updating the data plane. With the integration of CBF, Maat effectively addresses potential PCC violations with minimal memory overhead, even under heavy load.



(a) f_1 forward to S_2 (IP Address: 192.168.1.2).



(b) f_2 forward to S_3 (IP Address: 192.168.1.3).



(c) f_2 insert **CBF**.

Fig. 7: An example of keep PCC ($\Delta = 1000(packets)$).

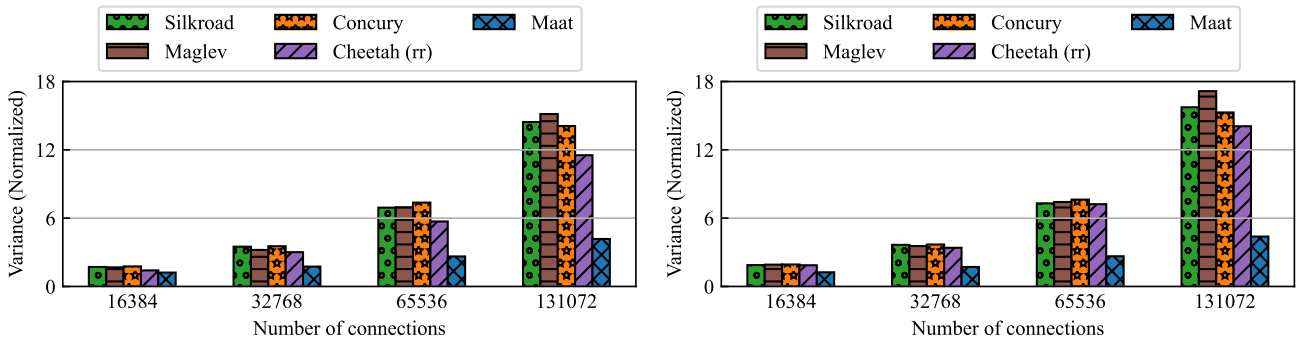
IV. IMPLEMENTATION AND EVALUATION

In this section, we evaluate the performance of Maat based on three types of evaluations: 1) algorithm micro-benchmark (§IV-A); 2) Maat implemented based on DPDK (§IV-B), and 3) Maat running on a Tofino switch (§IV-C). Our experiments aim to answer the following questions.

1) **Can Maat achieve better fairness and guarantee PCC well?** (§IV-A)
2) **How does the performance of traffic processing in Maat?** (§IV-B)
3) **How does Maat perform on a Tofino switch?** (§IV-C)

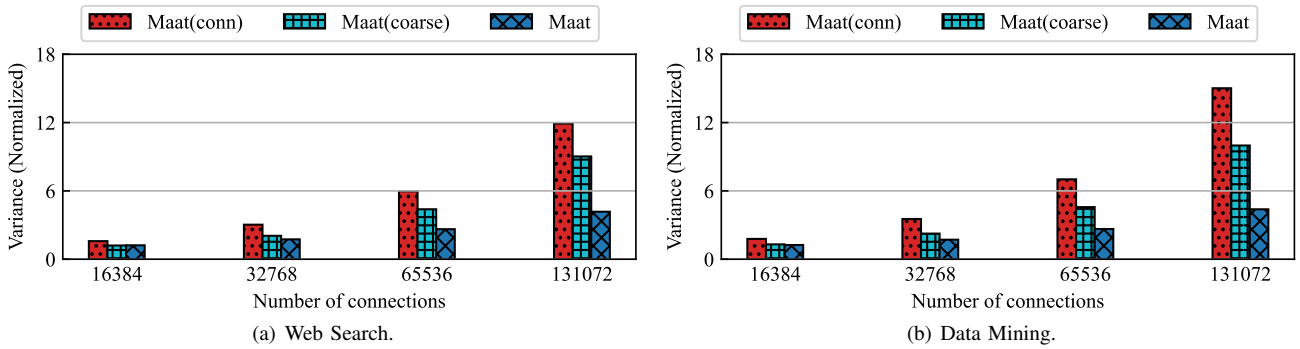
Metric. We evaluate the fairness across servers using both the variance of the server loads and the flow completion time (FCT). FCT refers to the time between the client initiating a connection and receiving the last ACK. Furthermore, we utilize memory overhead and processing throughput to evaluate the performance of Maat. We run at least 30 times for most experiments and take the average.

Workload. For algorithm evaluation (§IV-A), we use two real workloads: 1) Web search workload [35] from a production cluster running web search service; 2) Data mining workload [42] containing many small flows. Both workloads are heavy-tailed: a small percentage of large flows contribute to most of the traffic. For testbed evaluation (§IV-B and §IV-C), We mainly use the following two tools to generate traffic to LB: 1) the Pktgen-DPDK [43]; 2) the WRK [44]. We use a



(a) Web Search. (b) Data Mining.

Fig. 8: Variance among servers' load of various L4 LB schemes for increasing connections.



(a) Web Search. (b) Data Mining.

Fig. 9: Variance among servers' load of different collect traffic status methods for increasing connections.

single server to run up to 32 NGINX web servers (one per hyper-thread), isolated using Linux network namespaces to allow packets to be accepted on the correct CPU core [12]. We generate requests from clients using heavy-tailed distributions.

Parameter selection. We set the parameters of Maat intuitively: 1) We set the threshold Δ to 100K ($1K = 1000$) packets. The reason for setting a larger threshold is that we found that using Maat with a larger Δ can not only use fewer resources to maintain PCC fully but also improve fairness. 2) For the setting of counting Bloom filter size, the main factor we consider is how many resources Maat requires to maintain PCC fully. We find that setting the size of CBF (2 bits per bucket) to 5.6KiB guarantees PCC even under heavy load. When deploying Maat in an actual operating environment, operators should fine-tune the settings of the above parameters according to changes in scenarios and requirements. In all cases, unless otherwise specified, we follow the settings of the above parameters.

A. *Evaluation of algorithm micro-benchmark*

We comprehensively compare Maat with the following schemes: Silkroad [4], Maglev [8], Concury [24], and Cheetah [12], where Cheetah chooses a round-robin mechanism for its scheduling method. These solutions are implemented based on the open-source code provided by [45] and [46]. The server used for the algorithm micro-benchmark experiment has an Intel(R) Xeon(R) Silver 4210R CPU, 2.40GHz, 128GB DDR4 memory, and 13.75MB L3 cache shared by 40 logical cores.

Fairness of different L4 LB schemes. Fig. 8 shows the fairness of different L4 LB schemes through the variance of the server loads. Silkroad [4] mentions that the DIP pool in modern data centers is updated about 10 times per minute, which means that the data plane of LB is updated approximately every 6 seconds. Therefore, in the topology of 1 VIP and 32 DIP, we set the number of flows from 16K to 130K. This is according to the experimental settings of Cheetah [12]. We find that under the same number of connections, Maat improves by 30.62% to 74.42% compared to other hash-based schemes. As the number of connections increases, the fairness issues of hash-based schemes become increasingly obvious, while Maat always maintains good fairness. Maat shows an improvement of 13.4% to 63.26% over Cheetah (rr). The above experimental results show that the fairness of Maat has been significantly improved compared with existing schemes, verifying that the scheduling method of power of one random choice can greatly improve fairness.

Fairness of different statistical methods. Fig. 9 illustrates the fairness of using different methods to count traffic states within the same LB scheme (Maat). This experiment compares three statistical methods: 1) Maat (conn): Counts only the number of connections, ignoring the difference in the impact of flow size on the server load. This can lead to load imbalance. 2) Maat (coarse): Counts traffic in a coarse-grained manner. For example, categorizing any flow over 10KB as a large flow. This method does not differentiate significantly between large flows of vastly different sizes (e.g., 50KB vs. 10MB), potentially causing load imbalance. 3) Maat: this method counts traffic at packet-level granularity, allowing for more evenly distributed incoming traffic. As shown in Fig. 9, Maat achieves

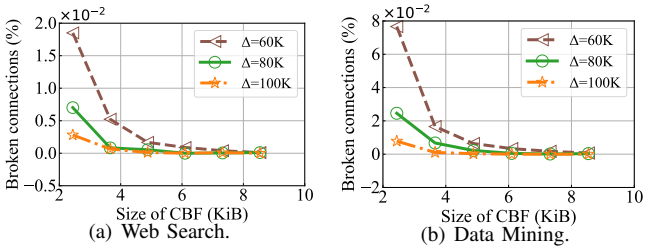


Fig. 10: Broken connections ratio for the increasing size of CBF (2.44 KiB∼8.54 KiB).

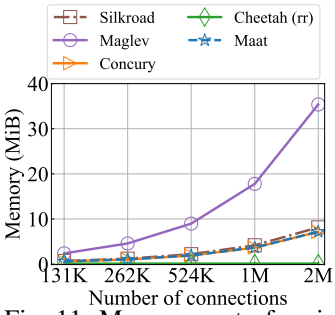


Fig. 11: Memory cost of various schemes.

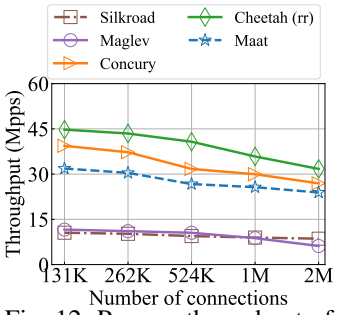


Fig. 12: Process throughput of various schemes.

the best performance, improving fairness by 23.04%∼64.99% compared to Maat (conn) and by 4.5%∼53.78% compared to Maat (coarse). Compared with other schemes that do not require statistical traffic status (e.g., silkroad, maglev, concury, cheetah (rr)), Maat(conn) and Maat(coarse) have little improvement. These results highlight the effectiveness of the packet-level statistical methods employed by Maat.

PCC Violations under the different workload. Avoiding PCC violations is a key feature of L4 LB. Maat does not experience broken connections when the DIP pool updates, but the power of one random choice brings potential broken connections. We solve this problem using CBF. Therefore, we evaluated the memory overhead required by CBF to fully guarantee PCC under heavy load (130K connections) before the next data plane update. The results are shown in Fig. 10. When Δ is 100K packets, CBF only needs 5.6 KiB memory overhead to fully guarantee PCC under both workloads. The main reason is that in Maat, only a small part of traffic (1.59%) requires CBF insertion. For modern servers, a few KiB of memory overhead is negligible. Table II shows the PCC violations of several existing typical LB schemes [12], [24]. Similar to Concury and Cheetah, Maat can guarantee 100% PCC when sufficient memory resources are provided. However, Concury struggles to achieve better fairness, while Cheetah requires modifications to the existing protocol stack.

TABLE II: Broken connections ratio of various schemes.

Schemes	Broken connections
Hash ([3], [47])	11%
Consistent-Hash ([8])	3%
Concury ([24])	0%
Cheetah ([12])	0%
Maat	0%∼0.0766%

B. Evaluation of DPDK implementation

Maat runs on a dual-socket Intel Xeon Silver 4210R CPU @ 2.40GHz, with a total of 40 logical CPU cores (10 cores per socket). We implement Maat using the Data Plane Development Kit (DPDK), a series of libraries for fast user-space packet processing [43]. In this testbed, we connect two commodity servers (*S1* and *S2*) to the Maat, and the configuration information is the same as the server where Maat is located. The Servers are connected via 100G Mellanox Connect-X 5 NICs. Logically, *S1* acts as a client and *S2* emulates 32 backend servers (DIPs). To evaluate the memory overhead and processing throughput of LB, we use Pktgen-DPDK [43] on the *S1* to generate traffic and send them to the *S2* through the LB. To evaluate flow completion time, we generate HTTP requests through WRK on *S1*, and each core of *S2* performs a constant amount of CPU-intensive work to schedule the 8KB file [34].

Memory Usage. Fig. 11 shows the memory cost required to store different connections for Maglev, SilkRoad, Concury, Cheetah, and Maat respectively. Cheetah does not need to maintain the mapping relationship between connections and servers as it stores the DIP in the packet header. Therefore, the memory resources occupied by Cheetah do not increase as the number of connections increases. To maintain PCC, the remaining schemes need to maintain the mapping relationship between connections and DIPs on LBs. As the number of connections increases, we find that the memory cost of Maglev is 79.75% higher than the other three schemes. SilkRoad, Concury, and Maat occupy similar storage spaces. Compared with Concury, Maat introduces the following additional memory overhead: 1) State information about the DIPs needed to improve fairness; 2) CBF. When the number of connections is 1.04 million, the Maat additional overhead only accounts for 0.18% of the total overhead. However, it can improve fairness by about 70.39% compared to Concury.

Process Throughput. We also investigate the processing throughput of different L4 LB schemes. The higher the processing throughput, the fewer LBs we need to deploy. In this experiment, the metric is millions of packets per second (Mpps). In Fig. 12, we find that Cheetah (rr) has a 24.63% improvement over Maat because it does not need to perform operations such as inserting and querying table entries in LB. However, it cannot be ignored that the actual deployment of Cheetah requires modification of the protocol stack. In contrast, Maat is easy to implement without any changes to end-hosts or protocol stack, and can be incrementally deployed in existing networks. Compared with Concury, the throughput of Maat is reduced by 14.01% due to additional compute-intensive operations required to improve the fairness of server loads. However, compared with Maglev and Silkroad, Maat's operations on the data plane are simpler, resulting in improvements of 65.09% and 60.5%, respectively.

Maat improves flow completion time with heavy-tailed workloads. Fig. 13 shows the distribution of flow completion times for 32 servers. *S1* uses WRK to generate heavy-tail

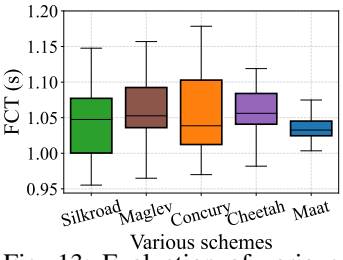


Fig. 13: Evaluation of various L4 LB schemes.

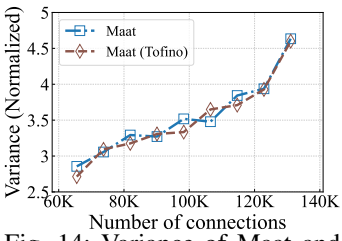


Fig. 14: Variance of Maat and Maat (Tofino).

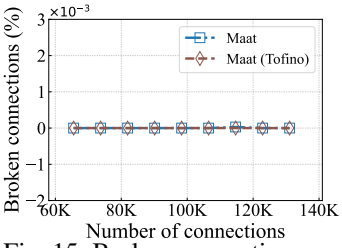


Fig. 15: Broken connections ratio of Maat and Maat (Tofino).

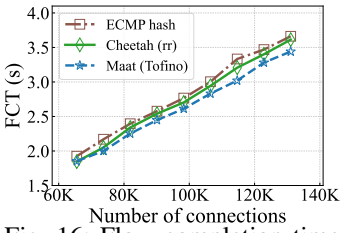


Fig. 16: Flow completion time for Tofino experiment.

TABLE III: Resource consumption of Maat and Cheetah (rr).

Resource	Maat	Cheetah (rr)
Hash Bits	3.8%	2.1%
Gateway	8.3%	4.2%
SRAM	2.2%	1.8%
VLIW Actions	6.0%	2.6%
ALU Instruction	12.5%	4.2%

distributed HTTP requests (130K connections). For Maglev, Silkroad, and Concury, we find that some servers in these three schemes have high FCT, which verifies that hash-based LB has poor fairness performance as mentioned above, while the FCT of Maat is improved by up to 10.58% compared to these solutions. Compared with these hash-based schemes, the improvement of Cheetah (rr) is not significant, because under heavy-tailed distribution, some servers are loaded unpredictably, and Cheetah (rr) struggles to cope with this scenario. This experiment demonstrates the advancement of the scheduling method (power of one random choice) used in Maat, which can help us make full use of the resources of all servers.

C. Evaluation of Tofino implementation

We implement Maat on the Tofino target, where the data plane contains about 700 lines of P4 code. Due to the limitation of hardware resources that are exclusively accessed at all stages of the pipeline, we do not count the first packet of each flow, but this does not affect our measurement of the load on different servers. In Fig. 14 and Fig. 15, we compare the hardware implementation (Maat (Tofino)) and the ideal implementation (Maat), and we can find that our compromised implementation on the Tofino switch does not affect the improvement of fairness and the maintenance of PCC. We use thrift to implement communication between the data plane and the control plane. Once a DIP pool update event occurs, the ASDP needs to be changed, which is done by the control plane through the thrift API. Maat also uses the thrift API to clear the CBF and update the DIP_table to forward new connections to the correct DIP. In this experiment, we use the WRK HTTP request generator on one machine as the client and 32 cores of another machine as the 32 servers, where the two servers are connected through a Tofino switch.

Maat is compared with the following two methods in this experiment: 1) a simple hash-based LB implemented in P4; 2) stateless Cheetah using round-robin (rr), with an increasing number of connections for 1MB files. Table III shows the hardware resource usage of the packet processing pipeline at the programmable switch under Maat and Cheetah (rr). Maat uses more matching action tables and stages in the pipeline to implement comparison and calculation logic. These operations require more stateful ALU and SRAM, resulting in higher consumption of resources such as gateway, SRAM, and ALU instructions. In short, compared to Cheetah (rr), Maat consumes more hardware resources due to fine-grained scheduling of incoming traffic. However, the limited resource consumption is acceptable compared to the benefits of load balancing. In Fig. 16, we observe that FCT of Maat has a 9.67% improvement compared to hash-based LB and still a 6.01% improvement compared to Cheetah with round-robin. Maat achieves better FCT under higher loads because it takes advantage of the opportunities brought by random selection to more fully utilize all available server resources.

V. CONCLUSION

This paper presents Maat, an efficient Layer-4 load balancer for data center networks. By proposing a new scheduling method called the power of one random choice, Maat does not compromise fairness to maintain PCC as hash-based load balancers do. Moreover, Maat does not need to modify the existing protocol stack to introduce more advanced schedule mechanisms. Maat also introduces CBF to distinguish which traffic is forwarded to hash-selected servers and which traffic is forwarded to randomly selected servers, resolving potential PCC violations. We implemented Maat on both a software prototype and a programmable ASIC Tofino switch. The evaluation results show that compared with other existing L4 LBs, Maat can fully maintain PCC at cost of negligible memory overhead and significantly improve fairness.

REFERENCES

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

[2] 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 *Proc. of ACM SIGCOMM*, pp. 51–62, 2009.

[3] 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," pp. 207–218, 2013.

[4] 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. of ACM SIGCOMM*, pp. 15–28, 2017.

[5] C. Zeng, L. Luo, T. Zhang, Z. Wang, L. Li, W. Han, N. Chen, L. Wan, L. Liu, Z. Ding, *et al.*, "Tiara: A scalable and efficient hardware acceleration architecture for stateful layer-4 load balancing," in *Proc. of USENIX NSDI*, pp. 1345–1358, 2022.

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

[7] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, *et al.*, "The quic transport protocol: Design and internet-scale deployment," in *Proc. of ACM SIGCOMM*, pp. 183–196, 2017.

[8] 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 *Proc. of USENIX NSDI*, pp. 523–535, 2016.

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

[10] 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," pp. 27–38, 2014.

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

[12] T. Barbette, E. Wu, D. Kostić, G. Q. Maguire, P. Papadimitratos, and M. Chiesa, "Cheetah: A high-speed programmable load-balancer framework with guaranteed per-connection-consistency," *IEEE/ACM Transactions on Networking*, vol. 30, no. 1, pp. 354–367, 2021.

[13] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, *et al.*, "Conga: Distributed congestion-aware load balancing for datacenters," in *Proc. of ACM SIGCOMM*, pp. 503–514, 2014.

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

[15] Z. Liu, Y. Zhao, Z. Fan, T. Yang, X. Li, R. Zhang, K. Yang, Z. Jiang, Z. Zhong, Y. Huang, *et al.*, "Burstbalancer: Do less, better balance for large-scale data center traffic," *IEEE Transactions on Parallel and Distributed Systems*, 2023.

[16] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal, "Fastpass: A centralized" zero-queue" datacenter network," in *Proc. of ACM SIGCOMM*, pp. 307–318, 2014.

[17] 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. of ACM HotNets*, pp. 155–161, 2016.

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

[19] A. R. Curtis, W. Kim, and P. Yalagandula, "Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection," in *Proc. of IEEE INFOCOM*, pp. 1629–1637, 2011.

[20] B. S. Majewski, N. C. Wormald, G. Havas, and Z. J. Czech, "A family of perfect hashing methods," *The Computer Journal*, vol. 39, no. 6, pp. 547–554, 1996.

[21] Y. Yu, D. Belazzougui, C. Qian, and Q. Zhang, "Memory-efficient and ultra-fast network lookup and forwarding using othello hashing," *IEEE/ACM Transactions on Networking*, vol. 26, no. 3, pp. 1151–1164, 2018.

[22] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, "An improved construction for counting bloom filters," in *Annual European Symposium*, pp. 684–695, Springer, 2006.

[23] Barefoot. https://www.barefootnetworks.com/products/brief-tofino/.

[24] 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 *Proc. of ACM SoCC*, pp. 179–192, 2020.

[25] Y. Yu, X. Li, and C. Qian, "Sdlb: A scalable and dynamic software load balancer for fog and mobile edge computing," in *Proc. of ACM MECOMM*, pp. 55–60, 2017.

[26] J. Zhang, S. Wen, J. Zhang, H. Chai, T. Pan, T. Huang, L. Zhang, Y. Liu, and F. R. Yu, "Fast switch-based load balancer considering application server states," *IEEE/ACM Transactions on Networking*, vol. 28, no. 3, pp. 1391–1404, 2020.

[27] A. M. Abdelmoniem and M. Canini, "Towards mitigating device heterogeneity in federated learning via adaptive model quantization," in *Proc. of ACM EuroMLSys*, pp. 96–103, 2021.

[28] B. Sharma, V. Chudnovsky, J. L. Hellerstein, R. Rifaat, and C. R. Das, "Modeling and synthesizing task placement constraints in google compute clusters," in *Proc. of ACM SoCC*, pp. 1–14, 2011.

[29] S. K. Garg, A. N. Toosi, S. K. Gopalaiyengar, and R. Buyya, "Sla-based virtual machine management for heterogeneous workloads in a cloud datacenter," *Journal of Network and Computer Applications*, vol. 45, pp. 108–120, 2014.

[30] T. Lin, N. Tarafdar, B. Park, P. Chow, and A. Leon-Garcia, "Enabling network function virtualization over heterogeneous resources," in *Proc. of IEEE APNOMS*, pp. 58–63, 2017.

[31] P. Xiao, W. Qu, H. Qi, and Z. Li, "Detecting ddos attacks against data center with correlation analysis," *Computer Communications*, vol. 67, pp. 66–74, 2015.

[32] L. Garber, "Denial-of-service attacks rip the internet," *computer*, vol. 33, no. 04, pp. 12–17, 2000.

[33] X. Guo, L. Zhu, D. Zhang, and C. Wu, "Libra: a stateful layer-4 load balancer with fair load distribution," in *Proc. of IEEE IPCCC*, pp. 246–253, 2022.

[34] 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 *Proc. of USENIX NSDI*, pp. 667–683, 2020.

[35] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center tcp (dctcp)," in *Proc. of ACM SIGCOMM*, pp. 63–74, 2010.

[36] M. Mitzenmacher, "The power of two choices in randomized load balancing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 10, pp. 1094–1104, 2001.

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

[38] J. Zhang, Y. Gao, S. Wen, T. Pan, and T. Huang, "Loom: Switch-based cloud load balancer with compressed states," in *Proc. of IEEE ICNP*, pp. 1–11, 2021.

[39] Y. Yu, D. Belazzougui, C. Qian, and Q. Zhang, "Othello hashing for scalable and fast name switching," in *Proc. of IEEE ICNP*, 2017.

[40] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal, "The bloomier filter: an efficient data structure for static support lookup tables," in *Proc. of ACM/SIAM SODA*, pp. 30–39, 2004.

[41] S. Geravand and M. Ahmadi, "Bloom filter applications in network security: A state-of-the-art survey," *Computer Networks*, vol. 57, no. 18, pp. 4047–4064, 2013.

[42] B. Montazeri, Y. Li, M. Alizadeh, and J. Ousterhout, "Homa: A receiver-driven low-latency transport protocol using network priorities," in *Proc. of ACM SIGCOMM*, pp. 221–235, 2018.

[43] Pktgen-DPDK. https://github.com/pktgen/Pktgen-DPDK.

[44] WRK. https://github.com/wg/wrk.

[45] Cheetah. https://github.com/cheetahlb/.

[46] Concury. https://www.dropbox.com/s/ruou2l340uu1f4u/concury%20code.zip.

[47] 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 *Proc. of USENIX ATC*, pp. 473–485, 2015.