Hermes: Enhancing Layer-7 Cloud Load Balancers with Userspace-Directed I/O Event Notification

Tian Pan[†], Enge Song[†], Yueshang Zuo[†], Shaokai Zhang[†], Yang Song[†], Jiangu Zhao[†], Wengang Hou[†], Jianyuan Lu[†], Xiaoqing Sun[†], Shize Zhang[†], Ye Yang[†], Jiao Zhang[*], Tao Huang[*], Biao Lyu[‡†], Xing Li[‡†], Rong Wen[†], Zhigang Zong[†], Shunmin Zhu[§†]
[†]Alibaba Cloud [*]Purple Mountain Laboratories [‡]Zhejiang University [§]Hangzhou Feitian Cloud
alibaba_cloud_network@alibaba-inc.com

Abstract

Layer-7 load balancers (L7 LBs) improve service performance, availability, and scalability in public clouds. They rely on I/O event notification mechanisms such as *epoll* to dispatch connections from the kernel to userspace workers. However, early epoll versions suffered from the thundering herd problem. Epoll exclusive (available since Linux 4.5) mitigates this but introduces LIFO wakeups, causing connection concentration on a few workers. Reuseport (Linux 3.9) hashes connections across workers but suffers from hash collisions and lacks awareness of worker load. Since each worker serves multi-tenant traffic, inter-worker load balancing is critical to avoid worker overload and preserve tenant performance isolation.

In this work, we present *Hermes*, a userspace-directed I/O event notification framework to enhance L7 LBs. Hermes uses userspace worker status to direct kernel-space connection dispatch. It implements lock-free concurrency management for inter-process worker status updates and retrievals, as well as scheduling decision synchronization from userspace to the kernel. In the kernel, Hermes leverages eBPF to non-intrusively override the reuseport socket selection for custom worker scheduling. Hermes has been deployed on O(100K) CPU cores in Alibaba Cloud, handling O(10M) RPS of traffic. It reduces daily worker hangs by 99.8% and lowers the unit cost of L7 LB infrastructure by 18.9%.

CCS Concepts

• **Networks** → **Cloud computing**; **Middle boxes / network appliances**; **Application layer protocols**.

Keywords

Layer-7 load balancer, multi-tenancy, I/O event notification, Linux epoll, epoll exclusive, reuseport, closed-loop control, eBPF

Tian Pan, Enge Song, and Yueshang Zuo contributed equally to this work. Yang Song and Shunmin Zhu are co-corresponding authors.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGCOMM '25, Coimbra, Portugal
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1524-2/25/09
https://doi.org/10.1145/3718958.3750469

ACM Reference Format:
Tian Pan, Enge Song, Yueshang Zuo, Shaokai Zhang, Yang Song, Jiangu Zhao, Wengang Hou, Jianyuan Lu, Xiaoqing Sun, Shize Zhang, Ye Yang, Jiao Zhang, Tao Huang, Biao Lyu, Xing Li, Rong Wen, Zhigang Zong, Shunmin Zhu. 2025. Hermes: Enhancing Layer-7 Cloud Load Balancers with Userspace-Directed I/O Event Notification. In *ACM SIGCOMM 2025 Conference (SIGCOMM '25), September 8–11, 2025, Coimbra, Portugal.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3718958.3750469

1 Introduction

An L7 LB in public clouds routes traffic based on application-layer attributes (*e.g.*, HTTP headers), enabling fine-grained control over request distribution. It enhances the performance, availability, and scalability of cloud-hosted services [44, 48, 74] and has become one of the core products offered by major cloud service providers [3, 4, 35, 36]. In Alibaba Cloud, the L7 LB clusters have scaled to O(100K) CPU cores, handling O(10M) requests per second (RPS) and serving tenants across 33 global regions. Given the diverse and computation-intensive nature of L7 processing at these LBs (*e.g.*, HTTP-based routing, encryption/decryption, protocol translation, and compression/decompression), the industry typically deploys these devices on multicore servers or VMs [48]. In our L7 LB implementation, to support multi-tenancy, we assign different OS ports to serve different tenants and spawn multiple userspace worker processes to listen for incoming connections on these ports. Since each worker handles traffic from a large number of tenants, preventing worker overload is crucial to preserving inter-tenant performance isolation [54, 57, 73]. Therefore, effective load balancing across userspace workers is essential.

TCP connections are established in the kernel via a three-way handshake, while accepted connections are handled by userspace workers. Therefore, I/O event notification mechanisms [10, 25, 29, 55] are required to dispatch new connections from the kernel to userspace. A good notification mechanism helps achieve balanced workload distribution across workers. For stability and maintainability, we build our L7 LBs on Linux *epoll* [10], a mature and efficient I/O event notification facility widely adopted in modern Internet applications [2, 9, 19, 26, 27, 32]. Early versions of epoll suffered from the thundering herd problem when multiple workers listened on the same port [33]. To address this, Linux 4.5 introduced epoll exclusive, which allows only one worker to be awakened for incoming events on a shared port [11]. However, epoll exclusive places all waiting workers for a port into a queue and tends to wake the most recently enqueued worker, often leading to skewed connection distribution. As an alternative, reuseport (introduced in Linux 3.9) allows multiple sockets to bind to the same port, enabling

connection dispatch across these sockets via hashing [31]. Each worker can then be assigned a dedicated socket, thereby avoiding epoll exclusive's unfair wakeup issue. However, stateless hashing may perform poorly under heavy-hitter traffic with hash collisions. Furthermore, reuseport cannot detect userspace worker failures and may continue to dispatch connections to unavailable workers.

A key limitation of epoll is that the kernel dispatches connections without awareness of userspace worker runtime status. Unlike L4, L7 connections exhibit significant variation in processing load (*e.g.*, encryption, compression, or simple data copying), which the kernel cannot estimate based solely on the number of packets in its queue. To address this, we propose *Hermes*, a userspace-directed I/O event notification framework to enhance our L7 LBs. Hermes treats userspace worker status as a first-class citizen in L7 load balancing decision making, constructs flexible and efficient connection dispatch control, and customizes kernel dispatch behavior non-intrusively with eBPF [8]. Specifically, Hermes selects worker availability, pending event number, and accumulated connection number as reference userspace performance metrics, and adds only a few lines of code to the original epoll event loop for metrics collection and kernel updates. Hermes performs worker-triggered distributed scheduling, where each worker applies a coarse-grained filter at the end of its epoll event loop to identify a subset of available workers, then reports them to the kernel. The kernel then applies a fine-grained filter using eBPF to choose the final worker to schedule. This two-level filtering prevents incoming traffic from being continuously directed to a single worker. Hermes implements efficient lock-free concurrency management of shared memory for worker status updates and retrievals, as well as eBPF maps for synchronizing scheduling decisions between userspace and the kernel. In kernel space, Hermes overrides the default hash-based socket selection of reuseport and harnesses the limited programmability of eBPF to implement custom scheduling using bitwise operations.

Our major contributions are summarized as follows:

- We propose a closed-loop I/O event notification framework that incorporates userspace worker runtime status into in-kernel connection dispatch to improve L7 load balancing. Hermes is well suited for cloud L7 LBs facing diverse and rapidly changing traffic patterns, where no single scheduling policy can optimally handle all tenant workloads. As epoll underpins many applications, Hermes can be readily applied to various epoll-based use cases.
- When adapting the closed-loop control framework of Hermes to cloud-scale L7 LBs, we address several technical issues, including how to select appropriate worker status metrics; how to efficiently update and retrieve these metrics; how to prevent the selected worker from being overloaded by directed traffic; when and how frequently scheduling should be performed; how to efficiently synchronize between userspace and kernel space; how to achieve non-intrusive kernel modifications; and how to harness eBPF's constrained programmability to perform custom scheduling.
- Hermes has been deployed at scale across all 33 regions of Alibaba Cloud for over two years, sustaining O(10M) RPS of production traffic. Compared to epoll exclusive, Hermes reduces the standard deviation of per-worker CPU utilization and connection counts by 90% and 99.4%, respectively, while delivering the best or near-best performance across various workload scenarios (other epoll modes underperform in some scenarios). After deploying Hermes, the average daily worker hangs have decreased by 99.8%, and the unit cost of cloud infra for our L7 LBs has dropped by 18.9%.

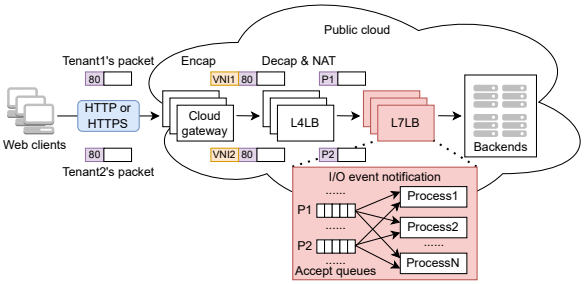


Figure 1: Multi-tenant L7 LBs in Alibaba Cloud.

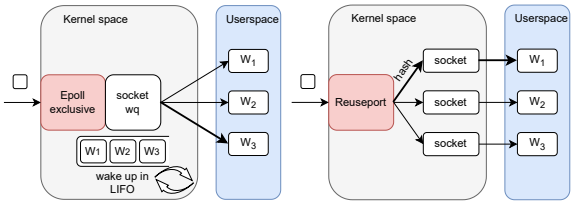
2 Background and Motivation

2.1 L7 LBs in Alibaba Cloud

Multi-tenant implementation of L7 LBs. In Alibaba Cloud, the L7 LB (*a.k.a.*, application LB [3, 4, 35, 36]) distributes HTTP/HTTPS traffic to various backends. Most L7 LB traffic originates from Internet web clients, with a portion from cloud-hosted services. The L7 LB functions as a reverse proxy, handling connection termination and packet payload processing. Common processing tasks include: (1) parsing HTTP packets and routing requests based on user policies; (2) decrypting HTTPS traffic from clients to reduce the SSL processing burden on backends; (3) performing protocol translation for other client protocols (*e.g.*, QUIC) to ensure compatibility with backends; and (4) compressing responses generated by backends before sending them to clients to reduce Internet bandwidth consumption. In our cloud, the L7 LB cluster has scaled to O(100K) CPU cores, handling O(10M) RPS and serving tenants globally.

Fig. 1 shows how our L7 LBs handle multi-tenant traffic. When incoming HTTP/HTTPS traffic (port 80/443) arrives from the Internet, the cloud gateway [64, 66] encapsulates the traffic with a VXLAN header, using the VNI to distinguish different tenants [58]. Before the traffic reaches the L7 LB, the L4 LB decapsulates the VXLAN header and performs NAT, mapping different tenants' traffic originally towards port 80 or 443 to distinct new Dports through header rewriting (*i.e.*, P1, P2, ..., as shown in Fig. 1). At the L7 LB, we bind separate listening sockets to these Dports to handle traffic from different tenants. The kernel assigns each listening socket its own accept queue to hold connections that have completed the TCP handshake but have not yet been accepted by the userspace worker processes. To reduce context switch overhead and ensure predictable performance, userspace workers are pinned one-to-one to CPU cores. As traffic management (*e.g.*, rate limiting) can be enforced at port granularity, this multi-port design enables tenant traffic isolation and fine-grained processing at our L7 LBs.

Epoll-based connection processing. For stability and maintainability, we handle connections at our L7 LBs using Linux epoll [10]. Epoll outperforms select [29] and poll [25], which scale poorly with a large number of file descriptors due to linear scans. Fig. A1 in Appendix shows how epoll handles concurrent connections in a worker process. First, multiple listening sockets are created and bound to different ports (we allocate multiple ports to serve multiple tenants). Second, these listening sockets are added to an epoll instance via epoll_ctl(). Third, the epoll event loop is established



(a) Epoll exclusive **(b) Reuseport**

Figure 2: Two I/O event notification modes with epoll.

and epoll_wait() is called to wait for events (*e.g.*, a new connection request). Fourth, when epoll_wait() detects that any listening socket is ready (*i.e.*, one or more new connections are waiting in the accept queue), it returns and calls accept() to dequeue a connection from the accept queue. Fifth, accept() creates a new conn_fd for the established connection, which is then added to the epoll instance to monitor future events (*e.g.*, data arrival, readiness to write, or disconnection) on that connection (line 19-24 in Fig. A1). At this point, epoll_wait() monitors both the listening sockets and the newly accepted connection. For event processing, our L7 LBs work in a run-to-completion manner over the epoll event loop.

2.2 Uneven Connection Dispatch with Epoll

The thundering herd problem. As shown in Fig. 1, in our design, each worker process listens on multiple ports, and each port is also listened to by multiple worker processes at the same time, leading to the "thundering herd" problem. That is, when multiple workers wait on the same event (*e.g.*, network data arriving on a socket), all of them are woken up simultaneously when the event occurs, even though only one can actually handle it, leading to unnecessary CPU waste. This behavior occurs in early versions of epoll [33]. Nginx uses an accept mutex to allow only one worker to accept new connections at a time [20]. However, since the accept mutex is a locking mechanism, it introduces additional overhead.

Epoll exclusive. Epoll exclusive, introduced in Linux 4.5, effectively mitigates the thundering herd problem [11]. When multiple worker processes listen on the same socket, they are added to its wait queue via epoll_ctl(). In early epoll versions, a socket state change (*e.g.*, data arrival) triggers the socket to wake up all waiting worker processes in the queue. With epoll exclusive, when a worker registers on a socket via epoll_ctl(), it can explicitly declare that it needs to be woken up exclusively via the WQ_FLAG_EXCLUSIVE flag. When an event occurs, if all workers have set the exclusive flag, the wait queue traversal stops immediately after the first idle worker is woken up to handle the event (Fig. A2 in Appendix).

The wait queue is implemented as a list in the kernel and worker processes are added to the head of the list when they call epoll_ctl(). That is, the worker that most recently joins the wait queue will be prioritized to be woken up during each list traversal, unless it is currently busy. This, however, leads to an imbalance in connection dispatch across workers pinned to CPU cores, as most connections concentrate on only a few workers (as shown in Fig. 2a). Such "LIFO" wakeup behavior of epoll exclusive has also been observed by both the Linux community [23, 34] and the industry [37].

Epoll roundrobin (rr). To address the unfair wakeup issue with epoll exclusive, the Linux community proposed epoll rr [23, 34], which moves the recently awakened worker process to the tail of the list. While epoll rr resolves the fairness issue, its cache-cold nature hampers the performance of some cache-sensitive applications built on epoll, and it has not been merged into the kernel.

Table 1: Request size and processing time distributions.

Region	Request size (bytes)			Processing time (ms)		
	P50	*P90*	*P99*	*P50*	*P90*	*P99*
Region1	243	312	2491	2	9	42
Region2	831	3730	10132	10	77	8190
Region3	566	1951	50879	3	278	49005
Region4	721	1140	4638	4	14	239

Reuseport. Reuseport was introduced in Linux 3.9 [31]. Reuseport and epoll offer distinct connection dispatch mechanisms at different stages of the network stack. Reuseport functions at the socket binding stage, deciding which socket handles a connection when the initial SYN packet arrives, whereas epoll operates at the event notification stage, deciding which worker is woken up to accept a connection already queued on a socket after the handshake. They can be combined to help mitigate the thundering herd problem.

With reuseport enabled via the socket option SO_REUSEPORT, multiple sockets can be bound to the same port. Consequently, instead of multiple workers sharing a single listening socket, each worker can listen on a dedicated socket, all bound to the same port. The kernel distributes incoming connections among these sockets via hashing for load balancing. Since each socket is exclusively associated with a single worker, epoll's notification mechanism no longer suffers from wakeup order issues (as shown in Fig. 2b).

However, reuseport's stateless hashing may perform poorly in extreme cases where heavy hitters collide in the hash space. Additionally, if a worker hangs due to a heavy task or crashes, the stateless reuseport may continue to dispatch new connections to the unresponsive worker (epoll exclusive handles this by assigning new connections to the next available worker in the wait queue).

We show an example demonstrating behaviors of epoll exclusive and reuseport in Fig. A3 in Appendix.

Userspace dispatcher. To address uneven connection distribution with epoll, one workaround is to decouple event processing from event fetching by assigning separate workers to process epoll events rather than those that have fetched the events from epoll_wait(). In this approach, a userspace dispatcher collects epoll events and distributes them to backend workers according to fair scheduling policies. This design is common in some database management systems with relatively expensive backend jobs [22]. However, it does not fit our L7 LBs. In database systems, most CPU resources are consumed by backend job processing, so the userspace dispatcher rarely becomes a bottleneck. In contrast, for network applications, a userspace dispatcher on the critical path may become overloaded under high connections-per-second (CPS) traffic. As for our run-to-completion design over epoll, the dispatcher resides within the kernel, reducing the likelihood of it becoming a bottleneck.

2.3 Measuring Load Imbalance in Our Cloud

We measure the load imbalance caused by the default-enabled epoll exclusive in our cloud across various dimensions.

Traffic characteristics across regions. Table 1 shows the request size and processing time across four global regions. Variations in service types and traffic loads lead to significant differences in processing time. Notably, Region3 handles more WebSocket requests

Table 2: CPU utilization imbalance within a device and across devices in a region with 363 L7 LB devices.

	Max CPU util	Min CPU util	Worker-level Avg
Device with Max CPU util diff	74.3%	2.3%	19.6%
Device with Min CPU util diff	14.4%	0.1%	1.18%
Device-level Avg	49.3%	1.88%	8.79%

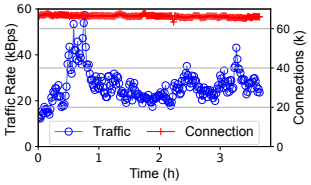


Figure 3: Traffic rate and #connections through a port.

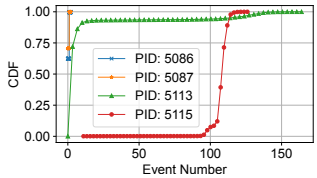
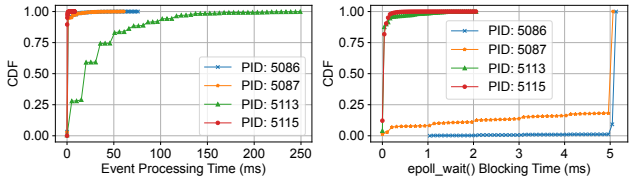


Figure 4: CDF of #events returned from epoll_wait().



(a) Event processing time **(b) epoll_wait() blocking time**

Figure 5: CDF of event processing time and epoll_wait() blocking time in one minute in our cloud.

due to its specific service composition. Since a WebSocket connection is typically counted as a single request, the measured request size and processing time are much larger (although WebSocket requests are large, each connection counts as one request, making their overall share small; hence, the P99 is high while P50 and P90 remain low). Considering our L7 LB needs to handle multi-tenant traffic with diverse characteristics, we aim to adopt a scheduling strategy that can accommodate a broader range of traffic models.

CPU load imbalance in a region (Region2). Even within the same region, variations in service types and traffic loads across tenants, as well as the use of epoll exclusive, cause significant differences in CPU utilization of our L7 LBs (in our implementation, each L7 LB device is deployed as a VM with dedicated CPU cores and hosts multiple L7 LB instances purchased by tenants). Table 2 shows two L7 LB devices with the max/min CPU core utilization difference and their max/min/avg CPU core utilization, as well as the average values of all 363 L7 LBs in the region. CPU load imbalance presents two issues: (1) elevated latency and consequent client packet retransmissions due to single-core overload; (2) increased infra costs from scaling out additional VMs to avoid CPU overload.

Lag effect of connection load imbalance. Fig. 3 shows a scenario in which, after a large number of long-lived connections are established, a sudden traffic surge occurs simultaneously on these connections under specific conditions. This causes the CPU utilization imbalance, resulting from uneven connection distribution among workers under epoll exclusive, to be suddenly and sharply amplified at a later time. When connection traffic surges concurrently, some CPU cores that are already handling a large number of connections become overloaded, while others remain underutilized. The overloaded CPUs continue to process incoming connection requests, resulting in significantly increased processing latency. Our L7 LB has a 200-300μs normal processing latency, but due to

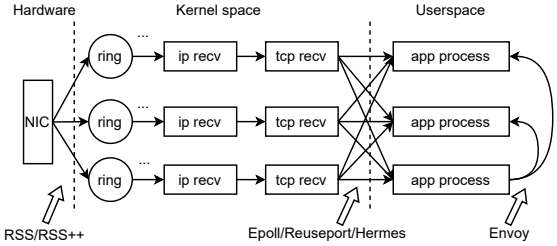


Figure 6: Traffic scheduling at different positions.

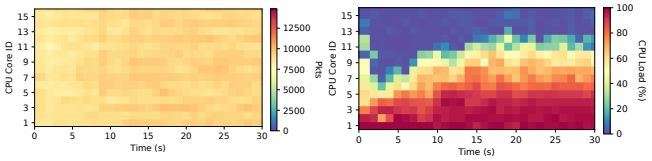
the sudden traffic bursts mentioned above, we observed the P999 latency spiking to 30ms, causing customer complaints. From our experience, many services exhibit such traffic pattern. For example, quantitative trading establishes long-lived connections and may have sudden traffic bursts if certain trading conditions are met.

Load imbalance across workers within an LB. Fig. 4 shows the CDF of the number of events returned from epoll_wait() for four workers on the same L7 LB device over 10s. It shows that PID 5113 and 5115 are busier than the others, as their epoll instances tend to collect more events from kernel space each time. Fig. 5a shows the CDF of the event processing time after these events are returned from epoll_wait(). It indicates that PID 5113 has a significantly longer event processing time, even though it handles fewer events than PID 5115. This suggests that PID 5113 handles more computation-intensive tasks. Fig. 5b shows the CDF of epoll_wait()'s blocking time. If no events arrive within a certain period, epoll_wait() returns after 5ms to execute the event loop, according to our settings. It can be observed that PID 5086 and 5087 are relatively idle, with most of their epoll_wait() calls blocking for the entire 5ms. In contrast, PID 5113 and 5115 are more active.

3 Design Principles

Treat userspace worker status as a first-class citizen in L7 load balancing. In network systems, the traffic scheduler can take effect at different stages of packet processing, as shown in Fig. 6. Broadly, it can be categorized into three levels: NIC, protocol stack, and application. RSS [16] and RSS++ [40] primarily operate at the NIC level. The NIC distributes packets to different CPU cores' ring buffers based on the packet's 5-tuple using a configured hash algorithm, thereby balancing the packet processing load across CPU cores. Epoll and reuseport operate within the kernel protocol stack. Packets trigger the kernel via interrupts, undergo IP and TCP header parsing, are aggregated into connections, and then directed to the appropriate userspace worker for handling based on specific policies. This ensures efficient distribution of new connections among userspace workers. Userspace L7 LBs, such as Envoy [9], can redistribute incoming connections at the application level using locks, albeit with additional overhead (Envoy's exact balance [1]).

For L3/L4 middleboxes, the processing target is packets, and the CPU cycles required for each packet are generally similar (as they involve operations like table lookups). Load can therefore be estimated based on simple metrics, such as the number of packets in the queue, to achieve fair multicore scheduling [40]. However, for L7, the processing target shifts to connection requests, which vary significantly in size and processing complexity, ranging from simple data copying to computationally demanding operations such as encryption and compression. This makes mechanisms like RSS used



(a) L4 LB heatmap (RSS) (b) L7 LB heatmap (epoll)

Figure 7: On an L7 LB, packets are evenly distributed across NIC queues, but CPU core utilization is highly unbalanced.

in L4 LBs ineffective for L7 LBs. The measurement of our L7 LBs indicates that while packets are evenly distributed across NIC queues, CPU core utilization differences remain significant (Fig. 7). This is because different connection requests follow different software execution paths, with processing time varying widely. The load of connection processing is difficult to estimate in kernel space based on the number of packets in the queue. Instead, in userspace, more granular metrics such as the number of events, the types of event handlers, and the sizes of packets associated with those events can be obtained to estimate the workload of each connection, allowing for more effective scheduling. Existing event notification facilities like epoll exclusive are unaware of such userspace information.

Essentially, L7 processing is often computation-intensive, whereas L3/L4 is forwarding-intensive. We frequently see L3/L4 processing with kernel bypass to reduce kernel overhead [41, 50], but for L7, we still rely on the standard Linux protocol stack. Stability is just one reason, but more importantly, the kernel is no longer the bottleneck for L7 workloads; most CPU time is spent on userspace worker tasks. Based on our experience, a 2Gbps traffic load can cause the CPU utilization of a 32-core L7 LB to reach 50%, whereas at the same CPU utilization, an L4 LB can forward 30Gbps of traffic. Therefore, the status of userspace workers should be the primary factor when making connection scheduling decisions for L7 LBs.

Build flexible and efficient closed-loop I/O event notification. In some single-user, well-defined service scenarios, a specific kernel scheduling method works well. However, in public clouds, different regions exhibit diverse traffic and service models. Even in the same region, due to multi-tenancy, there are still various models (with different services favoring different scheduling strategies, *e.g.*, cache-hot or load balancing). Therefore, we cannot rely on a single, fixed kernel scheduling method to solve all cases (*i.e.*, there is no silver bullet). Instead, a flexible scheduling framework is preferable.

Even with userspace information, connection dispatch still occurs in the kernel, so it is necessary to build a feedback control loop to notify the kernel of the userspace information. There are many design choices to consider, such as how userspace and the kernel should interact. Should the kernel ask userspace for input each time it schedules a connection, or should userspace update the kernel periodically? Additionally, what data format should be used for communication between userspace and the kernel? Should userspace send the information directly to the kernel for parsing, or should userspace perform some preprocessing? Regardless of the approach, efficiency is a key factor we need to consider.

Customize kernel functionality in a non-intrusive way. Connection scheduling based on userspace input requires kernel modifications. However, these modifications must accommodate a wide range of use cases, and specific needs are difficult to merge into

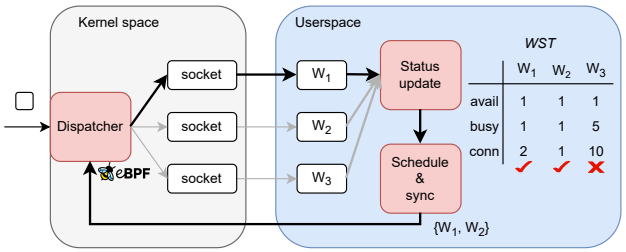


Figure 8: Userspace-directed I/O event notification.

the kernel (*e.g.*, epoll rr), as the kernel is designed for general-purpose use. In the past, we injected new functionality into the kernel through kernel modules. Faulty code, however, could directly cause a kernel crash, and reproducing it required constructing real data streams, making debugging difficult. Moreover, as the kernel evolves, the data structures that kernel modules depend on may change, leading to a proliferation of preprocessor conditionals in our code to maintain compatibility across kernel versions. In Linux 4.5, the kernel introduced the SO_ATTACH_REUSEPORT_EBPF hook [12], providing a chance to pass userspace input to the kernel for connection scheduling. eBPF can verify code to prevent kernel crashes, and it also provides a stable API to external applications by encapsulating helper functions. Its CO-RE mechanism [6] addresses data structure compatibility across kernel versions.

4 Overview of Hermes

4.1 Userspace-Directed Event Notification

Based on the above design principles, we propose Hermes, which constructs a "feedback" control loop between kernel space and userspace with three processing stages: (1) each userspace worker handles connections dispatched from the kernel and updates its real-time status (*e.g.*, the number of pending events) to a Worker Status Table (WST) located in a shared memory region; (2) a userspace scheduler periodically computes a set of worker candidates for accepting new connections based on the worker status fetched from the WST and synchronizes the computed results with the kernel; (3) in kernel space, a dispatcher selects a worker for connection dispatch from the userspace-computed results, and then the loop returns to stage 1. Fig. 8 illustrates the framework of Hermes. We elaborate on its three processing stages as follows.

Stage 1: worker status update. This stage maintains an inter-process table (*i.e.*, the WST) in the shared memory accessible to all workers. It records the runtime status of each worker and is updated during application request processing. The table's content is application-specific; here, we use our L7 LB as an example.

Each row in the WST represents a scheduling metric, and each column holds the metric values for a particular worker. For an L7 LB, three metrics are considered: a boolean variable *avail*, representing whether the current worker is available (a worker may become stuck in request handling or crash in extreme cases, making it unavailable to handle new requests); an integer *busy*, which counts the pending events that have been triggered but not yet handled by the worker (in an event-driven architecture, "events" refer to kernel-detected state changes on file descriptors, such as a socket becoming readable or writable), and an integer *conn*, which counts the concurrent connections during L7 processing.

Stage 2: connection scheduling across workers. This stage takes the WST from the previous stage as input, identifies a set of suitable worker candidates (*e.g.*, $\{W_1, W_2\}$ in Fig. 8), and synchronizes them to the kernel dispatcher for accepting new connections. For L7 LBs, we adopt three scheduling principles. First, we select workers that can respond promptly, filtering out those that are abnormal or slow to respond. Next, to mitigate the risk of future worker overload due to synchronized traffic surges across a large number of established connections, we prefer workers with fewer accumulated connections. Finally, we choose less busy workers with fewer pending events to reduce request processing latency.

Stage 3: connection dispatch in kernel space. This stage takes the previously selected worker list as input and implements custom connection dispatch logic overriding the default hash-based reuseport socket selection within the kernel through the eBPF hook SO_ATTACH_REUSEPORT_EBPF. For each new connection, a worker is selected from the userspace-provided worker list rather than blindly selecting from all workers via reuseport hashing.

We give a walkthrough example to show how Hermes outperforms epoll exclusive and reuseport in Fig. A4 in Appendix.

4.2 Epoll Event Loop of Hermes

Based on the above description of Hermes, we observe that two stages are performed in userspace: (1) each worker collects its own metrics and updates them in the WST (status update); (2) worker candidates for accepting new connections are selected based on the metrics of all workers in the WST, and the results are synchronized with the kernel (schedule & sync). Fig. 9 shows how these two stages are integrated into the original epoll event loop in userspace. Specifically, to determine whether a worker is hanging, we record the time when the worker enters the while loop (line 12) and later compare it with the current time to check if it has been stuck in the loop for an extended period. To count the pending events for a worker, we add the number of events returned by epoll_wait() to the total (line 14), and subtract 1 for each processed event (line 18). To track accumulated connections on a worker, we increment the count by 1 on each connection establishment (line 25) and decrement it on connection termination (line 37). The schedule & sync stage is performed at the end of the while loop (line 20).

As demonstrated, to implement Hermes, we have made only minor modifications to the original epoll event loop, and the kernel-space eBPF extension is modular and independent. Therefore, Hermes is easy to implement and introduces minimal additional overhead. Considering epoll's wide adoption, these modifications can also be incorporated into event frameworks such as libevent [18] and exposed to third-party applications through an SDK for broader applicability. In the next section, we provide a detailed explanation of our design choices and implementation techniques.

```
1  // initialize
2  // create and bind listening sockets ( listen_fds ) to all ports , omitted
3  // create the epoll instance
4  ep_fd = epoll_create () ;
5  for ( ls : listen_fds ) {
6      event->handler = accept_handler; // to handle the first event
7      // add the listening socket to the epoll instance
8      epoll_ctl (ep_fd, EPOLL_CTL_ADD, ls, event);
9  }
10 // infinite event loop
11 while (1) {
12     + shm_avail_update(current_time);
13     event_num = epoll_wait(ep_fd, event_list , MAX_EVENTS, timer);
14     + shm_busy_count(event_num);
15     // handle currently available events returned from epoll_wait ()
16     for (event : event_list ) {
17         event->handler(event);
18         + shm_busy_count(-1);
19     }
20     + schedule_and_sync();
21 }
22 // process new connections
23 accept_handler() {
24     conn_fd = accept () ;
25     + shm_conn_count(1);
26     // ... omitted
27     event->handler = other_handler; // e.g., read HTTP header/body
28     // add the new connection to the epoll instance
29     epoll_ctl (ep_fd, EPOLL_CTL_ADD, conn_fd, event);
30 }
31 // handle other events
32 other_handler() {
33     // ... omitted
34     if (err | fin) {
35         epoll_ctl (ep_fd, EPOLL_CTL_DEL, conn_fd, event);
36         close (conn_fd);
37         + shm_conn_count(-1);
38     }
39 }
```

Figure 9: Modified epoll event loop to achieve userspace-directed I/O event notification in Hermes.

5 System Implementation

5.1 Technical Issues

5.1.1 *Worker Status Metrics Selection.*

Select the most effective metrics. Based on years of operational experience, we find that high-performance, highly available L7 LBs should assign new connections to workers that are not in an abnormal state (*e.g.*, hung or crashed), have shorter processing time for pending tasks, and maintain fewer accumulated connections. Assigning connections to abnormal workers may lead to unprocessed requests, connection resets, or indefinite delays. Similarly, assigning connections to workers already handling many tasks can result in high response latency. However, we cannot rely solely on these two criteria. A worker may have only a few pending tasks but still maintain a large number of inactive connections. If these connections suddenly become active, they can easily overload or crash the worker. Additionally, workers typically manage connections using preallocated memory pools of fixed capacity. When connections are unevenly distributed among workers, overall system capacity can degrade significantly. In the past, we observed cases where some workers exhausted their connection pool resources and were unable to accept new connections, despite low CPU utilization. To meet these scheduling requirements, we must collect effective userspace metrics as the basis for making connection dispatch decisions.

Collect metrics with low overhead. The effectiveness of metrics directly affects the scheduling quality, while the overhead of collecting them impacts system performance. For example, Unique Set Size (USS) accurately measures the amount of memory used by a process but cannot be quickly obtained through a system call. To accurately measure the USS of each worker process, specific tools (such as smem [30], pmap [24]) or scripts to parse the smaps file of processes are required. To ensure that metrics collection does not

degrade connection scheduling or the overall performance of the L7 LB, the overhead of metrics collection should be kept low.

5.1.2 Connection Scheduling across Workers.

Concurrent metrics updates and retrievals. Due to address space isolation, each worker can only collect its own metrics; however, traffic scheduling requires visibility into the global status of all workers, necessitating inter-process communication. The accuracy of scheduling heavily depends on timely updates and retrievals of worker metrics. For multi-tenant L7 LBs, a massive number of connections and tasks per connection must be handled, leading to frequent updates of each worker's metrics. Therefore, an efficient inter-process metrics synchronization mechanism is essential.

Low-overhead, high-availability scheduling. The scheduling procedure to find the optimal worker can either be triggered by the kernel upon receiving a new connection or performed in userspace, with scheduling decisions subsequently updated to the kernel. For our L7 LBs, the number of new connections per second can reach O(100K). Performing kernel-side scheduling for each new connection would require frequent interactions with userspace to obtain worker status, which is not scalable. Therefore, we perform scheduling in userspace and periodically update the results to the kernel.

Given that the scheduler runs in userspace, there are two design options: using either a dedicated process or reusing existing worker processes. Binding a dedicated process to a CPU core significantly improves scheduler performance predictability. However, for cloud service providers, every CPU core is valuable for processing multi-tenant traffic, and using one solely for the scheduler is too costly. If the scheduler is not bound to a CPU core, it must compete with other processes for CPU resources, making it difficult to guarantee predictable scheduling performance. Additionally, a single scheduler responsible for all traffic is more vulnerable to a single point of failure, especially when handling malformed requests.

Reusing existing workers to execute the scheduler avoids extra costs and provides redundancy by embedding one scheduler in each worker, making it preferable for cloud service providers. However, this reuse strategy still faces the following issues:

First, the primary task of a worker is to handle service logic, not scheduling. When a worker also runs the scheduler, processing too many events in one epoll event loop can delay timely scheduling. Second, if scheduling decisions are updated too slowly, new connections may be assigned to the wrong worker, potentially causing worker overload and impacting system stability. Third, reusing workers may cause multiple processes to update scheduling results to the kernel concurrently. An efficient update mechanism is needed to avoid resource contention, which can degrade service performance. Fourth, coupling the worker and scheduler in a single process requires keeping the scheduler's algorithm simple and efficient to avoid affecting its execution frequency and the worker's high-throughput task handling. Fifth, embedding the scheduler into the worker requires placing its logic at the right point in the event loop, as execution timing impacts scheduling effectiveness.

5.1.3 Connection Dispatch via eBPF Hooks.

Make kernel-user interactions correct and efficient. In eBPF, *maps* are key-value data structures used for storing state and sharing data between kernel-space eBPF programs and userspace. eBPF maps are stored in the kernel but can be accessed from userspace via system calls. Using maps, we can pass userspace scheduling decisions to the kernel for new connection dispatch. However, in Hermes, kernel reads and userspace updates to the map occur independently, raising the risk of inconsistent data if an update overlaps with a kernel read. In addition to ensuring correctness, the efficiency of kernel-user interactions should also be considered.

Harness the limited programmability of eBPF. It is known that, for security and performance reasons, eBPF's programmability is limited: it does not support loops, recursive calls, or complex hash computations. Moreover, we need to avoid making eBPF programs overly complex, as this can impact the performance and stability of kernel functions. Implementing worker selection and connection dispatch in the kernel under these constraints is challenging.

5.2 Cascading Worker Filtering

5.2.1 The Selected Userspace Metrics.

For effective worker scheduling, we select three worker status metrics — timestamp of entering event loop, pending event number, and accumulated connection number.

To identify abnormal workers, we analyze their common traits and find that they often get stuck in the epoll event loop without returning for a long time. For example, we once encountered a case where request processing delay surged from 30ms to 440s because a worker was stuck on a read event. The worker's processing was slower than the upstream data writing, preventing timely buffer draining. To prevent one tenant's events from blocking others, we must detect worker hangs promptly. A key indicator of a worker hang is when the event loop stops rotating. Therefore, we record the timestamp when each worker enters the event loop (line 12 in Fig. 9) and, during scheduling, compare it with the current time to determine if the worker is stuck. As each worker embeds a scheduler, when a worker hangs, others can detect its hung status by accessing the timestamp it leaves in the shared memory.

Based on our experience, L7 task processing time can be estimated using the number of events in the event list, event handler type, and the size of packets associated with each event. Unfortunately, the event list contains only event descriptors with the handler type and a data pointer; the actual packet data can be accessed only during later processing, leaving the packet size unknown in advance. Historical data shows that the handler type alone is insufficient for accurate estimation without packet size information, but the number of events correlates well with processing time. Therefore, we use only the event number for estimation.

The accumulated number of connections can be easily collected by the worker by instrumenting at the connection establishment and release points, as shown in line 25 and line 37 in Fig. 9.

5.2.2 Worker Filtering Strategies.

Worker filtering logic. To filter out unavailable workers, we read their timestamps and compare them with the current time. If the difference exceeds a threshold, the corresponding worker is considered hung, as it has not reentered the while loop for an extended period (line 9-10 in Algo. 1). For the metrics of pending event number and connection number, smaller values indicate more preferred workers. Therefore, we use their average values as a baseline to select workers with values below the baseline. Moreover, to prevent

selecting too few available workers, we raise the baseline slightly by adding a small offset θ to the average (line 11-13 in Algo. 1).

Algorithm 1: Scheduler

```
1  Function schedule_and_sync():
2      W ← {w_1, ··· , w_n}
3      time, event, conn ← Read_SHM()
4      W ← FilterTime(time, W)
5      W ← FilterCount(conn, W)
6      W ← FilterCount(event, W)
7      SelWorker ← Array2INT(W)
8      BPF_MAP_UPDATE(SelWorker, M_Sel)
9  Function FilterTime(R, W):
10     return {w_i | currentTime() − R_i < Threshold, w_i ∈ W}
11 Function FilterCount(R, W):
12     Avg ← CalculateAverage({R_i | w_i ∈ W})
13     return {w_i | R_i < Avg + θ, w_i ∈ W}
```

Worker filtering order. Based on different weights for performance and stability across the three metrics, we design a cascading worker filtering algorithm (line 4-6 in Algo. 1), with earlier filtering stages prioritized for quickly excluding unsuitable workers. Since the timestamp identifies unavailable workers to which new connections should not be assigned, we use it for first-level filtering. The connection number identifies workers with fewer connections, helping reduce the risk of traffic surges from many concurrent connections. The pending event number identifies workers that can respond quickly, reducing the processing delay. Our production data shows that long-lived connections are common. Evenly distributing these connections enhances both system stability and capacity, which are crucial in a multi-tenant system. Accordingly, we perform second-level filtering based on the connection number and third-level filtering based on the pending event number.

5.3 Multi-Worker Cooperative Scheduling

5.3.1 Lock-Free Shared Metrics Data Structure.

In the modified epoll event loop (Fig. 9), workers update their metrics in the WST, while the scheduler retrieves the entire WST to filter workers based on global metrics. This requires concurrent updates and reads of shared variables. In our implementation, we use shared memory to store the WST and employ a lock-free approach for accessing shared variables to improve performance (Fig. 10).

Specifically, we partition the shared memory by worker, so that when a worker updates its own status metrics, it does not interfere with others, eliminating the need for write locks to block updates from other workers. For the scheduler reading all worker status, we also omit read-write locks, allowing other workers to update their status while the scheduler is reading. The benefit of not adding read-write locks is avoiding performance drops caused by blocking, although this may lead to data inconsistency during reads. However, for our system, we argue that such inconsistency has a negligible impact on scheduling decisions, for the following reasons:

First, the probability of a worker status update during the scheduler read is low, and the chance of reading data that is being updated is even lower. Empirical measurements show that reading data from a few workers takes only tens of ns, while updates to worker status metrics occur much less frequently, typically every few ms.

Second, even if, in rare cases, a few workers are updated during the scheduler read, it will not cause a significant deviation in the scheduling decisions. This is because the most recently updated data better reflects the workers' runtime status, and scheduling based on the latest worker status is unlikely to cause serious errors.

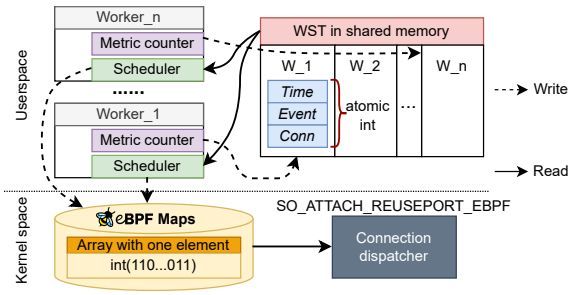


Figure 10: Concurrency handling of WST and eBPF maps.

However, since each worker has three status variables (*i.e.*, timestamp, event, connection), to prevent dirty data from updates to the same status variable during reading, each status variable is stored using atomic<int>, ensuring atomicity for both read and write. When one process writes to an atomic<int> variable, other processes will observe a consistent value when reading that variable during this period. These techniques allow Hermes to efficiently manage the WST in the shared memory without explicitly introducing locks.

5.3.2 Worker-Triggered Distributed Scheduling.

We address the technical issues of reusing existing workers to execute the scheduler as follows.

Concurrent scheduling for real-time responsiveness. We use the following strategies to ensure that the scheduler executes at a sufficient frequency, thereby maintaining scheduling accuracy: (1) we set the epoll_wait() timeout to 5ms, ensuring that each worker executes the event loop and performs a scheduling operation at least once every 5ms, even in the absence of I/O events; (2) multiple workers run their own schedulers concurrently, providing real-time responsiveness even when some workers are busy handling heavy tasks; (3) if all workers hang, our alert mechanism will detect the issue and identify the root cause: if the system is simply saturated due to high CPU or memory usage from heavy traffic, more VMs will be scaled out to increase capacity; if it is a failure, the faulty L7 LB will be taken out of service and replaced with healthy VMs.

Worker overload prevention with two-stage filtering. Even when multiple workers independently trigger scheduling, the combined frequency at which userspace updates the kernel for connection dispatch remains significantly lower than the new connection arrival rate (*e.g.*, O(100K)/s). If userspace passes only a single available worker to the kernel at a time, the kernel will direct all new connections to that worker, leading to worker overload. To address this, we propose a two-stage worker filtering mechanism. First, the scheduler in userspace selects multiple available workers (coarse-grained filtering) and passes them to kernel space, where new connections are distributed among them (fine-grained filtering). If the number of workers selected in coarse-grained filtering is insufficient, the kernel will fall back to reuseport-based scheduling.

Concurrency management of scheduling results. In Hermes, multiple workers may simultaneously output scheduling results to the kernel (see Fig. 10), and we need an efficient data structure to carry these scheduling results. We use 1 to represent an available worker and 0 for an unavailable one, storing these results in an

array whose length equals the number of workers, *e.g.*, {1, 1, 0, 0, 1} indicates that workers with ID 1, 2, and 5 are selected. However, this array-based data structure requires explicit locking to prevent race conditions, *e.g.*, when one worker is updating the array, other workers must wait for the update to complete, which degrades system throughput. To address this, we use a bitmap (*e.g.*, 11001) to record the available workers, which is then encoded as a 64-bit integer. By leveraging atomic<int>, we can manage concurrent updates across workers without explicitly using locks.

O(n) scheduling time complexity. As shown in Algo. 1, the scheduler is lightweight, as it only uses a single-level loop to check the worker status. Therefore, its complexity is only O(n).

Scheduling timing in epoll event loop. In Fig. 9, we place the scheduler at the end of the epoll event loop so that it executes only after the worker has handled the current batch of events. This design captures the worker's most up-to-date load status, allowing the scheduler to determine whether the worker remains suitable for handling new connections. In contrast, placing the scheduler at the beginning of the loop risks observing stale or misleading status. A worker may appear idle before calling epoll_wait(), but could immediately receive a burst of events afterward. If the scheduler marks the worker as available based on that outdated information, it may overload the worker and delay both new and ongoing connections.

5.4 Cost-Effective eBPF Implementation

Concurrency management of eBPF maps. The scheduling results generated in userspace need to be updated to the kernel. We use eBPF maps as an intermediary between userspace and kernel space to facilitate data sharing (see Fig. 10). Since eBPF maps do not natively provide an interface to pass int-type data, we use an eBPF map of type BPF_MAP_TYPE_ARRAY to transmit the results. To accommodate the int-based bitmap from userspace, we create an array with a single int element in the eBPF map (*i.e.*, M_{Sel} in Algo. 1). Since eBPF maps inherently support atomic<int>, concurrent reads and writes can be performed correctly without locks.

Final worker filtering with bitwise operations. The eBPF kernel module in Hermes dispatches new connections to userspace workers. The module first checks whether the number of available workers reported by userspace is sufficient (*e.g.*, > 1). This involves counting how many bits in the int-based bitmap provided by userspace are set to 1, representing how many workers have passed the coarse-grained filtering, denoted as n (line 3 in Algo. 2). If n is too small, the kernel takes no further action and falls back to the reuseport mode (which has already been initialized). If n satisfies the condition, fine-grained filtering is applied to the n workers (line 4). To evenly distribute new connections among these workers, we use the hash value of the packet's 4-tuple for load balancing (note that this hash value is precomputed by the kernel). The implementation consists of two steps: (1) use reciprocal_scale() to scale the hash value to the range of 1 to n, denoted as Nth (line 5); (2) locate the Nth non-zero bit in the bitmap, which identifies the selected worker ID (line 6). Due to eBPF's limited programmability, we rely on classic bitwise operations such as CountNonZeroBits() and FindNthNonZeroBit() based on [5, 14], while bpf_map_lookup_elem() and reciprocal_scale() are functions provided by the kernel.

Reuseport socket selection. Although we have computed the final worker ID for accepting new connections, the kernel does not interpret it directly, as the kernel can only perform scheduling over sockets. To bridge this gap, during Hermes program initialization, we allocate another eBPF map M_{socket} (map type BPF_MAP_TYPE_REUSEPORT_SOCKARRAY) to record the mapping between worker IDs and sockets. Finally, we inform the kernel of the scheduled socket through bpf_sk_select_reuseport() (line 7).

Algorithm 2: eBPF-based Connection Dispatch

```
   input: The eBPF map M_Sel with userspace selected workers, the eBPF map
          M_socket with worker to socket mapping.
1  Function conn_dispatch_socket_select(M_Sel, M_socket):
2      C ← bpf_map_lookup_elem(M_Sel)
3      n ← CountNonZeroBits(C)
4      if n > 1 then
5          Nth ← reciprocal_scale(4 − tuple.hash, n)
6          ID ← FindNthNonZeroBit(C, Nth)
7          return bpf_sk_select_reuseport(M_socket, ID)
```

6 Evaluation

6.1 Methodology

Hermes has been deployed across all Alibaba Cloud regions for over two years. Given that this paper focuses on intra-server load balancing, deploying multiple LBs yields results equivalent to those obtained with a single LB. Therefore, to demonstrate Hermes's benefits over epoll exclusive and reuseport, we redeploy one LB with epoll exclusive and another with reuseport, along with others with Hermes, in a single LB cluster (8 LBs in total for load sharing and failure recovery). Each LB is a 32-core VM with 128GB memory, running Linux 4.19. This LB cluster serves about 1500 tenants, handling hundreds of thousands of RPS of production traffic.

6.2 Performance of Hermes

Performance in specific cases. During evaluation with production traffic, we found that both epoll exclusive and reuseport exhibit performance degradation in certain cases. To assess the performance of Hermes in these specific cases, we collected and replayed traffic from them. Additionally, we replayed traffic at 2 to 3 times the original rate to emulate "medium" and "heavy" workloads. Table 3 shows the throughput of a single L7 LB and the end-to-end request processing time for the three solutions. If the request processing time exceeds 50% or the throughput differs by more than 20% from the optimal, it will be marked as "×". A solution that has never performed optimally in a case or has multiple "×" marks will be considered ineffective in that case (labeled as "×" too). These cases are prevalent in public clouds and cover almost all traffic models, although their distribution varies across regions due to workload differences (see Table 4). Epoll exclusive and reuseport perform poorly in the commonly occurring case 3 and case 4, respectively.

Case 1: Reuseport/Hermes > Exclusive. Case 1's traffic model is characterized by high CPS and low average processing time at the LB, typically seen during stress testing or traffic spike scenarios. Epoll exclusive performs poorly for two reasons. First, the low average processing time leaves workers idle. As a result, the LIFO wakeup behavior causes a few workers to handle most of the connections, leading to high single-core load. Second, the overhead of dispatching new connections is O(1) for Hermes and reuseport, but

Table 3: Hermes performance in specific cases compared with epoll exclusive and epoll with reuseport

		Light workload			*Medium workload*			*Heavy workload*		
		Avg (ms)	**P99 (ms)**	**Thr (kRPS)**	**Avg (ms)**	**P99 (ms)**	**Thr (kRPS)**	**Avg (ms)**	**P99 (ms)**	**Thr (kRPS)**
Case1:	*Epoll exclusive(×)*	0.890 (×)	9.71 (×)	76.1	2.62	25.22	129.5	7.09	27.45	207.8 (×)
High CPS,	*Epoll with reuseport(✓)*	**0.439**	**6.12**	**79.4**	**2.21**	**24.36**	**139.8**	5.10	26.93	281.4
Low Avg processing time	*Hermes(✓)*	0.595	6.95	78	2.59	25.72	128.6	**5.02**	**26.55**	**319.5**
Case2:	*Epoll exclusive(×)*	1.05	11.76	38.2	1.86 (×)	16.76 (×)	36.9	121.27 (×)	1030 (×)	16 (×)
High CPS,	*Epoll with reuseport(×)*	21.93 (×)	1480 (×)	10 (×)	88.7 (×)	1640 (×)	0.64 (×)	212.75 (×)	1820 (×)	0.27 (×)
High Avg processing time	*Hermes(✓)*	**0.99**	**10.82**	**38.3**	**1.05**	**10.94**	**37.8**	**10.47**	**376.56**	**24**
Case3:	*Epoll exclusive(×)*	0.413 (×)	2.228 (×)	193	0.587	61.398 (×)	631.2	0.974	67.649 (×)	830.7
Low CPS,	*Epoll with reuseport(✓)*	**0.259**	1.368	**206.4**	0.466	**14.685**	631.1	0.784	35.817	818.2
Low Avg processing time	*Hermes(✓)*	**0.259**	**1.289**	202.9	**0.453**	15.741	**631.8**	**0.770**	**28.251**	**932.9**
Case4:	*Epoll exclusive(✓)*	5.99 (×)	79.46 (×)	34.2	**13.68**	**300.12**	**33.1**	**95.08**	**593.04**	**16.5**
Low CPS,	*Epoll with reuseport(×)*	3.98 (×)	50.06	35.6	47.35 (×)	303.34	14.97 (×)	182.00 (×)	1240 (×)	5 (×)
High Avg processing time	*Hermes(✓)*	**1.99**	**40.43**	**39.4**	16.08	348.96	27.1	135.75	623.37	14.5

Table 4: Distribution of 4 cases in Table 3 across regions.

	Region1	***Region2***	***Region3***	***Region4***	***Avg***
Case1	19.45%	0.77%	6.6%	2.81%	7.4075%
Case2	0.55%	7.83%	2.9%	7.41%	4.6725%
Case3	65.61%	9.27%	60.8%	89.07%	56.1875%
Case4	14.39%	82.13%	29.7%	0.71%	31.7325%

O(#ports) for exclusive. The reason is that, for exclusive, all ports are registered with the epoll instance. In contrast, for Hermes and reuseport, each worker's epoll instance monitors only a single port. This is especially relevant under high CPS, as the dispatch overhead may affect performance. Hermes experiences higher latency under low and medium loads primarily due to the overhead of userspace-directed scheduling, though it remains lower than that of exclusive. Under heavy load, Hermes exhibits the best performance.

Case 2: Hermes > Exclusive > Reuseport. Case 2's traffic model exhibits high CPS and high average processing time, typically observed in stress testing or traffic spike scenarios involving time-consuming tasks (*e.g.,* compression). High average processing time keeps workers in a busy or hung state. Since Hermes avoids dispatching requests to busy or hung workers, it delivers the best performance. Unlike reuseport's stateless hashing, which may introduce significant queuing delays, epoll exclusive avoids scheduling connections to already hung workers. However, as in Case 1, the performance of exclusive degrades rapidly under heavy load.

Case 3: Hermes/Reuseport > Exclusive. In the case of low CPS and low average processing time, typically seen in finance and chat applications with long-lived connections, epoll exclusive performs unacceptably poorly. Its LIFO wakeup behavior causes connections to concentrate on a few workers, leading to worker overload during subsequent request surges. Both reuseport and Hermes distribute long-lived connections well, but under heavy load, Hermes exhibits more balanced load distribution due to userspace awareness.

Case 4: Hermes/Exclusive > Reuseport. In the case of low CPS and high average processing time, typically seen in web services, reuseport performs unacceptably poorly. The high processing time stems from CPU-intensive tasks performed by the LB, such as SSL handshakes and regex-based routing for the web services. Once established, these computationally expensive connections cannot be migrated, and only new ones can be scheduled. Reuseport performs the worst, as its stateless hashing introduces queuing delays when assigning new connections to the overloaded workers. Hermes and epoll exclusive are on par, with Hermes experiencing slightly higher delays under high load. This is because exclusive can promptly prevent busy workers from accepting new connections, while our closed-loop scheduling has a certain delay in identifying unavailable workers and notifying the kernel to enforce their exclusion.

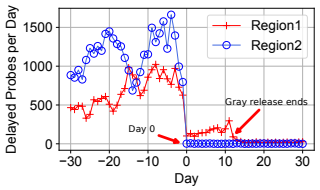


Figure 11: #Delayed probes per day before/after Hermes.

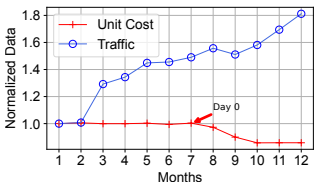


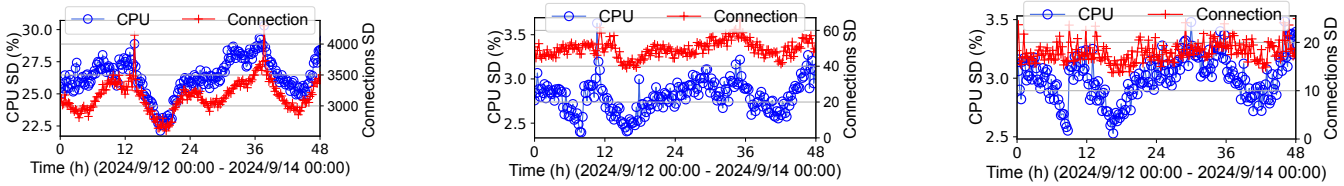
Figure 12: Unit cost of cloud infra before/after Hermes.

To summarize, the closed-loop design of Hermes offers strong adaptability. Hermes performs close to the best level across all cases, while both epoll exclusive and reuseport exhibit performance degradation in certain cases. Hermes is more suitable for multi-tenant cloud LBs with diverse and rapidly changing traffic patterns.

#Hung workers before and after Hermes deployment. To detect promptly when a worker hangs, we periodically send probes to all workers and measure their end-to-end delays. The LB contains no probe processing logic, so under normal conditions, the delay should not exceed 1ms. Internal network transmission delays exceeding 200ms are unacceptable to cloud service providers, as they may lead to client timeouts with a 499 status code, causing service disruptions and customer complaints. To show Hermes's effectiveness in balancing worker load and reducing worker hangs, we report the #probes exceeding 200ms over a day in two regions before and after Hermes deployment (epoll exclusive was enabled before Hermes), as shown in Fig. 11. In Region1 and Region2, delayed probes are reduced by 99.8% and 99%, respectively, indicating a significant improvement in tenant experience.

Hermes is deployed through a canary release, a strategy that gradually rolls out a new version to a subset of targets to reduce risk and ensure stability. During the rollout, new-version VMs with Hermes are gradually added to the L7 LB cluster, while old-version VMs are phased out. Once a VM is removed, it no longer handles new connections, but existing connections continue to transmit packets until the traffic on that VM fully drains. At that point, no more health check probes are sent to it. The time it takes for existing connections to drain depends on the client type. For example, some mobile clients drop connections quickly due to network changes, while IoT clients or cloud services may keep connections alive for a long time. In Region1, a few probes continued reaching old-version VMs even after the release, lasting up to 11 days until all connections expired. In Region2, connections drained faster, and probes quickly shifted to new VMs (see Fig. 11).

Unit cost of cloud infra before and after Hermes deployment. Before deploying Hermes, frequent worker hangs due to load imbalance forced us to maintain a low safety threshold for each LB;



(a) Epoll exclusive (b) Reuseport (c) Hermes

Figure 13: Hermes load balancing performance under production traffic (vs. epoll exclusive/reuseport).

Table 5: Overhead (CPU utilization) of Hermes components.

	Userspace			*Kernel space*
	Counter	**Scheduler**	**System call**	**Dispatcher**
Light	0.122%	0.272%	0.275%	0.005%
Medium	0.412%	0.381%	0.590%	0.019%
Heavy	0.897%	0.531%	0.965%	0.043%

specifically, we scaled out more LBs whenever CPU utilization exceeded 30%. After deploying Hermes, hung workers are eliminated (see Fig. 11), enabling us to increase the LB's safety threshold to 40%. As a result, we can handle the same traffic with fewer VMs, thus lowering cloud infra costs. Due to the increasing traffic volume, we are unable to demonstrate overall cost reduction. Instead, we present the unit cost for a region (= LB's total cloud infra cost / total traffic) to measure the cost savings from Hermes, as shown in Fig. 12 (data has been normalized to comply with company policies). After the Hermes release, the unit cost decreases monthly, with a peak reduction of 18.9%. It is important to note that while further increasing the safety threshold could continue to lower costs, it may impact disaster recovery across AZs, as sufficient redundancy must be reserved to handle traffic migrated from failed AZs.

Load balancing performance of Hermes in production. Fig. 13 shows the standard deviation (SD) of CPU utilization and #connections among LB workers over two days, with the three epoll modes enabled. We record the CPU utilization and #connections of each worker at every sampling point. A smaller SD indicates a more balanced distribution among the workers. As shown, the CPU SDs for exclusive, reuseport, and Hermes are 26%, 2.7%, and 2.7%, respectively, while the SDs of #connections are 3200, 50, and 20. Due to epoll exclusive's LIFO wakeup behavior, connections concentrate on a few workers, making its load balance significantly worse than the other two. Reuseport distributes new connections with hashing, which can theoretically achieve perfect balance. However, due to the varying durations of connections, with short connections closing quickly, the actual #connections on workers is less balanced. Hermes, on the other hand, selects workers with fewer connections, resulting in optimal load balance. This also helps reduce the CPU overload risk due to sudden traffic surges on these connections.

Overhead of Hermes. We use Perf's flame graph to analyze the CPU usage of each function. Table 5 reports the overhead of Hermes, which ranges from 0.674% to 2.436% in terms of CPU utilization under varying traffic loads. L7 LBs rarely experience sustained heavy loads, as they can proactively scale out. Heavy load cases only occur briefly during traffic migration from other AZs or sudden spikes. Therefore, the overhead is below 1% most of the time. Among its components, the dispatcher is the most lightweight as it only involves simple eBPF bitwise operations. The counter, which needs to record per-connection counts and uses atomic<int>, incurs increasing overhead as #connections grows. The scheduler involves simple

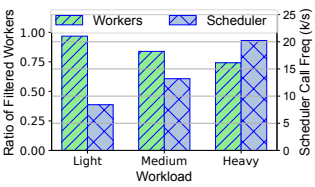


Figure 14: Filtered workers and scheduling frequency.

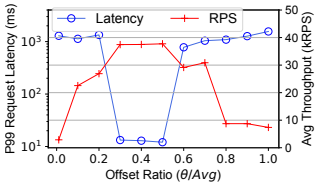


Figure 15: How to set the appropriate offset in Algo. 1.

computations with low overhead. Updating eBPF maps requires system calls and context switches, incurring additional overhead.

#Workers passing coarse-grained filtering. Fig. 14 shows the ratio of workers passing the coarse-grained filter to the total number of workers under different workloads. The ratio decreases when the workload increases, as more workers are in a busy state.

Call frequency of scheduler. Fig. 14 also shows that the scheduler's call frequency increases with the workload. As the duration of epoll_wait() significantly decreases under heavy traffic load, the scheduler is executed more frequently. In fact, a higher traffic load triggering more frequent scheduler execution aligns well with our needs. That is, to prevent the scheduling strategy from being updated too slowly under increasing load, which may lead to CPU core overload, a higher scheduling frequency must be maintained. Under heavy workload, the scheduling frequency can reach 20k/s.

Selection of offset θ. To prevent too few workers from being selected in coarse-grained filtering, which could result in a small number of workers handling too many connections, we introduce an offset θ into coarse-grained filtering. Fig. 15 shows the impact of θ/Avg on the average P99 latency and the average throughput. A smaller offset results in fewer workers passing the coarse-grained filter, causing new connections to concentrate on a small number of workers, which negatively affects request processing latency and system throughput. On the other hand, an excessively large offset allows workers with relatively high connection or event load to be selected, potentially delaying the processing of new connections and degrading overall system performance. As shown in Fig. 15, setting θ/Avg to 0.5 yields the best performance.

7 Experiences

Deployment issues from replacing epoll exclusive with Hermes. We encountered two issues during the deployment of Hermes to replace epoll exclusive in production:

Sudden load imbalance on tenants' backend servers. Some tenants reported sudden load imbalance across their backend servers behind our L7 LBs, with certain servers receiving 2-3x the traffic of others. The affected tenants typically operate large backend server pools that frequently scale out and in. Through debugging, we identified

the root cause. When a tenant's backend servers change, the controller updates the server list on each LB worker to ensure correct forwarding. Each worker then resumes round-robin distribution of connection requests starting from the first server on the updated server list. Since all workers share the same list order and start point, the first few servers receive disproportionately more traffic.

Under epoll exclusive, load balancing among backend servers largely relies on a single worker, as it handles most requests. This typically ensures that #requests handled by the worker significantly exceeds #backend servers, resulting in relatively even round-robin distribution. Hermes, however, distributes traffic evenly across all workers, leading to each worker handling fewer requests. Consequently, after list updates, the servers at the beginning of the list receive more traffic due to synchronized round-robin restarts. To mitigate this, we randomize each worker's starting offset after list updates, improving the fairness of traffic distribution.

More connections established with backend servers. Unlike epoll exclusive, which funnels most requests through only a few workers, Hermes distributes traffic evenly across all workers. This improves load balancing but reduces connection reuse, as connections established with backend servers are more evenly spread across workers. The impact becomes more pronounced when backend servers reside in on-premises IDCs and are accessed over the Internet, where costly handshake negotiations (*e.g.*, TCP or TLS) over long distances can significantly increase end-to-end request latency. The problem can be addressed by switching from per-worker connection pools to a shared pool among workers to enhance connection reuse.

Will the 64-bit atomic<int> limit the use of Hermes on high-capacity servers/VMs (*e.g.*, 128 cores)? In the current Hermes implementation, we use a 64-bit atomic<int> to synchronize coarse-grained filtering decisions from userspace to the kernel, supporting up to 64 workers without requiring locks. This design raises concerns about scalability in terms of the number of supported workers, given that modern server CPUs often feature 128 or more cores.

To address this, Hermes employs a two-level worker selection mechanism. Workers are grouped into sets of 64. When distributing new connections, we first select a worker group using a simple 4-tuple hash to choose an eBPF map (*i.e.*, level-1 selection). Within that group, we apply the original Hermes logic based on the atomic<int> recorded in the eBPF map to select a specific worker. Each worker group maintains an independent Worker Status Table (WST), updated exclusively by workers within the group. This additional layer of hashing enables support for more than 64 workers.

In practice, however, machines with very high core counts are seldom used due to the increased blast radius of failures. Instead, we typically deploy multiple 32-core VMs to build our L7 LBs, achieving comparable throughput while reducing cost (as a high-end machine is much more expensive than multiple mid-range machines offering equivalent performance) and limiting failure impact.

How worker failures impact tenant services? Our L7 LB needs to support numerous complex and frequently evolving application-layer protocols such as TLS and HTTP/3, necessitating frequent code updates all the time. Even with thorough testing and canary releases, worker crashes from corner cases cannot be fully eliminated. For instance, Nginx reported 51 bugs from 2024.9 to 2025.1 [39]. Our L7 LB is likely to encounter similar core dumps. When a worker crashes, both epoll exclusive and reuseport exhibit critical flaws.

Reuseport. The stateless hashing in reuseport may direct traffic to hung or crashed workers, leading to improper request handling. Based on our experience, it typically takes tens of seconds to detect worker crashes (*e.g.*, via proactive probing in the cloud [49, 56, 63, 65, 72, 76]). For an LB with 32 workers, this results in roughly 1/32 of tenant traffic being impacted during this detection time window.

Epoll exclusive. Epoll exclusive avoids assigning new connections to hung workers, but uneven load distribution can concentrate connections on a few workers. This raises the risk of severe tenant traffic disruption if one of these heavily loaded workers crashes. In one incident, a client issued an RFC-unsupported request, attempting to upgrade from an HTTP/2 connection to a WebSocket connection [59], which triggered a crash. Although only a single worker was affected, over 70% of connections had to be re-established. The tenant's server became overloaded due to an extremely high CPS, resulting in a recovery time of several dozen minutes.

Hermes. Hermes can balance the load across all workers, mitigating the risk of a large blast radius caused by traffic concentrating on a few workers. Additionally, Hermes detects worker anomalies by tracking the event loop entry timestamp, allowing it to quickly bypass unresponsive workers and minimize service downtime.

Will multi-tenancy/multiple ports undermine the benefits of Hermes? We initially hypothesize that the load imbalance caused by epoll exclusive, where new connections are disproportionately handled by the last worker added via epoll_ctl(), could be mitigated in a multi-tenant L7 LB by deliberately assigning different workers as the "last added" for different ports serving different tenants. In theory, this could scatter tenant traffic across workers. However, this assumption does not hold in practice due to two key reasons:

First, tenant traffic is highly dynamic. Once a worker is deliberately registered as the "last added" for a given port, this worker-to-port assignment remains fixed. If traffic across different ports is evenly distributed, this static assignment indeed effectively scatters tenant traffic across workers. However, in practice, whether a port receives significant traffic at runtime is unpredictable. Moreover, since #ports (O(10K)) vastly exceeds #workers (O(10)), it is almost certain that multiple ports will share the same "last added" worker. If those ports simultaneously receive bursty traffic, load imbalance will still occur, rendering such static worker assignment ineffective.

Second, tenant traffic is heavily skewed. A small number of top tenants contribute the majority of traffic (*e.g.*, the top three tenants account for 40%, 28%, and 22% of the overall traffic in one of our regions, and 23%, 10%, and 4% in another). This means that even if most ports are well scattered across workers, the dominant tenants will still concentrate load on only a few workers.

In contrast, Hermes provides stable and fine-grained control via userspace-directed scheduling, ensuring robust and balanced load distribution despite traffic dynamics or skew.

For page limit, we leave more experiences to Appendix.

8 Related Work

With the rise of NFV [70], traffic load balancing across CPU cores becomes critical for software middlebox performance [15, 16, 40–43, 45, 46, 51–53, 62, 67–69, 75]. Among these, most studies focus on packet-granularity load balancing for L3/L4 forwarding-intensive tasks [15, 16, 40–42, 52, 69, 75], *e.g.*, RSS++ [40] performs packet

scheduling at the NIC by reprogramming its RSS indirection table [16]. This differs from L7 load balancing, where userspace workers are selected by the kernel to handle incoming connections. Some studies build custom request schedulers to optimize tail latency [45, 46, 51, 53, 62, 67, 68]. These systems typically host more workers than CPU cores and allow workers to migrate between CPU cores. By contrast, our L7 LBs adopt a worker-to-CPU core binding strategy to reduce context switch overhead. Moreover, we prefer the fully validated Linux epoll for stability and lower maintenance costs, avoiding ties to specialized OSes [68] or hardware [45, 53].

A few studies focus on L7 LBs but do not address intra-server load balancing [48, 60, 71, 74]. [48] addresses scalability and availability of cloud L7 LBs. [60] leverages SmartNICs to offload L7 LBs. [71] relies on programmable switches to accelerate L7 LBs. [74] uses direct server return to reduce the load on L7 LBs. They all can benefit from Hermes. Based on our experience, userspace workload handling in L7 LBs consumes more CPU than kernel-space connection management, and thus requires more optimization efforts.

In 2015, the Linux community discussed fairness issues in epoll exclusive, and proposed epoll rr as a workaround [34]. However, epoll rr has not been merged into the kernel due to its cache-unfriendly behavior [23]. Cloudflare also reported load imbalance of epoll exclusive in its Nginx-based servers and proposed using reuseport as a remedy [37]. However, reuseport's stateless hashing may perform poorly in skewed workloads. Recently, the Linux community proposed sched_ext, allowing users to customize process schedulers with eBPF [13, 28]. The vision of userspace-defined scheduling in sched_ext aligns with Hermes. The difference is that sched_ext schedules processes, whereas Hermes schedules connections. As Linux's next-generation asynchronous I/O framework, io_uring [17] uses a default interrupt mode with a fixed wakeup order (similar to epoll, but in FIFO order), which may cause uneven process load. Hermes can also be extended to improve io_uring.

With the SO_ATTACH_REUSEPORT_EBPF hook introduced in Linux 4.5 [12], applications can override the default hash-based reuseport socket selection using eBPF programs. Facebook leverages this to steer traffic during update releases [61]. Nginx adopts this to schedule traffic with the same QUIC connection ID to the same worker [21]. Additionally, Cloudflare designs a similar programmable socket selection mechanism, sk_lookup [7], that enables flexible scheduling by directing incoming traffic to specific listening sockets [47]. In contrast to their static or application-defined traffic steering logic, Hermes performs closed-loop connection dispatch by dynamically adapting to worker status at runtime.

9 Conclusion

This work addresses the challenge of balancing connection distribution in multicore L7 LBs. To overcome the limitations of inter-worker load balancing in existing epoll mechanisms, we present Hermes, a userspace-directed I/O event notification framework with closed-loop connection dispatch. By leveraging eBPF to override the reuseport socket selection, Hermes adaptively selects the most available worker based on userspace status, reducing CPU overload risks. Hermes is also applicable to other epoll-based applications.

Acknowledgements. The authors would like to thank the shepherd, Michio Honda, and the anonymous reviewers for their constructive comments, which helped improve the paper.

Ethics. *This work does not raise any ethical issues.*

References

[1] [n. d.]. Accelerated Offload Connection Load Balancing in Envoy; Envoy's First Hardware Feature. https://community.intel.com/t5/Blogs/Tech-Innovation/Cloud/Accelerated-Offload-Connection-Load-Balancing-in-Envoy-Envoy-s/post/1464274. ([n. d.]).
[2] [n. d.]. Apache HTTP Server. https://httpd.apache.org/. ([n. d.]).
[3] [n. d.]. Application Load Balancer overview. https://cloud.google.com/load-balancing/docs/application-load-balancer. ([n. d.]).
[4] [n. d.]. Azure Application Gateway. https://learn.microsoft.com/en-us/azure/application-gateway/overview. ([n. d.]).
[5] [n. d.]. Bit Twiddling Hacks. https://graphics.stanford.edu/~seander/bithacks.html#SelectPosFromMSBRank. ([n. d.]).
[6] [n. d.]. BPF CO-RE. https://docs.ebpf.io/concepts/core/. ([n. d.]).
[7] [n. d.]. BPF sk_lookup merge commit. https://github.com/torvalds/linux/commit/e57892f50a07953053dcb1e0c9431197e569c258. ([n. d.]).
[8] [n. d.]. Dynamically program the kernel for efficient networking, observability, tracing, and security. https://ebpf.io/. ([n. d.]).
[9] [n. d.]. Envoy is an open source edge and service proxy. https://www.envoyproxy.io/. ([n. d.]).
[10] [n. d.]. epoll - I/O event notification facility. https://man7.org/linux/man-pages/man7/epoll.7.html. ([n. d.]).
[11] [n. d.]. epoll: add EPOLLEXCLUSIVE flag. https://lwn.net/Articles/667087/. ([n. d.]).
[12] [n. d.]. Expose socket options for setting a classic or extended BPF program for use when selecting sockets in an SO_REUSEPORT group. https://github.com/torvalds/linux/commit/538950a1b7527a0a52ccd9337e3fcd304f027f13. ([n. d.]).
[13] [n. d.]. Extensible Scheduler Class. https://www.kernel.org/doc/html/next/scheduler/sched-ext.html. ([n. d.]).
[14] [n. d.]. Hamming weight. https://en.wikipedia.org/wiki/Hamming_weight. ([n. d.]).
[15] [n. d.]. Introduction to Intel Ethernet Flow Director and Memcached Performance. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/intel-ethernet-flow-director.pdf. ([n. d.]).
[16] [n. d.]. Introduction to Receive Side Scaling. https://learn.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling. ([n. d.]).
[17] [n. d.]. io_uring - Asynchronous I/O facility. https://man7.org/linux/man-pages/man7/io_uring.7.html. ([n. d.]).
[18] [n. d.]. Libevent: An Event Notification Library. https://libevent.org/. ([n. d.]).
[19] [n. d.]. Nginx. https://nginx.org/en/. ([n. d.]).
[20] [n. d.]. Nginx core functionality. https://nginx.org/en/docs/ngx_core_module.html#accept_mutex. ([n. d.]).
[21] [n. d.]. Our Roadmap for QUIC and HTTP/3 Support in NGINX. https://blog.nginx.org/blog/our-roadmap-quic-http-3-support-nginx. ([n. d.]).
[22] [n. d.]. Overview of PostgreSQL Internals. https://www.postgresql.org/docs/current/overview.html. ([n. d.]).
[23] [n. d.]. [PATCH v3 0/3] epoll: introduce round robin wakeup mode. https://lore.kernel.org/lkml/54EDF7D8.60201@akamai.com/T/. ([n. d.]).
[24] [n. d.]. pmap(1) - Linux man page. https://linux.die.net/man/1/pmap. ([n. d.]).
[25] [n. d.]. poll, ppoll - wait for some event on a file descriptor. https://man7.org/linux/man-pages/man2/poll.2.html. ([n. d.]).
[26] [n. d.]. Redis - The Real-time Data Platform. https://redis.io/. ([n. d.]).
[27] [n. d.]. The Reliable, High Performance TCP/HTTP Load Balancer. https://www.haproxy.org/. ([n. d.]).
[28] [n. d.]. Sched_ext Schedulers and Tools. https://github.com/sched-ext/scx. ([n. d.]).
[29] [n. d.]. select, pselect, FD_CLR, FD_ISSET, FD_SET, FD_ZERO, fd_set - synchronous I/O multiplexing. https://man7.org/linux/man-pages/man2/select.2.html. ([n. d.]).
[30] [n. d.]. smem(8) - Linux man page. https://linux.die.net/man/8/smem. ([n. d.]).
[31] [n. d.]. The SO_REUSEPORT socket option. https://lwn.net/Articles/542629/. ([n. d.]).
[32] [n. d.]. Squid: Optimising Web Delivery. https://www.squid-cache.org/. ([n. d.]).
[33] [n. d.]. The thundering herd like problem when multi epolls on one fd. https://fa.linux.kernel.narkive.com/99dtET8i/the-thundering-herd-like-problem-when-multi-epolls-on-one-fd. ([n. d.]).
[34] [n. d.]. [v2,2/2] epoll: introduce EPOLLEXCLUSIVE and EPOLLROUNDROBIN. https://patchwork.kernel.org/project/linux-fsdevel/patch/7956874bfdc7403f37afe8a75e50c24221039bd2.1424200151.git.jbaron@akamai.com/. ([n. d.]).
[35] [n. d.]. What is ALB? https://www.alibabacloud.com/help/en/slb/application-load-balancer/product-overview/what-is-alb. ([n. d.]).
[36] [n. d.]. What is an Application Load Balancer? https://docs.aws.amazon.com/elasticloadbalancing/latest/application/introduction.html. ([n. d.]).

[37] [n. d.]. Why does one NGINX worker take all the load? https://blog.cloudflare.com/the-sad-state-of-linux-socket-balancing/. ([n. d.]).

[38] [n. d.]. Workload isolation using shuffle-sharding. https://aws.amazon.com/builders-library/workload-isolation-using-shuffle-sharding. ([n. d.]).

[39] 2024. Issues of Nginx. https://github.com/nginx/nginx/issues?q=is%3Aissue%20label%3Abug%20. (2024).

[40] Tom Barbette, Georgios P. Katsikas, Gerald Q. Maguire, and Dejan Kostić. 2019. RSS++: load and state-aware receive side scaling. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies (CoNEXT '19)*. Association for Computing Machinery, New York, NY, USA, 318–333. https://doi.org/10.1145/3359989.3365412

[41] Tom Barbette, Cyril Soldani, and Laurent Mathy. 2015. Fast Userspace Packet Processing. In *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS '15)*. IEEE Computer Society, USA, 5–16.

[42] Tom Barbette, Chen Tang, Haoran Yao, Dejan Kostić, Gerald Q. Maguire Jr., Panagiotis Papadimitratos, and Marco Chiesa. 2020. A High-Speed Load-Balancer Design with Guaranteed Per-Connection-Consistency. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 667–683. https://www.usenix.org/conference/nsdi20/presentation/barbette

[43] Adam Belay, George Prekas, Mia Primorac, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2016. The IX Operating System: Combining Low Latency, High Throughput, and Efficiency in a Protected Dataplane. *ACM Trans. Comput. Syst.* 34, 4, Article 11 (Dec. 2016), 39 pages. https://doi.org/10.1145/2997641

[44] Tianyi Cui, Wei Zhang, Kaiyuan Zhang, and Arvind Krishnamurthy. 2021. Offloading load balancers onto SmartNICs. In *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '21)*. Association for Computing Machinery, New York, NY, USA, 56–62. https://doi.org/10.1145/3476886.3477505

[45] Alexandros Daglis, Mark Sutherland, and Babak Falsafi. 2019. RPCValet: NI-Driven Tail-Aware Balancing of μs-Scale RPCs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 35–48. https://doi.org/10.1145/3297858.3304070

[46] Henri Maxime Demoulin, Joshua Fried, Isaac Pedisich, Marios Kogias, Boon Thau Loo, Linh Thi Xuan Phan, and Irene Zhang. 2021. When Idling is Ideal: Optimizing Tail-Latency for Heavy-Tailed Datacenter Workloads with Perséphone. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 621–637. https://doi.org/10.1145/3477132.3483571

[47] Marwan Fayed, Lorenz Bauer, Vasileios Giotsas, Sami Kerola, Marek Majkowski, Pavel Odintsov, Jakub Sitnicki, Taejoong Chung, Dave Levin, Alan Mislove, Christopher A. Wood, and Nick Sullivan. 2021. The ties that un-bind: decoupling IP from web services and sockets for robust addressing agility at CDN-scale. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference (SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 433–446. https://doi.org/10.1145/3452296.3472922

[48] Rohan Gandhi, Y. Charlie Hu, and Ming Zhang. 2016. Yoda: a highly available layer-7 load balancer. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*. Association for Computing Machinery, New York, NY, USA, Article 21, 16 pages. https://doi.org/10.1145/2901318.2901352

[49] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. 2015. Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis. *SIGCOMM Comput. Commun. Rev.* 45, 4 (Aug. 2015), 139–152. https://doi.org/10.1145/2829988.2787496

[50] Intel. [n. d.]. Data Plane Development Kit (DPDK). ([n. d.]). http://www.dpdk.org

[51] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive Scheduling for μsecond-scale Tail Latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 345–360. https://www.usenix.org/conference/nsdi19/presentation/kaffes

[52] Georgios P. Katsikas, Tom Barbette, Dejan Kostić, Rebecca Steinert, and Gerald Q. Maguire Jr. 2018. Metron: NFV Service Chains at the True Speed of the Underlying Hardware. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 171–186. https://www.usenix.org/conference/nsdi18/presentation/katsikas

[53] Antoine Kaufmann, SImon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. 2016. High Performance Packet Processing with FlexNIC. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. Association for Computing Machinery, New York, NY, USA, 67–81. https://doi.org/10.1145/2872362.2872367

[54] Praveen Kumar, Nandita Dukkipati, Nathan Lewis, Yi Cui, Yaogong Wang, Chonggang Li, Valas Valancius, Jake Adriaens, Steve Gribble, Nate Foster, and Amin Vahdat. 2019. PicNIC: predictable virtualized NIC. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM '19)*. Association for Computing Machinery, New York, NY, USA, 351–366. https://doi.org/10.1145/3341302.3342093

[55] Jonathan Lemon. 2001. Kqueue: A Generic and Scalable Event Notification Facility. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*. USENIX Association, USA, 141–153.

[56] Kefei Liu, Zhuo Jiang, Jiao Zhang, Shixian Guo, Xuan Zhang, Yangyang Bai, Yongbin Dong, Feng Luo, Zhang Zhang, Lei Wang, Xiang Shi, Haohan Xu, Yang Bai, Dongyang Song, Haoran Wei, Bo Li, Yongchen Pan, Tian Pan, and Tao Huang. 2024. R-Pingmesh: A Service-Aware RoCE Network Monitoring and Diagnostic System. In *Proceedings of the ACM SIGCOMM 2024 Conference (ACM SIGCOMM '24)*. Association for Computing Machinery, New York, NY, USA, 554–567. https://doi.org/10.1145/3651890.3672264

[57] Jianyuan Lu, Tian Pan, Shan He, Mao Miao, Guangzhe Zhou, Yining Qi, Shize Zhang, Enge Song, Xiaoqing Sun, Huaiyi Zhao, Biao Lyu, and Shunmin Zhu. 2024. CloudSentry: Two-Stage Heavy Hitter Detection for Cloud-Scale Gateway Overload Protection. *IEEE Transactions on Parallel and Distributed Systems* 35, 4 (2024), 616–633. https://doi.org/10.1109/TPDS.2023.3301852

[58] Mallik Mahalingam, Dinesh Dutt, Kenneth Duda, Puneet Agarwal, Lawrence Kreeger, T Sridhar, Mike Bursell, and Chris Wright. 2014. *Virtual extensible local area network (VXLAN): A framework for overlaying virtualized layer 2 networks over layer 3 networks*. Technical Report.

[59] P McManus. 2018. RFC 8441: Bootstrapping WebSockets with HTTP/2. (2018).

[60] YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and Kyoung-Soo Park. 2020. AccelTCP: Accelerating Network Applications with Stateful TCP Offloading. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 77–92. https://www.usenix.org/conference/nsdi20/presentation/moon

[61] Usama Naseer, Luca Niccolini, Udip Pant, Alan Frindell, Ranjeeth Dasineni, and Theophilus A. Benson. 2020. Zero Downtime Release: Disruption-free Load Balancing of a Multi-Billion User Website. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 529–541. https://doi.org/10.1145/3387514.3405885

[62] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 361–378. https://www.usenix.org/conference/nsdi19/presentation/ousterhout

[63] Tian Pan, Xingchen Lin, Haoyu Song, Enge Song, Zizheng Bian, Hao Li, Jiao Zhang, Fuliang Li, Tao Huang, Chenhao Jia, and Bin Liu. 2021. INT-probe: Lightweight In-band Network-Wide Telemetry with Stationary Probes. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*. 898–909. https://doi.org/10.1109/ICDCS51616.2021.00090

[64] Tian Pan, Kun Liu, Xionglie Wei, Yisong Qiao, Jun Hu, Zhiguo Li, Jun Liang, Tiesheng Cheng, Wenqiang Su, Jie Lu, Yuke Hong, Zhengzhong Wang, Zhi Xu, Chongjing Dai, Peiqiao Wang, Xuetao Jia, Jianyuan Lu, Enge Song, Jun Zeng, Biao Lyu, Ennan Zhai, Jiao Zhang, Tao Huang, Dennis Cai, and Shunmin Zhu. 2024. LuoShen: A Hyper-Converged Programmable Gateway for Multi-Tenant Multi-Service Edge Clouds. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 877–892. https://www.usenix.org/conference/nsdi24/presentation/pan

[65] Tian Pan, Enge Song, Zizheng Bian, Xingchen Lin, Xiaoyu Peng, Jiao Zhang, Tao Huang, Bin Liu, and Yunjie Liu. 2019. INT-path: Towards Optimal Path Planning for In-band Network-Wide Telemetry. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*. 487–495. https://doi.org/10.1109/INFOCOM.2019.8737529

[66] Tian Pan, Nianbing Yu, Chenhao Jia, Jianwen Pi, Liang Xu, Yisong Qiao, Zhiguo Li, Kun Liu, Jie Lu, Jianyuan Lu, Enge Song, Jiao Zhang, Tao Huang, and Shunmin Zhu. 2021. Sailfish: accelerating cloud-scale multi-tenant multi-service gateways with programmable switches. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference (SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 194–206. https://doi.org/10.1145/3452296.3472889

[67] Aleksey Pesterev, Jacob Strauss, Nickolai Zeldovich, and Robert T. Morris. 2012. Improving network connection locality on multicore systems. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys '12)*. Association for Computing Machinery, New York, NY, USA, 337–350. https://doi.org/10.1145/2168836.2168870

[68] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 325–341. https://doi.org/10.1145/3132747.3132780

[69] Hugo Sadok, Miguel Elias M. Campista, and Luís Henrique M. K. Costa. 2018. A Case for Spraying Packets in Software Middleboxes. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks (HotNets '18)*. Association for Computing Machinery, New York, NY, USA, 127–133. https://doi.org/10.1145/

3286062.3286081

[70] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. 2012. Making middleboxes someone else's problem: network processing as a cloud service. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '12)*. Association for Computing Machinery, New York, NY, USA, 13–24. https://doi.org/10.1145/2342356.2342359

[71] Xiaoyi Shi, Lin He, Jiasheng Zhou, Yifan Yang, and Ying Liu. 2025. Miresga: Accelerating Layer-7 Load Balancing with Programmable Switches. In *Proceedings of the ACM on Web Conference 2025 (WWW '25)*. Association for Computing Machinery, New York, NY, USA, 2424–2434. https://doi.org/10.1145/3696410.3714809

[72] Enge Song, Yang Song, Chengyun Lu, Tian Pan, Shaokai Zhang, Jianyuan Lu, Jiangu Zhao, Xining Wang, Xiaomin Wu, Minglan Gao, Zongquan Li, Ziyang Fang, Biao Lyu, Pengyu Zhang, Rong Wen, Li Yi, Zhigang Zong, and Shunmin Zhu. 2024. Canal Mesh: A Cloud-Scale Sidecar-Free Multi-Tenant Service Mesh Architecture. In *Proceedings of the ACM SIGCOMM 2024 Conference (ACM SIGCOMM '24)*. Association for Computing Machinery, New York, NY, USA, 860–875. https://doi.org/10.1145/3651890.3672221

[73] Enge Song, Nianbing Yu, Tian Pan, Qiang Fu, Liang Xu, Xionglie Wei, Yisong Qiao, Jianyuan Lu, Yijian Dong, Mingxu Xie, Jun He, Jinkui Mao, Zhengjie Luo, Chenhao Jia, Jiao Zhang, Tao Huang, Biao Lyu, and Shunmin Zhu. 2022. MIMIC: SmartNIC-aided Flow Backpressure for CPU Overloading Protection in Multi-Tenant Clouds. In *2022 IEEE 30th International Conference on Network Protocols (ICNP)*. 1–11. https://doi.org/10.1109/ICNP55882.2022.9940340

[74] Ziqi Wei, Zhiqiang Wang, Qing Li, Yuan Yang, Cheng Luo, Fuyu Wang, Yong Jiang, Sijie Yang, and Zhenhui Yuan. 2024. QDSR: Accelerating Layer-7 Load Balancing by Direct Server Return with QUIC. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. USENIX Association, Santa Clara, CA, 715–730. https://www.usenix.org/conference/atc24/presentation/wei

[75] Lei Yan, Yueyang Pan, Diyu Zhou, George Candea, and Sanidhya Kashyap. 2024. Transparent Multicore Scaling of Single-Threaded Network Functions. In *Proceedings of the Nineteenth European Conference on Computer Systems (EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 1142–1159. https://doi.org/10.1145/3627703.3629591

[76] Shunmin Zhu, Jianyuan Lu, Biao Lyu, Tian Pan, Chenhao Jia, Xin Cheng, Daxiang Kang, Yilong Lv, Fukun Yang, Xiaobo Xue, Zhiliang Wang, and Jiahai Yang. 2022. Zoonet: a proactive telemetry system for large-scale cloud networks. In *Proceedings of the 18th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT '22)*. Association for Computing Machinery, New York, NY, USA, 321–336. https://doi.org/10.1145/3555050.3569116

Appendices

Appendices are supporting material that has not been peer-reviewed.

A Supplementary Code Examples

A.1 Original Epoll Event Loop

```
// initialize
// create and bind listening sockets ( listen_fds ) to all ports , omitted
// create the epoll instance
ep_fd = epoll_create () ;
for ( ls : listen_fds ) {
    event->handler = accept_handler; // to handle the first event
    // add the listening socket to the epoll instance
    epoll_ctl (ep_fd, EPOLL_CTL_ADD, ls, event);
}
// infinite event loop
while (1) {
    event_num = epoll_wait(ep_fd, event_list , MAX_EVENTS, timer);
    // handle currently available events returned from epoll_wait ()
    for (event : event_list ) {
        event->handler(event);
    }
}
// process new connections
accept_handler () {
    conn_fd = accept () ;
    // ... omitted
    event->handler = other_handler; // e.g., read HTTP header/body
    // add the new connection to the epoll instance
    epoll_ctl (ep_fd, EPOLL_CTL_ADD, conn_fd, event);
}
// handle other events
other_handler () {
    // ... omitted
    if (err | fin) {
        epoll_ctl (ep_fd, EPOLL_CTL_DEL, conn_fd, event);
        close (conn_fd);
    }
}
```

Figure A1: Simplified epoll event loop for connection processing in a worker process.

A.2 LIFO Wakeup Behavior of Epoll Exclusive

```
static void __wake_up_common(wait_queue_head_t *q, unsigned int mode, int
        nr_exclusive, int wake_flags, void *key)
{
    wait_queue_t *curr, *next;
    list_for_each_entry_safe (curr, next, &q->task_list, task_list ) {
        unsigned flags = curr->flags;
        if (curr->func(curr, mode, wake_flags, key) &&
            (flags & WQ_FLAG_EXCLUSIVE) && !--nr_exclusive)
            break;
    }
}
```

Figure A2: Epoll exclusive walks the list of waiters for the shared socket and wakes up the first waiter that is idle (blocked on epoll_wait()) and ready to handle the incoming events (Linux kernel source code).

B Walkthrough Examples of Existing Epoll Modes and Hermes

An example demonstrating epoll exclusive and reuseport. We use Fig. A3 to show the workload imbalance with epoll exclusive and reuseport. The input sequence of requests from different connections is a, b_1, b_2, b_3, b_4, and the processing time of a is twice that of each b. For epoll exclusive, the new connections will be prioritized to W_3 unless it is busy. For epoll with reuseport, when W_1 is already processing a, its stateless hashing may still dispatch new connections to this busy worker.

Figure A3: Workload imbalance cases with two epoll modes.

A walkthrough example of Hermes. We use the previous example on request scheduling (in Fig. A3) to show how Hermes's scheduling strategy outperforms epoll exclusive and epoll with reuseport. The input sequence of requests from different connections is a, b_1, b_2, b_3, b_4. Request a contains two events, with each event taking $2t$ to process. Each request b also contains two events, with each event taking $1t$ to process. A worker process is considered unavailable if its processing time exceeds $2t$. The step-by-step request scheduling process with Hermes is shown in Fig. A4.

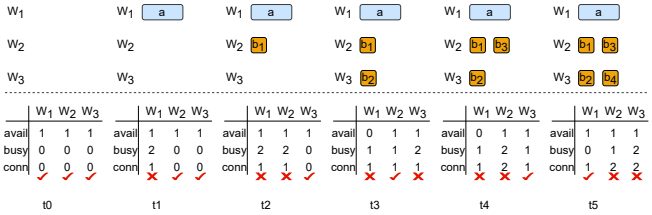


Figure A4: Balanced workload scheduling with Hermes.

The scheduling starts at t0, with all worker processes available and both *busy* and *conn* initialized to 0 for all workers. W_1, W_2, W_3 are all able to accept new connections.

At t1, assume W_1 takes request a, its *busy* and *conn* counts increase. As the request of the new connection involves two events, its *busy* rises to 2 and *conn* rises to 1, leaving only W_2 and W_3 available to accept new connections.

At t2, suppose W_2 takes the next request b_1, with its *busy* and *conn* increased to 2 and 1, respectively. Now, only W_3 can accept new connections.

At t3, after W_3 accepts request b_2, we find that the request a on W_1 has consumed too much CPU time, making W_1 unavailable. At this time, as W_2 has processed one event, its *busy* decreases to 1, making W_2 the next scheduling option.

At t4 and t5, W_2 and W_3 handle the next b_3 and b_4. At t5, W_1 becomes available again as all events have been processed.

This example demonstrates that Hermes can evenly distribute requests to achieve inter-worker load balancing.

C Additional Experiences

Why export worker status metrics from userspace rather than query them directly in the kernel? While the metrics used by Hermes can, in theory, be obtained within the kernel, collecting them there incurs overhead higher than expected. This overhead stems from a fundamental asymmetry between the kernel and

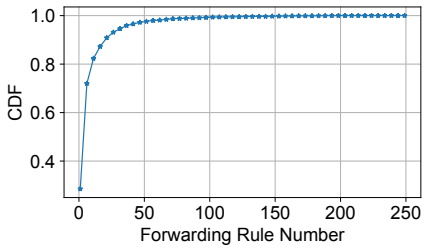


Figure A5: CDF of #forwarding rules per port in a region.

userspace: the kernel operates with a global scope and has visibility into all system activities, but it lacks the application-specific context necessary to identify which subset of data is relevant for a given scheduling requirement without explicit guidance from userspace. Consequently, even if metrics collection were performed in the kernel, userspace programs would still need to convey their context and intent, resulting in frequent and costly kernel-userspace interactions.

Hermes instead collects metrics and makes scheduling decisions entirely in userspace, passing only the scheduling results back to the kernel. This design yields two key benefits. First, it reduces kernel-userspace communication by synchronizing only the scheduling results instead of a large set of raw metrics. Second, it retains complex scheduling logic in userspace, circumventing the limited programmability of eBPF and enabling significantly greater flexibility. For instance, our scheduler exposes an HTTP interface that allows dynamic policy updates, supports fallbacks to reuseport, and facilitates rapid iteration of future scheduling algorithms.

Load balancing vs cache locality. Cache-aware connection dispatch, such as consistently directing requests with the same DIP and Dport (which may access the same data) to the same worker, can improve efficiency in many L7 scenarios (*e.g.*, databases and CDNs) by maximizing cache hits. However, based on our experience, such cache locality is largely unnecessary for L7 LBs in multi-tenant public clouds, for two reasons. First, there is no data locality: the L7 LB acts as a middlebox and does not cache any tenant data. Second, there is no code locality: as shown in Fig. A5, tenant forwarding rules vary significantly, with different rules triggering different code paths. As a result, even if all traffic from a tenant is directed to the same CPU core, instruction and data cache reuse remains limited, offering no clear efficiency gain. In practice, metrics such as throughput, latency, and SLA adherence are far more critical to both tenants and cloud service providers. Accordingly, Hermes is designed with the primary goal of achieving balanced load distribution rather than preserving cache locality.

Despite this, as a versatile framework, Hermes can still be extended to support cache-aware scheduling for workloads that may benefit from it. It introduces a group-based scheduling model that enables balancing cache locality and load distribution. Specifically, Hermes partitions workers into groups. Upon a new connection, the dispatcher uses the DIP and Dport to select a group via hashing, and then selects a worker within the group using a bitmap that reflects runtime worker load status from userspace. For example, in Fig. A6, a new connection is mapped to Group1 via hashing, and Worker2 is chosen based on the bitmap 01 to accept the connection. This design ensures that requests with the same DIP and Dport

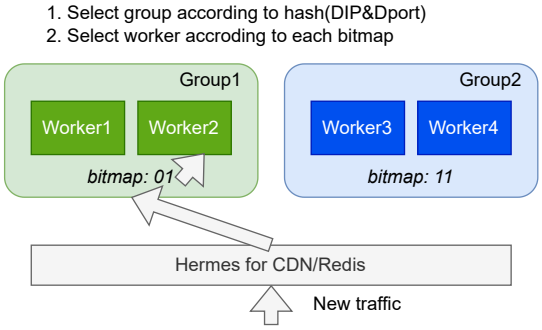


Figure A6: Hermes for applications requiring cache locality.

are directed to the same group (preserving locality), while load is still distributed across multiple workers (ensuring balance). The grouping granularity controls the trade-off.

By comparison, existing kernel-based mechanisms provide partial and inflexible support for cache locality. In epoll exclusive, the LIFO wakeup policy unintentionally concentrates traffic onto tail-added workers, increasing their cache hits. In reuseport, hashing on the 4-tuple consistently routes similar connections to the same worker, offering a degree of locality. However, both mechanisms suffer from imbalanced load distribution and rely on hardcoded in-kernel policies that are difficult to adjust or override. Hermes generalizes these mechanisms: with a single group, it behaves like standard Hermes; with one worker per group, it reduces to reuseport. Unlike kernel-fixed strategies, Hermes enables fine-grained control over the trade-off between cache locality and load balance.

Exception handling case 1: single worker hangs. Many popular L7 LBs (*e.g.*, Nginx) use epoll in edge-triggered mode to handle events. Once a socket state changes (*e.g.*, data arrival), the worker is notified and must drain the buffer completely; otherwise, no further events will be delivered. In extreme cases where the upstream network is fast but downstream processing is time-consuming (*e.g.*, SSL encryption or data compression offloaded to the L7 LB), the rate at which data is consumed from the buffer can fall behind the rate at which it is filled. As a result, the worker becomes trapped in a loop of reading, processing, and reacting to new data, unable to return to the main event loop. When this occurs, requests from both new and existing connections assigned to the worker cannot be processed in a timely manner.

Hermes offers solutions to mitigate this abnormal condition:

1. *For new connections.* Hermes tracks the timestamp of each worker's most recent entry into the epoll event loop. If this timestamp is not updated for a long time, Hermes marks the worker as unavailable and stops assigning new connections to it. Similarly, epoll exclusive will not assign new connections to a busy worker. In contrast, reuseport employs stateless hashing, which may continue to schedule new connections to the already overloaded worker, exacerbating the worker hang issue.

2. *For existing connections.* To minimize overhead from cache misses, context switches, and inter-core synchronization, modern L7 LBs typically bind the handling of one connection to a single CPU core. This affinity makes it difficult to migrate established connections between workers. When a worker hangs, events on its associated connections are stalled until a long-running task completes. In our cloud, there have been cases where the request

processing delay surged from 30ms to 440s. If the delay exceeds the client's timeout threshold, frequent disconnections occur.

To mitigate the impact, Hermes triggers proactive service degradation: when a CPU core remains highly utilized, Hermes sends TCP RSTs to terminate a subset of connections, allowing them to reconnect and be rescheduled to healthy workers (filtered by Hermes). Although proactively disconnecting connections is noticeable to tenants, most can tolerate it. Even if some connections are quickly disconnected and reconnected, as long as application-layer requests return valid results, the end user experience is not significantly affected. In other words, L7 users prioritize the eventual success of their requests and the correctness of service logic, even at the expense of L4 connection stability. Additionally, tenants that frequently trigger worker hangs are migrated to a sandbox, enabling physical isolation to prevent interference with others.

Exception handling case 2: all workers hang. In production, we have observed severe traffic bursts that drive CPU utilization of all workers on an L7 LB device beyond acceptable thresholds. At this point, node-local scheduling becomes ineffective, necessitating a transition to cluster-wide scheduling.

Hermes offers solutions to this abnormal situation:

1. *For network attacks.* As traffic ingress points, L7 LBs are frequent targets of SYN flood and Challenge Collapsar (CC) attacks, which can lead to CPU exhaustion across all workers. Hermes leverages anomaly detection techniques to identify malicious traffic patterns and promptly migrates the directly affected tenants to isolated sandboxes. This prevents them from degrading the performance of other tenants. Once the migration is complete, CPU usage on the original workers returns to normal.

2. *For traffic surges from legitimate services.* For legitimate workload surges, Hermes employs a phased scaling strategy to alleviate pressure on workers. Each tenant may purchase one or more L7 LB instances, which are deployed on VM-based L7 LB devices provisioned within our cloud infrastructure. To isolate failures across tenants, cloud service providers usually adopt shuffle sharding [38], ensuring that each tenant's L7 LB instance is deployed on a subset of VMs, which are further managed in groups.

To handle worker overload progressively, Hermes enters the following three phases:

Phase1: During traffic surges, we first perform a scale-out operation by distributing the traffic load of the overloaded L7 LB instance across other existing VM groups.

Phase2: If Phase1 fails to alleviate the overload, we scale up the instance by adding more available VMs to its existing VM groups.

Phase3: If overload persists, we provision new VMs and create additional VM groups for the instance to absorb the overflow traffic.