HEELS: A Host-Enabled eBPF-Based Load Balancing Scheme

Rui Yang
EPFL, Switzerland
rui.yang@epfl.ch

Marios Kogias
Imperial College London & Azure Research
m.kogias@imperial.ac.uk

ABSTRACT

Layer 4 (L4) load balancing is crucial in cloud computing and elastic microservices. Existing L4 load balancer designs can be split into two main categories: centralized designs using a hardware or software middlebox, and decentralized designs in which every node can play the role of the load balancer. Centralized designs offer better scheduling policies and easier worker node management, but suffer from I/O and CPU limitations. Decentralized designs scale better, but are harder to manage. We introduce HEELS, a novel load balancing scheme designed for internal cloud workloads and microservices, achieving the best of both worlds. HEELS uses the load balancer only during the connection establishment and allows clients and servers to communicate directly after that. Supporting general L4 load balancers and requiring no kernel changes, HEELS is readily deployable on the public cloud. We implement HEELS as a set of eBPF programs split across the client and server. Our evaluation shows that HEELS introduces minimal overheads, works with off-the-shelf load balancers (*e.g.*, Katran by Meta), and significantly reduces the costs of cloud load balancers.

CCS CONCEPTS

• **Networks** → **Cloud computing**; **Data center networks**; Network performance analysis;

KEYWORDS

L4 Load balancing, Data center networking, eBPF

ACM Reference Format:
Rui Yang and Marios Kogias. 2023. HEELS: A Host-Enabled eBPF-Based Load Balancing Scheme. In *1st Workshop on eBPF and Kernel Extensions (eBPF '23), September 10, 2023, New York, NY, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3609021.3609307

1 INTRODUCTION

Layer 4 (L4) load balancing distributes the network traffic at the granularity of TCP connections. It is the cornerstone of various networked infrastructures ranging from edge systems [1, 4], to datacenters [3, 6, 30] and cloud computing [10, 14]. There exist two primary designs for L4 load balancers: centralized and decentralized. Centralized L4 load balancers are widely deployed and depend on a single network middlebox that can be implemented either in software [6, 30, 31] or hardware [3, 27]. In contrast, decentralized designs are more prevalent in the cloud [5, 17] or at the edge [4, 24], where every node plays both the role of the load balancer and the server, eliminating the need for a dedicated middlebox.

Both designs share the same goal of balancing CPU and I/O load across multiple worker nodes. Ideally, a good load balancing scheme should have the following properties: *(i)* efficiency – balancing the traffic fast and evenly; *(ii)* scalability – not suffering from I/O bottlenecks easily; *(iii)* per-connection-consistency (PCC) – allowing easy updates to the worker pool (*e.g.*, server leave) without breaking existing connections, and *(iv)* deployability – being compatible with various infrastructures without any modifications.

Unfortunately, there is no perfect load balancing solution that meets all these requirements, and picking the right one is a trade-off based on the use case. Decentralized approaches, such as Cloudflare's Unimog [4] or kube-proxy [16] in Kubernetes, are scalable, but can lead to load imbalance due to the absence of a global view. Centralized approaches can make superior load balancing decisions, but can easily become an I/O bottleneck. Although they could use programmable hardware to scale better [27], such hardware is not widely accessible. Similar trade-offs also apply to PCC: stateful designs guarantee PCC but are hard to scale, while stateless designs can break existing connections when updating the worker pool.

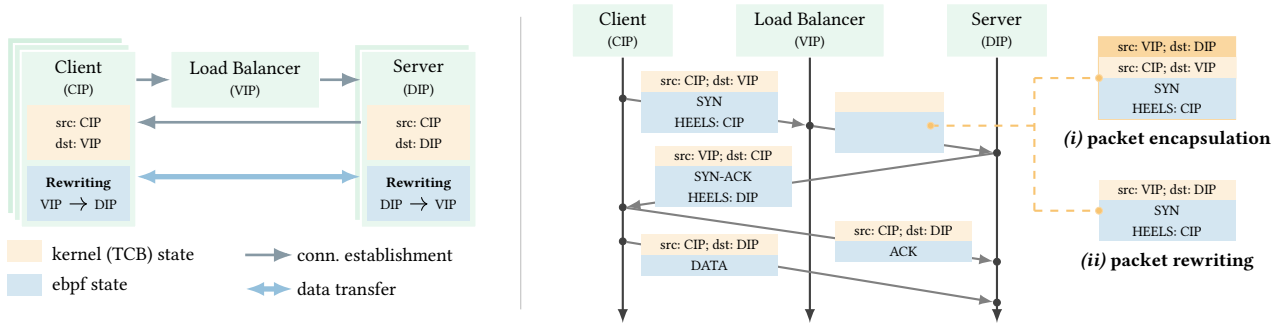
Recent work has made efforts to break these trade-offs. For instance, CRAB [14] targets internal cloud workloads and introduces a new communication pattern, which uses load balancers only for connection establishment. After that, CRAB allows clients and servers to communicate directly, avoiding potential I/O bottlenecks at the load balancer. While this design achieves scalability and PCC, CRAB suffers from several shortcomings, resulting in poor deployability. First, CRAB depends on a customized load balancer to support its communication pattern, making it incompatible with real-world load balancers. Second, CRAB requires kernel changes at both client and server through direct kernel modifications or kernel module loading, which is unrealistic in production.

We present **H**ost-**E**nabled **e**BPF-Based **L**oad Balancing **S**cheme (HEELS), a new L4 load balancing scheme which fulfils all the above properties without the limitations of CRAB. At a high level, HEELS also focuses on internal cloud workloads and follows the same communication pattern as CRAB. This allows HEELS to participate only during connection establishment and be able to implement any load balancing policy. In contrast to CRAB, HEELS instantiates this design with a generalized mechanism, making it compatible with a wide range of existing open-source and proprietary load balancers. Additionally, HEELS requires no kernel modifications to any end-host and allows safe and fast deployment in the real world, addressing the limitations of CRAB.

We implement HEELS as a set of communicating eBPF programs split across the client and the server nodes. We evaluate HEELS in a local testbed and on the public cloud, and show that: *(i)* HEELS

eBPF '23, September 10, 2023, New York, NY, USA

ACM ISBN 979-8-4007-0293-8/23/09…$15.00
https://doi.org/10.1145/3609021.3609307



(a) Per-connection state at end-hosts **(b) TCP sequence diagram of HEELS**

Figure 1: The communication pattern of HEELS, which supports two common L4 load balancers.

adds no latency to data transfer and minimal throughput overhead (~ 3%); *(ii)* HEELS is compatible with off-the-shelf load balancers (*e.g.,* Meta's Katran [24] and AWS' NLB [2]) and requires no kernel modification; and *(iii)* HEELS is readily deployable on the public cloud and can significantly reduce the costs of cloud load balancers.

2 BACKGROUND

In this section, we provide the necessary background on L4 load balancing. In modern data centers, a cluster of backend servers often appears to the outside as a single virtual unit, represented by an L4 load balancer. This load balancer listens to a virtual IP (VIP) that clients use to communicate with the service. When the client initiates a new TCP connection and sends a `SYN` packet to the load balancer (VIP), the load balancer assigns this connection to a particular server that listens to a designated IP (DIP). To forward the traffic to this server, the load balancer needs to modify the packets using either packet rewriting or encapsulation. Packet rewriting directly modifies the destination IP of the packet to DIP while packet encapsulation adds an outer IP header destined to the DIP. Depending on the load balancer's configuration, the server replies through the load balancer or directly to the client, with the former incurring more overhead.

CRAB: CRAB [14] is an L4 load balancing scheme which largely eliminates the overhead of the load balancer. The key idea of CRAB is to involve the load balancer only during the TCP handshake to make load balancing decisions, and exclude it after that. Since in this case, the load balancer does not handle any TCP data traffic, it does not need to maintain any per-connection state and avoids being the I/O bottleneck. Meanwhile, it can implement any centralized load balancing policy since it still participates in the handshake.

To achieve this, CRAB uses a customized TCP option to signal the IPs of servers to clients, facilitating direct communication between them. When the load balancer forwards the `SYN` to a picked server, it adds this TCP option to it, with its own IP address (VIP) appended. Upon sending the `SYN-ACK` packet to the client, the server echoes back this TCP option. Once the client receives the `SYN-ACK` containing VIP, it looks for the previously connection opened towards VIP, and updates the destination IP in its Transport Control Block (TCB) to DIP. This enables a CRAB client to redirect a connection, initially opened with the load balancer, to the server.

Despite its benefits, it is difficult to directly deploy CRAB as it requires modifications at both the load balancer and the end-hosts (*i.e.,* clients and servers). In particular, CRAB only supports a specific type of packet-rewriting load balancer, which needs to be customized to insert the TCP option. It also requires changes to the connection TCB at clients and servers, which can only be achieved through kernel modifications or kernel module loading.

3 DESIGN

We introduce HEELS, a novel connection load balancing scheme that targets internal cloud workloads and cloud-native systems. At a high level, HEELS follows the same communication pattern as CRAB, where the load balancer only participates in connection establishment. However, HEELS generalizes this pattern for a wide range of load balancers and does not need any kernel modifications, bypassing all CRAB's limitations. Hence HEELS exhibits all properties required for an effective load balancing scheme: efficiency, scalability, PCC, and especially, deployability.

Fig. 1a illustrates the overall setup of HEELS. At a high level, HEELS follows a centralized design where a dedicated load balancer serves multiple clients and servers. Fig. 1b describes the connection establishment and data transfer process with HEELS, for both packet-encapsulation and packet-rewriting load balancer designs. As shown, when a client opens a connection to the load balancer, the endpoints of this connection are the client IP (CIP) and the VIP. The client also includes the CIP as a TCP option in the `SYN` packet. The load balancer then forwards the `SYN` packet to a chosen server, using either packet encapsulation or rewriting. Upon receiving this `SYN` packet, a HEELS server always rewrites the `SYN`'s source IP to CIP (extracted from the HEELS TCP option) and destination IP to DIP (in the case of packet encapsulation). This modification occurs before the packet reaches the server's protocol stack, ensuring that the server's connection TCB recognizes CIP and DIP as the two endpoints. Prior to sending out the `SYN-ACK`, the server rewrites the source IP of the `SYN-ACK` to VIP and embeds its own IP (DIP) in another TCP option. This allows the client to *(i)* associate the `SYN-ACK` with the previous connection it opened towards the load balancer without any TCB modification, and *(ii)* acquire the address of the server (DIP). From that point on, whenever the client sends packets for this connection, it rewrites the outgoing packets' destination IP from VIP to DIP, to communicate directly with the

Table 1: The implementation description for all eBPF programs in HEELS

Location	Program Name	eBPF Hook	Description
client	client_sock	SOCK_OPS	• Add CIP in the HEELS-specific TCP option of SYN • Create sk_storage entry, with DIP extracted from SYN-ACK
	client_tc_egress	TC	• Rewrite the destination IP of outgoing data packets to DIP
server	server_sock	SOCK_OPS	• Add DIP in the HEELS-specific TCP option of SYN-ACK • Create sk_storage entry, with VIP extracted from SYN
	server_tc_ingress	TC	• Rewrite the source and destination IPs of incoming SYN • Replace CIP in the HEELS-specific TCP option of SYN with VIP
	server_tc_egress	TC	• Rewrite the source IP of all outgoing packets to VIP

server. Similarly, whenever the server sends packets to the client, it rewrites the outgoing packets' source IP from DIP to VIP, allowing the client to recognize these packets correctly.

The advantage of this design is twofold. First, it supports two common load balancers with a unified underlying mechanism, simplifying the deployment and management. Unlike CRAB, HEELS requires no changes to the load balancers themselves, making it even possible to support proprietary load balancers. Second, the rewriting mechanism at clients and servers eliminates the need for direct modifications of the TCB in the kernel. This frees HEELS from making any kernel changes at end hosts. Instead, HEELS maintains the above rewriting information as the per-connection state. As shown in Fig. 1a, the state is the DIP for the client, and the VIP for the server. Note that the load balancer does not maintain any state, which is another key to HEELS being compatible with a range of load balancers in the real world.

4 IMPLEMENTATION

HEELS is implemented as a series of eBPF programs, strategically placed in the client and server, using different network hooks. Note that HEELS doesn't require eBPF programs at the load balancers, making it highly deployable, and compatible with different load balancing approaches, either based on packet rewriting or packet encapsulation. Our implementation of HEELS consists of $\approx 1.2k$ lines of C code and requires a recent Linux kernel (5.6+) for eBPF support. Table 1 summarizes all eBPF programs used in HEELS, including their location, eBPF hook, and basic functionality. Currently, HEELS supports both Meta's Katran L4 load balancer and AWS Network Load Balancer (NLB) out-of-the-box. We chose these two because they represent packet-encapsulation and packet-rewriting load balancers, respectively. Currently, HEELS supports Katran's IP-IP encapsulation. Note that HEELS can be easily extended to support other common L4 load balancers. Supporting other L4 load balancers requires changes only to the server_tc_ingress program, which needs to process the packets sent by the load balancer. For example, to use HEELS with a new load balancer that implements a different encapsulation protocol, one has to add such decapsulation support in the said eBPF program.

On both the client and server, HEELS maintains per-connection state in a special per-socket eBPF data structure: BPF_MAP_TYPE_SK_STORAGE [19]. On the server side, SK_STORAGE is created to hold the VIP when receiving the SYN packet, while on the client side, it stores the DIP upon receiving the SYN-ACK. On both sides, it is accessed whenever sending outgoing packets that require rewriting. A key advantage of SK_STORAGE is that it has the same lifetime as the attached connection. Hence HEELS eliminates the need for any connection tracking or manual memory management for per-connection state. In the following, we explain in detail the implementation of HEELS on the client and server side.

Client side: On the client side, two different eBPF programs are employed, one hooked at TC [22] and the other hooked at SOCK_OPS [18]. Before sending out the SYN packet, the SOCK_OPS program adds the source IP (CIP) and source port as a TCP option in its TCP header. We implemented the option writing using this hook, instead of TC, to avoid TCP checksum pollution. Upon receiving the SYN-ACK packet from the server, the same SOCK_OPS program parses the TCP options to retrieve the server's IP (DIP). It then adds DIP in SK_STORAGE, which has the same lifetime with the connection and needs no special care from HEELS. From that point forward, the TC program takes charge of rewriting all outgoing packets for this connection. Specifically, the program changes the destination IP of packets from VIP to DIP, *i.e.,* the value retrieved from SK_STORAGE.

Server side: On the server side, the implementation of HEELS involves three eBPF programs hooked at different locations: *(i)* TC ingress, *(ii)* TC egress, and *(iii)* SOCK_OPS. The TC ingress program is in charge of handling and rewriting incoming SYN packets. Depending on the load balancer technology, this program will optionally perform packet decapsulation and header rewriting. It modifies the SYN packet to ensure its source IP is CIP and destination IP is DIP. CIP is retrieved from the TCP option in the packet, which was added by the client. Later when the network stack processes the SYN, it will open a connection between CIP and DIP. Additionally, this program identifies and communicates the load balancer IP (VIP) to the two programs at the server. This is done by retrieving and adding VIP to the TCP option that previously held CIP. Depending on the load balancer, VIP is either the destination IP of the encapsulation header (in Katran) or the source IP of the SYN (in AWS NLB). The SOCK_OPS program on the server has two roles. First, it retrieves VIP from the SYN packet and saves it in the SK_STORAGE. Second, it modifies the SYN-ACK packet to include DIP as a TCP option. Finally, the TC egress program has similar functionality to

the client – rewriting all outgoing packets' source IP to VIP, which is retrieved from the connection's SK_STORAGE.

5 EVALUATION

In this section, we evaluate HEELS's performance and the ease of its deployment in the real world. First, we benchmark the throughput and latency overhead of HEELS (§5.1). Then, we demonstrate the usefulness of HEELS by deploying it with AWS' Network Load balancer and Katran on the public cloud, highlighting its immediate cost-saving benefits (§5.2).

We run the benchmark experiments on three Intel Xeon E5-2637 @3.50GHz with 8 cores (16 hyper threads) and a 10G NIC. The three machines are connected by a Quanta/Cumulus 48x10GbE switch. For the cloud experiments, we employ five virtual machines with instance type `t3.large` from AWS. All the above servers and virtual machines are running Ubuntu 22.04 (kernel 5.15.0).

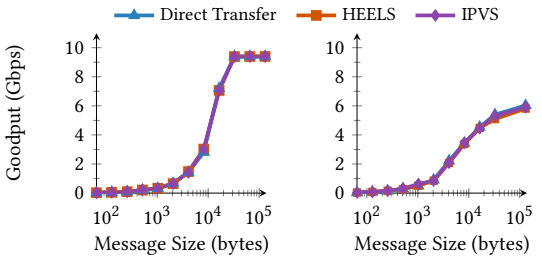
5.1 Performance Analysis

We begin our evaluation by studying the throughput and latency overhead brought by HEELS to both TCP connection establishment and data transmission. We perform the experiments in our local testbed. The testbed setup includes one client machine, one server machine, and one load balancer machine. These three machines are in the same rack, and directly connected with each other via a Top-of-Rack (ToR) switch.

To properly benchmark the performance, we use a customized implementation of Netperf's CRR (Connect-Request-Response) and RR (Request-Response) [12]. The original CRR benchmark measures the latency of establishing a connection, exchanging a single request/response with an 8-byte payload, and closing this connection. In the RR benchmark, clients establish a connection once and then use this persistent connection to exchange all requests/responses with the server. In our evaluation, we perform both CRR and RR benchmarking to measure latency and throughput. However, we customize the message sizes and the number of connections to observe the performance of HEELS under various traffic loads.

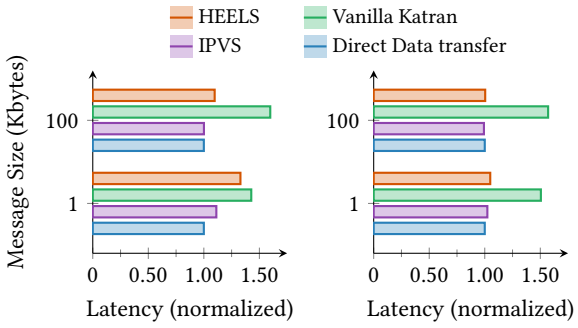
5.1.1 Throughput Overhead. To measure the throughput overhead HEELS introduces, we run the RR benchmark with increasing message sizes and measure the goodput (bytes of application data per unit time). The load balancer machine runs Katran, which is an eBPF-based L4 load balancer. To ensure maximal performance of Katran, we load its XDP programs in native mode. Note that since HEELS only deploys eBPF programs on the client and server, there exists no interference between HEELS and Katran. We compare HEELS with two baselines: *(i)* direct data transmission where the client and the server communicate directly with each other, and *(ii)* IP Virtual Server (IPVS) [20], a mechanism widely used in the decentralized design where the clients perform load balancing. We compare them with HEELS running with Katran.

Fig. 2 shows the goodput comparison between direct data transfer, IPVS, and HEELS. Throughout the experiments, we fix the number of concurrent TCP connections to 100. Fig. 2a represents the results when all CPU cores are enabled on the client and server machines, while Fig. 2b shows the results with only one CPU core enabled on each machine. Fig. 2a demonstrates that HEELS achieves the same goodput with direct data transfer and IPVS throughout



(a) all cores enabled **(b) single core enabled**

Figure 2: The goodput achieved by direct data transfer and HEELS in a 10Gbps network link.



(a) CRR benchmark **(b) RR benchmark**

Figure 3: Unloaded median latency measured for direct data transfer, vanilla katran, and HEELS.

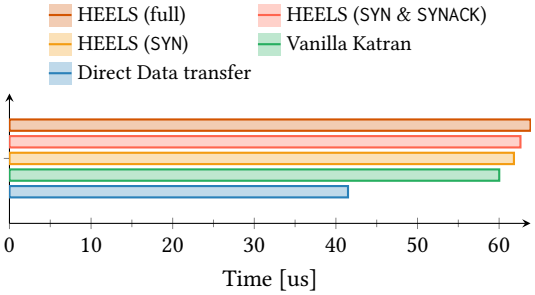


Figure 4: TCP handshake latency breakdown

the experiments. When transmitting 32KB of data, they all achieve a goodput of ~9.39Gbps, starting to saturate the 10G link.

Note that in our testbed, with the default kernel configuration, one single core alone can not saturate the 10G link. Therefore, the goal of the experiment in Fig. 2b is to measure the overhead introduced by HEELS and compare it with IPVS and direct data transfer, while being CPU-bound. As shown, when transmitting 260KB of data, both direct data transfer and HEELS start to reach ~100% CPU usage and their goodput is 6.248 Gbps and 6.043 Gbps, respectively. This indicates a 3.2% overhead brought by HEELS to data transmission. Note that IPVS also achieves similar performance with HEELS, as it brings a 2% ~ 3% overhead to direct data transmission throughout the experiment.

Table 2: Deploying costs for vanilla AWS NLB, HEELS w AWS NLB, and vanilla Katran in the cloud

Message size (Kbytes)	Price per hour ($/hr)		
	Vanilla AWS NLB	HEELS w AWS NLB	Vanilla Katran
8	0.028	0.027	0.092
1024	0.135	0.027	0.092
4096	0.459	0.027	0.092

5.1.2 Latency Overhead. Now we proceed to measure the latency overhead imposed by HEELS. We configure the CRR and RR benchmarks to use a single RX/TX queue and core, a single connection, and allow for a single outstanding request in this connection. We use the following three baselines: *(i)* direct data transmission, *(ii)* IPVS, and *(iii)* vanilla Katran load balancer without HEELS.

Fig. 3 plots the normalized unloaded latency (median) over 400 000 samples using both CRR and RR benchmarks, transmitting 1 KB and 100 KB data. In both benchmarks, vanilla Katran constantly introduces ~50% more latency than direct data transfer. For instance, when transmitting 1KB data, the CRR latency of direct data transfer and vanilla Katran is 105µs and 150µs, respectively. This is due to the fact that all packets sent from the client need to go through the Katran load balancer, resulting in a half RTT latency overhead. Since the SYN packets in HEELS also need to go through the load balancer, this overhead is unavoidable in its handshake phase. Indeed, as shown in the CRR benchmark, HEELS incurs the same overhead to the direct data transfer when sending 1KB data. However, this initial overhead gets amortized for longer connections. For instance, in the CRR benchmark, HEELS only introduces ~9% overhead to direct data transfer for 100KB data transmission. This observation aligns with the RR benchmark shown in Fig. 3b, where HEELS adds almost no extra latency in the data transmission phase, since the TCP handshake is excluded. Note that IPVS achieves similar performance with direct data transfer in both CRR and RR benchmarks, as packets in IPVS are always directly transmitted between the client and the server, without involving the load balancer.

Considering HEELS only exhibits latency overhead during connection establishment, we dive deeper and investigate the latency added by each eBPF program in this phase. Specifically, we incrementally enable and load the relevant eBPF programs, observing the duration of the connection establishment without any data transfer. Initially, we enable the `client_sock` program which adds the TCP option to the SYN packet (HEELS (SYN)). Then, we enable the `server_sock` program which adds the TCP option to SYN-ACK, as well as the `server_tc_ingress` program (HEELS (SYN & SYNACK)). Finally, we enable all the eBPF programs of HEELS. Fig. 4 summarizes the experiment results. We observe that HEELS slightly increases (3-4 µs) the connection establishment duration on top of Katran. This increase is distributed almost evenly among all the eBPF programs involved.

5.2 Deployment in the Cloud

In this section, we demonstrate the usefulness and cost benefits of HEELS by deploying it on the public cloud. We employ four virtual machines (VM) on AWS: one serving as the client and three as backend servers. Each VM uses the ENA driver [21] to allow loading XDP programs. For the purpose of observing unloaded latency, we only use a single client VM. In addition, we use a VM as the Katran load balancer and configure an AWS NLB to distribute traffic to the three backends. At each backend, we use NGINX [29] to serve a static file. We use wrk2 [33] at the client to generate HTTP requests over a persistent connection and measure the latency.

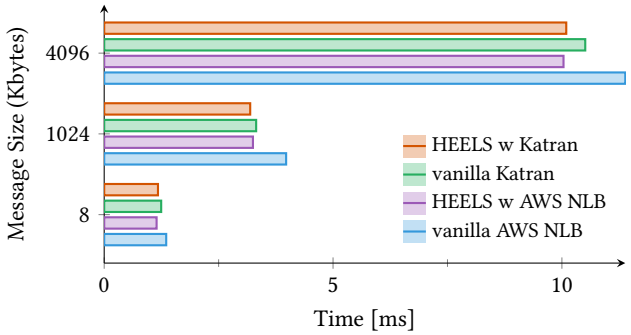


Figure 5: Unloaded median latency measured for AWS Network load balancer and HEELS.

We have the following four different configurations in this experiment: *(i)* vanilla Katran where the traffic is load balanced by the Katran load balancer directly, *(ii)* HEELS with the Katran load balancer, *(iii)* vanilla AWS NLB where the traffic is load balanced by the native load balancer unit, and *(iv)* HEELS with AWS NLB.

Our experiments show that HEELS works seamlessly with both AWS NLB and Katran on the cloud. It is worth noting that the HEELS presents no deployment issues in the public cloud. Fig. 5 plots the median latency measured with increasing file sizes served by NGINX at backends. We observed that vanilla AWS NLB has the highest latency. For instance, its latency is nearly 1.4ms higher than HEELS with AWS NLB, when requesting 4MB of data. There are several factors that may contribute to its higher latency. First, unlike Katran, AWS NLB is not a Direct Server Return (DSR [28]) load balancer, suggesting a higher RTT overhead. Second, since we configured the Katran load balancer with the same instance type as the client and servers, it is likely situated closer to the endpoints compared to AWS NLB. Notably, Fig. 5 also shows that HEELS with Katran and HEELS with AWS NLB achieve similar latency across different file sizes, as their traffic both follows the same data path.

Deploying HEELS on the cloud does not only improve the latency introduced by centralized load balancers, but also offers cost advantages for cloud users. The insight is that cloud providers often charge tenants by the amount of data traversing the load balancer they employ. With HEELS bypassing the load balancer in the data path, the load balancer costs are no longer dependent on the amount of data exchanged between clients and servers. This has the potential of significantly reducing the overall cost for load balancing internal cloud workloads. Indeed, in Table 2 we compute the load balancer costs per hour for the previous latency experiment setup, where the clients continuously request data from NGINX servers. The load balancer costs consist of *(i)* a flat rate of $0.027/hr, and *(ii)* a $0.006/hr rate for every GB processed in this hour. As shown in the table, the AWS NLB costs increase as the message size

grows because of the increasing data traversing the load balancer. In contrast, HEELS with AWS NLB constantly incurs a minimal flat rate since the SYN packets are so small that the AWS pricing model does not even take it into account. Note that the vanilla Katran VM also only incurs a constant flat rate as it resides in the same availability zone with clients and servers, where all traffic is free. Consequently, HEELS with Katran incurs the same cost as vanilla Katran, so we omit it from the table to avoid duplication.

6 DISCUSSION

Load Balancing Policies: Evaluating different load balancing policies is beyond the scope of this paper, which only focuses on the HEELS communication pattern. Previous works [3, 14] analyze the pros (*e.g.,* efficient load balancing and PCC guarantee) and cons (*e.g.,* poor scalability) of stateful services. HEELS can support any stateless (*e.g.,* hash-based) or state-full (*e.g.,* cache-based) load balancing policies. Note that HEELS can simplify the deployment of stateful services and eliminate their scalability problems since HEELS distributes the per-connection state at the end-hosts.

Packet Encapsulation for HEELS: Our initial HEELS' implementation depended on packet encapsulation instead of packet rewriting, where the client adds an encapsulation header to the egress packets instead of rewriting them. This approach turned out to be inapplicable as it led up to 50% overhead in some benchmarks. The reason is that we could only implement egress packet encapsulation in TC, as XDP only supports ingress. Encapsulating packets in the TC layer proves to be costly as it required an additional copy and allocation of sk_buff. In contrast, XDP is capable of performing encapsulation without copying, due to its preallocated headroom in the packet data structure.

All-client-side HEELS: An alternative design to HEELS is to implement all the packet rewriting logic on the client side, eliminating the need for any eBPF program on the server side. However, this approach requires the client side to implement a connection tracking mechanism and use an eBPF map to store the per-connection state. The lifetime of each entry in this map should match the lifetime of the connection TCB in the kernel. In addition, for every incoming packet, at least two hash lookups are needed – one by an eBPF map for packet rewriting and another by the kernel to find the associated TCB. We wanted to avoid the complexity and performance issues of this approach. Therefore, we chose to use eBPF per-connection storage (SK_STORAGE) and split this state at the server and client.

HEELS for other protocols: Another interesting question is if HEELS could apply to other transport protocols. While using HEELS for UDP is challenging due to the lack of connection establishment in UDP, we could potentially adapt HEELS for QUIC [11], a widely deployed transport protocol over UDP. QUIC has its own connection migration, allowing a connection to survive client migration (IP/port changes). However, this mechanism does not widely support server migration and suffers from certain overhead (*e.g.,* path validation) due to QUIC's security concerns. Given that UDP does not have the options necessary for HEELS, we could instead leverage IP options to convey the required information. We leave the exploration of using IP options in HEELS for future work.

7 RELATED WORK

L4 Load Balancers: L4 load balancers can be categorized as centralized and decentralized. Among centralized designs, software solutions [6, 9, 31] are the most popular but often introduce performance overheads, while hardware solutions [7, 27] achieve line-rate processing but require programmable hardware which is not widely accessible. Offloading the load balancing logic to individual clients, decentralized designs [4, 15] provide better scalability and deployability, but suffer from suboptimal load balancing decisions due to the lack of a global view. HEELS achieves the best of both worlds by using the load balancer only during connection establishment phase and allowing direct communication.

Another load balancer split is between stateful and stateless designs. Beamer [30] is a stateless load balancer that guarantees PCC through daisy chaining. In contrast, Cheetah [3] uses a cookie sent by the client to implement stateful load balancing policies. HEELS and CRAB [14] can implement stateful load balancing policies without maintaining any state at the load balancer since they both distribute the per-connection state to the endpoints.

eBPF Network Applications: eBPF has been widely used to accelerate different applications beyond load balancing. BMC [8] uses eBPF to implement an in-kernel cache for Memcached acceleration and Polycube [26] is an eBPF-based NFV framework. Spright [32] leverages eBPF for serverless communication. Syrup [13] is a user-defined scheduling framework dependent on eBPF. A follow-up work on NitroSketch [23] implements sketches in eBPF [25].

8 CONCLUSION

In this paper, we present HEELS, a new layer 4 load balancing scheme which offloads the load balancing logic to the endpoints. Inspired by CRAB, HEELS follows its communication pattern where the load balancer only participates in connection establishment. However, HEELS generalizes this pattern for a wide range of load balancers and requires no kernel modifications, bypassing all CRAB's limitations. We demonstrate that HEELS introduces minimal performance overhead, works natively with common L4 load balancers, and can significantly reduce the cost of cloud load balancers.

ACKNOWLEDGEMENTS

We thank our anonymous reviewers for their insightful comments and helpful feedback. We would like to also thank Edouard Bugnion, Andrii Vasylevskyi, Rüdiger Birkner, and Konstantinos Prasopoulos for many helpful discussions during the course of this project. This work does not raise any ethical issues.

REFERENCES

[1] João Taveira Araújo, Lorenzo Saino, Lennert Buytenhek, and Raul Landa. 2018. Balancing on the Edge: Transport Affinity without Network State.. In *Proceedings of the 15th Symposium on Networked Systems Design and Implementation (NSDI)*. 111–124.

[2] AWS. 2023. AWS Elastic Load Balancing. (2023). https://aws.amazon.com/elasticloadbalancing/ [Accessed: (06/2023)].

[3] Tom Barbette, Chen Tang, Haoran Yao, Dejan Kostic, Gerald Q. Maguire Jr., Panagiotis Papadimitratos, and Marco Chiesa. 2020. A High-Speed Load-Balancer Design with Guaranteed Per-Connection-Consistency.. In *Proceedings of the 17th Symposium on Networked Systems Design and Implementation (NSDI)*. 667–683.

[4] Cloudflare. 2020. Unimog - Cloudflare's edge load balancer. (2020). https://blog.cloudflare.com/unimog-cloudflares-edge-load-balancer/ [Accessed: (06/2023)].

[5] Docker. 2023. Docker Swarm. (2023). https://docs.docker.com/engine/swarm/ [Accessed: (06/2023)].

[6] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. 2016. Maglev: A Fast and Reliable Software Network Load Balancer.. In *Proceedings of the 13th Symposium on Networked Systems Design and Implementation (NSDI)*. 523–535.

[7] Rohan Gandhi, Hongqiang Harry Liu, Y. Charlie Hu, Guohan Lu, Jitendra Padhye, Lihua Yuan, and Ming Zhang. 2014. Duet: cloud scale load balancing with hardware and software.. In *Proceedings of the ACM SIGCOMM 2014 Conference*. 27–38.

[8] Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, and Gilles Muller. 2021. BMC: Accelerating Memcached using Safe In-kernel Caching and Pre-stack Processing.. In *Proceedings of the 18th Symposium on Networked Systems Design and Implementation (NSDI)*. 487–501.

[9] Github. 2016. Github Load Balancer. (2016). https://github.blog/2016-09-22-introducing-glb/ [Accessed: (06/2023)].

[10] Yutaro Hayakawa, Lars Eggert, Michio Honda, and Douglas Santry. 2017. Prism: a proxy architecture for datacenter networks.. In *Proceedings of the 2017 ACM Symposium on Cloud Computing (SOCC)*. 181–188.

[11] IETF. 2021. The QUIC Transport Protocol - IETF. (2021). https://datatracker.ietf.org/doc/html/draft-ietf-quic-transport-34 [Accessed: (06/2023)].

[12] Rick Jones. 2005. NetPerf. (2005). https://fossies.org/linux/netperf/doc/netperf.pdf [Accessed: (06/2023)].

[13] Kostis Kaffes, Jack Tigar Humphries, David Mazières, and Christos Kozyrakis. 2021. Syrup: User-Defined Scheduling Across the Stack.. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*. 605–620.

[14] Marios Kogias, Rishabh Iyer, and Edouard Bugnion. 2020. Bypassing the load balancer without regrets.. In *Proceedings of the 2020 ACM Symposium on Cloud Computing (SOCC)*. 193–207.

[15] Kubernetes. 2018. IPVS-based Kubernetes Load Balancing. (2018). https://kubernetes.io/blog/2018/07/09/ipvs-based-in-cluster-load-balancing-deep-dive [Accessed: (06/2023)].

[16] Kubernetes. 2023. Kube Proxy. (2023). https://kubernetes.io/docs/reference/command-line-tools-reference/kube-proxy/ [Accessed: (06/2023)].

[17] Kubernetes. 2023. Kubernetes Container Orchestrator. (2023). https://kubernetes.io/ [Accessed: (06/2023)].

[18] Linux. 2017. BPF_PROG_TYPE_SOCK_OPS. (2017). https://lwn.net/Articles/727189/ [Accessed: (06/2023)].

[19] Linux. 2023. BPF_MAP_TYPE_SK_STORAGE. (2023). https://docs.kernel.org/bpf/map_sk_storage.html [Accessed: (06/2023)].

[20] Linux. 2023. IPVS Virtual Server. (2023). http://www.linuxvirtualserver.org/software/ipvs.html [Accessed: (06/2023)].

[21] Linux. 2023. Linux kernel driver for Elastic Network Adapter (ENA) family. (2023). https://www.kernel.org/doc/html/latest/networking/device_drivers/ethernet/amazon/ena.html [Accessed: (06/2023)].

[22] Linux. 2023. Linux Trafic Control. (2023). https://man7.org/linux/man-pages/man8/tc.8.html [Accessed: (06/2023)].

[23] Zaoxing Liu, Ran Ben-Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. 2019. Nitrosketch: robust and general sketch-based monitoring in software switches.. In *Proceedings of the ACM SIGCOMM 2019 Conference*. 334–350.

[24] Meta. 2023. Katran. (2023). https://github.com/facebookincubator/katran [Accessed: (06/2023)].

[25] Sebastiano Miano, Xiaoqi Chen, Ran Ben Basat, and Gianni Antichi. 2023. Fast In-kernel Traffic Sketching in eBPF. *Comput. Commun. Rev.* 53, 1 (2023), 3–13.

[26] Sebastiano Miano, Fulvio Risso, Mauricio Vásquez Bernal, Matteo Bertrone, and Yunsong Lu. 2021. A Framework for eBPF-Based Network Functions in an Era of Microservices. *IEEE Trans. Netw. Serv. Manag.* 18, 1 (2021), 133–151.

[27] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs.. In *Proceedings of the ACM SIGCOMM 2017 Conference*. 15–28.

[28] NGINX. 2016. NGINX DSR: IP Transparency and Direct Server Return with NGINX and NGINX Plus as Transparent Proxy. (2016). https://www.nginx.com/blog/ip-transparency-direct-server-return-nginx-plus-transparent-proxy/ [Accessed: (06/2023)].

[29] NGINX. 2023. NGINX Reverse Proxy. (2023). https://docs.nginx.com/nginx/admin-guide/web-server/reverse-proxy [Accessed: (06/2023)].

[30] Vladimir Andrei Olteanu, Alexandru Agache, Andrei Voinescu, and Costin Raiciu. 2018. Stateless Datacenter Load-balancing with Beamer.. In *Proceedings of the 15th Symposium on Networked Systems Design and Implementation (NSDI)*. 125–139.

[31] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert G. Greenberg, David A. Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, Changhoon Kim, and Naveen Karri. 2013. Ananta: cloud scale load balancing.. In *Proceedings of the ACM SIGCOMM 2013 Conference*. 207–218.

[32] Shixiong Qi, Leslie Monis, Ziteng Zeng, Ian-Chin Wang, and K. K. Ramakrishnan. 2022. SPRIGHT: extracting the server from serverless computing! high-performance eBPF-based event-driven, shared-memory processing.. In *Proceedings of the ACM SIGCOMM 2022 Conference*. 780–794.

[33] Gil Tene. 2023. wrk2: a HTTP benchmarking tool. (2023). https://github.com/giltene/wrk2/ [Accessed: (06/2023)].