Load Balancers Need Cheap In-Band Feedback Control

Paper #74

ABSTRACT

Cloud load balancers (LBs) are critical components of interactive services today, distributing client requests over a server pool to improve performance and availability. There has been significant interest in building scalable LBs. However, little attention has been paid to the request-routing policies that LBs use. Today's policies are simple and static, like spreading connections evenly across servers.

Over the next decade, application compute will become increasingly granular, and request performance will be affected by software and system variability at time scales of 100μs–1ms. As a result, we believe that the status quo of static request-routing policies will be simply unviable to support high end-to-end application performance.

We advocate for a different approach: in-band feedback control operating purely locally at LBs to adapt request-routing to server performance. A key challenge is that high-speed LBs cannot directly measure server performance, since they only process requests and not responses. We present an initial design of an LB that adapts to a server latency inflation of 1 ms and reduces tail latencies in milliseconds, while only observing client-to-server traffic.

1 Introduction

Cloud load balancers (LBs) are crucial components of large interactive distributed services. LBs enable application logic to scale out to a pool of replicated servers, improving application performance by avoiding hot spots. From the perspective of users, LBs hide churn in the set of servers in the pool, providing higher availability for the service.

LBs are deployed widely to scale out user-facing applications running inside a compute cluster. LBs may run as *frontends*, routing client requests arriving from the Internet to the server pool [92, 60, 9, 87, 47, 49]. LBs may also run as *tier-to-tier* balancers, scaling out a single application tier (e.g., an in-memory database) of a complex application, routing requests sent from other tiers [6, 11, 13, 55, 10, 24, 41, 54, 39, 28]. LBs may run at layer-4 (using connection 4-tuples) or layer-7 (e.g., using HTTP-based service identifiers) to map requests to servers.

The networking community has witnessed significant work in designing scalable LBs [92, 60, 65, 85, 87, 47, 49, 107]. However, the policies LBs use to route requests to servers are simple and static: balance active connections among servers.

We believe that this status quo of static request-routing will be unviable in the next decade. With the advent of microservices, serverless, and rack-scale computing [73, 34, 37, 22, 75, 84, 109, 81, 70, 79], application compute tasks will become increasingly granular (§2.1). With finer granularity, server performance will be much more vulnerable to regression from system and software variability at time scales of 100μs–1 ms (§2.2). Variability will worsen tail latencies. Alternative techniques to deal with variability, such as overprovisioning, demand-driven scaling [5], and request duplication [58] will simply not work at these time scales. Applications will need LBs to adapt request-routing to highly-variable server performance.

Adapting to server performance requires that LBs are aware of it in the first place. Unfortunately, adding application-level instrumentation and shipping performance data to centralized controllers or even the LBs themselves presents significant challenges in data collection and freshness (§2.3).

Instead, we argue that LBs must implement *distributed feedback control* to directly react to server and network performance, through local measurement and control of their own request-routing policy. Such an approach has the potential to significantly improve application performance over the next decade, even without co-opting servers, clients, applications, or the network. We take inspiration from the long history of distributed feedback control in our community, e.g., for TCP congestion [71, 78, 86] and distributed wide-area traffic engineering [61, 77].

However, measuring end-to-end server performance directly at LBs is complicated by the fact that high-speed LBs are designed to minimize or avoid processing response traffic from servers to clients (§2.4), to minimize CPU consumption and reduce response latency [92].

This paper takes a first step towards in-band feedback control at LBs by presenting a technique to measure end-to-end client-server response latency without observing response traffic (§3). Our key insight is that it is possible to substitute the measurement of the delay between request and response by the delay between the request and a packet that a client transmits due to the response—a packet we call a *causally-triggered transmission.* We propose techniques to identify causally-triggered transmissions, enabling highly accurate ongoing measurements of server response latencies. We also design a simple control loop that adapts request-routing based on server response latencies.

Experiments show that even this simple controller can react to a server latency inflation of 1 ms and shift traffic in milliseconds, reducing tail latencies (§4). We conclude the paper with several open research questions on the design of measurement and controllers in this context (§5).

2 The Case for In-Band Feedback Control

2.1 Granularity and Network Delays

Over the next decade, application compute will be increasingly granular. Modern user-facing services break complex application logic into loosely-coupled components, termed microservices [73, 34], that collaboratively implement the application by exchanging messages over the cluster's interconnecting network. A single user-facing request may involve calls to thousands of microservices [12, 3, 14, 20], with the slowest microservice dominating response time [58]. To provide end-to-end latencies in the milliseconds, each microservice will need to finish its compute in microseconds. Systems support for "granular computing," e.g., serverless [37, 22], rack-scale [75, 84, 109, 81, 70, 79], anticipates and pushes this trend forward.

In the limit, the completion time of a compute task will be comparable to the round-trip propagation delay to the component that requested the task [89, 70]. It becomes important that each request not only reach a "good" server, but also traverse a lightly-loaded network path. A slightly slower server that is reachable faster may be preferable to a fast server with a congested network path. Today's LBs completely ignore the effects of network paths except at very coarse spatial granularities [26, 27].

Further, the frequency of load-balancing decisions increases with finer compute granularity. This makes it critical to get server selection "right" for each request, to provide good end-to-end application performance.

2.2 Performance Variability

Applications today run deep software stacks. Stemming from the need to ease portability and scalability, containerization [53, 32, 40, 42] packages application components and their software dependencies into self-contained execution environments. However, supporting feature-rich connectivity between containers requires new software layers in the network stack, including virtualized network interfaces (termed the *container network interface* [16]) and the service mesh [48, 45, 7]. These additional layers support translation between container and provider network addresses (providing containers the illusion of their own IP address space [38]), access control policies [8], and authentication between containers [21, 35]. Effectively, each network message between containers may traverse the software network stack twice as many times as packets between baremetal machines [110, 18].

The longer the lifetime of a message in software, the more variable its processing latency, due to inefficiencies in scheduling interrupts and threads (in user and kernel space) that must process the message. On Linux today, recovering from a single preemption may take hundreds of microseconds to a few milliseconds [52, 75, 80, 56]). Increasing the time spent by messages in the network stack also amplifies the impact of background tasks such as compaction and garbage collection [2, 88, 58] on processing latency. Recent works that improve operating system scheduling to shrink tail latencies [94, 75, 84, 62] use user-space networking stacks, and hence coexist poorly with multi-tenancy [95]; they are inapplicable to generic cloud deployments.

Unfortunately, the shrinking granularity of application compute (§2.1) makes request-processing performance increasingly vulnerable to low-level system variability over time. Variability is challenging to get rid of completely [58]. The consequence is that server request-processing performance may vary fast, e.g., in hundreds of microseconds, or within a few round-trip times in modern clusters. Typical approaches to handle performance variability are not viable at this time scale. Overprovisioning resources can get expensive [19]. Automatic scaling [53] to spin up new VMs and containers may take tens of seconds to take effect [5, 29]. Compared to sending the request to a fast server in the first place, timeout-based request duplication [58] will effectively double the response latency for a duplicated request when compute and network delays are comparable (§2.1).

We believe that adaptive request-routing at LBs is architecturally the right approach to address variability of the kinds discussed above. Instead of balancing active connections evenly across servers [60, 9, 92] as today's LBs do, the LBs of the future should *react to server performance directly*, since all servers are not equal at all times. Server performance may change in a few round-trip times. Yet, LBs reacting to server performance can make many favorable request-routing decisions even within just a few round-trip times, corresponding to all the requests arriving during this period. However, to adapt to changing server performance, LBs must first observe it—a challenging task that we discuss below.

2.3 Avoiding App Modification

One may wonder if performance information from applications might be obtained at LBs through out-of-band channels. For example, applications themselves may publish occupancy of application-level queues or server CPU/memory utilization to external monitoring systems, or even directly to LBs [101, 64, 26, 25]. Alternatively, centralized load-balancing controllers [44, 93] may consume performance information from servers and propagate control signals to update request-routing policies at LBs.

Implementing changes to applications to support such use cases is nontrivial. Anecdotally, getting wide deployment of "housekeeping" functionality into applications requires significant homogeneity in the deployed software environment [99]. Any degree of heterogeneity compounds the challenges of instrumenting source code [91, 76, 97]. The decomposition of a complex application into microservices reflects the organizational structure of the teams managing the different parts of the application's logic. LB designs that require instrumentation of source code across teams will face uphill battles for practical deployment.

If performance metrics could indeed be collected, the reactivity of load balancing would still depend on how quickly LBs can access fresh performance data or control signals. Server performance data would need to be collected centrally from across the server pool, and either the raw performance data or updated request-routing policies (computed by a centralized controller) must be propagated to LBs. Such propagation may occur through a storage or a pub/sub system. Given the performance of cloud storage systems today, e.g., [23, 1], we estimate that propagating data from applications via storage to a controller or even to each LB will take at least 10–100 milliseconds. Such data or signals will be too stale, given the rapidity of server performance variation (§2.2).

2.4 Minimizing Traffic Footprint

To avoid the staleness of centralized data collection and controllers, it is appealing to ask whether LBs can measure server performance directly themselves. Unfortunately, this is not easy to do, as we explain below.

Strictly speaking, LBs are just processing overheads for applications: they are glue logic to move data to and from applications running at servers. LBs must scale to handle large bandwidth of traffic and avoid additional latency due to their presence. Taming the CPU utilization of software LBs is a significant operational concern [92, 65, 98, 57]. It is especially crucial for frontend LBs since they handle *every* packet sent to the service from the Internet, including volumetric DDoS attack traffic such as SYN floods. However, the concerns of reducing CPU cycles and keeping latencies in check also apply to tier-to-tier LBs.

Specifically, many LBs implement *direct server return (DSR)*, an optimization that enables servers to send response traffic directly to clients bypassing the LB [92, 30, 39, 28]. DSR cuts the bandwidth and CPU requirements on LBs since the LBs need not process bandwidth-intensive response traffic. Moreover, DSR removes an additional hop on the server-to-client path, which would otherwise add latency.

Unfortunately, optimizations to improve LB performance by making them "low touch" on application traffic will also hinder the visibility that LBs have over server performance. Specifically, DSR makes it challenging for LBs to correlate requests with responses (since the latter are unobservable). Hence, it is difficult to measure a server's response latencies or request-processing rates directly at the LB. The assumption of observing both directions of traffic is ubiquitous in measurement works that aim to passively estimate round-trip times of connections from an intermediate vantage point [96, 82, 108, 90, 50, 72, 74, 104, 66, 83, 105].

To our knowledge, all the load-balancing systems that take server performance into account require TCP connection termination, enabling visibility into both request and response [6, 36, 24, 41, 13], or require application modification [51, 101]. TCP connection termination is CPU-expensive and is not always possible (e.g., frontend layer-4 LBs). Application modification creates other challenges (§2.3).

2.5 Goals for Next-Generation LBs

We believe that providing high performance to support emerging applications requires designing *distributed feedback control* at LBs, with local measurement and adaptation of request-routing policies. Ideal LBs of the future must:

- incorporate network and server processing delays into request-routing decisions (§2.1);
- react to server performance variation quickly (100μs–1ms) and on an ongoing basis (§2.2);
- require only locally observable information, avoiding application modification and storage (§2.3);
- operate under direct server return, observing only one direction of traffic, going from client to server (§2.4);
- impose minimal CPU and memory overhead due to feedback control on the critical request path; and
- meet standard LB requirements such as connection-to-server affinity and minimizing connection-breaking due to churn in the set of LBs and servers [60, 87, 49].

3 Design

In this section, we present the design of an LB that measures and optimizes *end-to-end* response latencies of connections balanced by it. The response latency of a server is the time interval between a request and its response as measured at the requesting client. However, for LBs implementing direct server return (DSR, §2.4), LBs cannot observe responses returning to clients. In the rest of this section, we present a novel measurement technique to estimate response latency under DSR, and a simple control algorithm that reacts to measured response latency. Our measurement technique may also apply more generally to passive round-trip time measurements with asymmetric routing [46].

Measuring proxy intervals using causally-triggered transmissions. Even if an LB does not observe a response packet, our key insight is that the LB could observe a packet *causally triggered by the response*. Hence, this triggered packet may be used to measure response latency, assuming that the latter lands at the LB "soon" after the response arrived at the client. The response latency is estimated as the delay between the request and the causally-triggered packet, both observed at the LB. The idea is illustrated in Fig.1(a). The proxy measurement is purely local to the LB, and can occur without client, server, application, or network coordination.

The proxy measurement will indeed be inaccurate relative to the response latency. Fig.1(b) illustrates the errors that are possible. T_{client} is the true response latency, but the proxy measurement T_{LB} differs from it in the following way: T_{LB} −

$T_{client} = T_{trigger} + O_3 - O_1$. Here, O_1 is the one-way delay for the first request from the client to the LB, O_2 is the delay for the request from the LB to reach the server and its response to reach the client, O_3 is the one-way delay for the causally-triggered packet from the client to the LB, and $T_{trigger}$ is the time it takes to trigger the packet after the response arrives. In our experience, O_1 and O_3 are statistically comparable, and $T_{trigger}$ is the bulk of the error in T_{LB}.

A simple instantiation of the proxy measurement idea is the estimation of the TCP round-trip time at the beginning of the connection by measuring the time interval between the SYN and the ACK packet of the TCP 3-way handshake [102, 46]. However, triggered packets are much more common and general beyond the TCP handshake. Other examples include: all TCP acknowledgments driven by packet receptions, including all ACK-clocked data transmissions; response-triggered dispatch of new requests due to flow control and concurrency limits in HTTP/2, QUIC, and RPC libraries [4, 15, 17]; and request-reply transactions serialized to respect data dependencies and ordering requirements in microservices [63, 43]. In general, any client-server pair that is prevented from transmitting data due to flow control (at the application or transport layer) will result in causally-triggered transmissions.

Unfortunately, identifying packets that are triggered due to responses of earlier requests is challenging. Consider Fig.1(c). There are several packets that an LB could consider as candidates for measurement. Without invoking detailed application or protocol knowledge (§2.3), it is unclear which of the packets is the one causally triggered by a response to a previous request.

Using inter-packet gaps to identify causally-triggered transmissions. Our observation is that in flow-controlled flows, some of the time gaps between successive packets are much longer than others. This is because a client will typically max out its quota of outstanding requests (determined by flow control), and wait for a reply before it is allowed to send subsequent packets. The wait produces the longer pause between transmissions: longer, typically, than the pauses between packet transmissions allowable by flow control, e.g., the window in case of TCP. A response breaks the pause in transmissions by re-opening the flow control quota.

Identifying triggered transmissions through pauses is reminiscent of *flowlet switching,* i.e., load-balancing bursts of packets in a TCP connection that are close together in time, an idea that has been harnessed for in-network load balancing [100, 103]. Flowlet switching uses a parameter, the *flowlet timeout*, which corresponds to the minimum idle time between flowlets. If the time gap between two successive packets in a connection exceeds this timeout, the second packet is said to belong to a new flowlet.

To identify triggered transmissions, one could attempt something similar, separating packets into batches based on a threshold on the inter-packet gap. The time gap between the first packets of successive batches provides a running estimate of the response latency of the connection. The algorithm

Algorithm 1: FIXEDTIMEOUT: Track causally-triggered transmissions through a fixed timeout to identify new batches of packets, executed at LB upon receiving each packet of flow f.

Input: Fixed inter-batch timeout, T
Input: Timestamp of the current packet's arrival, *now*
Input: The last time a new batch arrived for flow f, $f.time\_last\_batch$
Input: The last time a packet arrived for flow f, $f.time\_last\_pkt$
Output: An estimate of flow f's round trip time, $\hat{R}$, if a new sample is produced, else *undef*

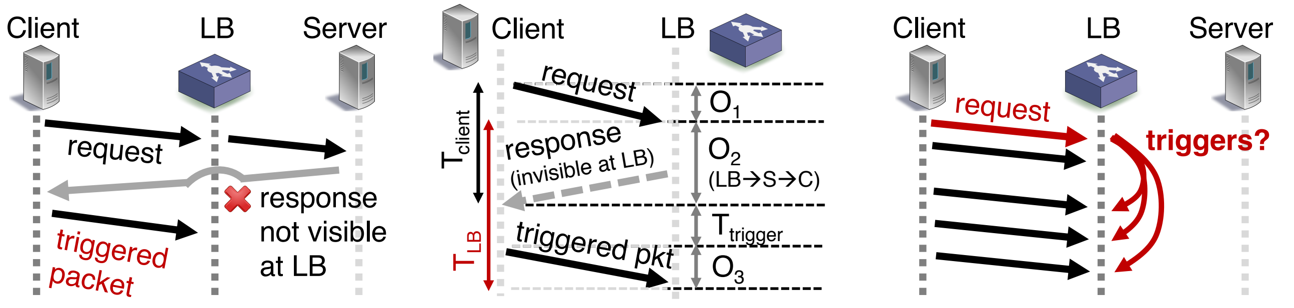
$\hat{R} = undef$
if $now - f.time\_last\_pkt > T$ **then**
  ▷ New batch: record response latency.
  $\hat{R} = now - f.time\_last\_batch$
  $f.time\_last\_batch = now$
end
$f.time\_last\_pkt = now$
return $\hat{R}$

FIXEDTIMEOUT shown in Algorithm 1 implements this approach; it must be executed upon the arrival of each packet belonging to flow f at an LB. The algorithm separates packets into batches and estimates response latency for flow f.

Unfortunately, setting the timeout parameter is nontrivial. Packets within a single batch of transmissions need not be transmitted back-to-back. Too low a timeout will incorrectly separate packets with small gaps into separate batches, and report artificially low response latencies. If the timeout is set too high, the algorithm will miss batches of packets, spanning multiple (true) packet batches, and inferring an erroneously high response latency.

The ideal timeout value that separates packets into batches depends on several factors that span the characteristics of both the workload and the underlying network. These factors include the propagation delay between the client and the server, the pattern of packet transmissions at the client (i.e., how flow control is implemented by the server and client), and the utilization contributed by the flow to the bottleneck link along the client-to-LB network path (higher the utilization, smaller the inter-packet time gap that separates batches). These factors change with the deployment and over time, and as such, it is challenging to determine a standard value applicable under all scenarios.

Using ensemble estimation and sample cliffs to demarcate triggered transmissions. We show that it is possible to take advantage of the specific kinds of errors contributed by incorrect timeouts *over time*, to triangulate to a timeout that works. Specifically, over a fixed epoch of time E (we use $E = 64$ ms), the number of samples obtained by FIXEDTIMEOUT (i.e., samples where $\hat{R}$ is not *undef*) for any timeout T, provides crucial information.



(a) Proxy measurement. (b) Errors in measurement. (c) Identifying triggered packets is hard.

Figure 1: Causally-triggered transmissions (§3): (a) It is possible to estimate the request ⇔ response latency at the client through a measurement of the request ⇔ triggered-packet latency at the LB. Measuring the latter only requires observing traffic going from client to server. (b) However, the proxy measurement T_{LB} may have errors relative to the desired measurement T_{client} (c) Identifying the packet triggered by the response of a given request is challenging.

Algorithm 2: ENSEMBLETIMEOUT: Track causally-triggered transmissions through an ensemble of timeouts and detection of a sample cliff. The algorithm is executed at the LB upon receiving each packet.

Input: k exponentially increasing timeouts $T_1, T_2, \cdots, T_k$
Input: Timestamp of the current packet's arrival, *now*
Input: The last time a new batch arrived for flow f, $f.time\_last\_batch_i$, one value maintained for each timeout T_i
Input: The last time a packet arrived for flow f, $f.time\_last\_pkt$
Input: Number of samples so far corresponding to T_i this epoch, N_i
Input: Epoch length, E
Input: Timeout chosen for current epoch, T_e
Output: An estimate of flow f's round trip time, $\hat{R}$
Output: A new timeout for the next epoch, T_e

```
1  for i ← 1 to k do
       ▷ For each timeout value
2      R̂_i = FIXEDTIMEOUT () with timeout T_i
3      if R̂_i not undef then
4          Increment sample count N_i for timeout T_i
5      end
6  end
7  if current packet is the first of a new epoch then
       ▷ Detect sample cliff
8      Pick m = argmax_i(N_i/N_{i+1})
       ▷ Reset all sample counters for next epoch
9      Set N_i ← 0 for all i
       ▷ For next epoch, use timeout T_m
10     T_e ← T_m
11 end
12 return R̂_e, T_e
```

Suppose the client transmits W packets on average within each round-trip time in the epoch. Suppose the true response latency is fixed at R over the duration of the epoch. If the timeout T were in fact close to the (unknown) ideal timeout T_{opt}, the number of samples obtained by FIXEDTIMEOUT will be equal to the number of true round-trip times within the epoch, i.e., E/R. However, if $T < T_{opt}$, FIXEDTIMEOUT will surely identify each round-trip time as a new batch, but it may also add additional erroneous samples of $\hat{R}$ (incorrectly assuming that some packets are from different batches). Specifically, FIXEDTIMEOUT may produce 2 to W times more samples of $\hat{R}$ than E/R, since one "true" batch of packets may be identified as anything between 2 to W distinct batches. On the other extreme, if $T > T_{opt}$, each sample $\hat{R}$ will span several round-trip times. The algorithm will produce far fewer than E/R samples.

Our key insight is to look for a drastic reduction in the number of samples collected with increasing timeouts T_i over an epoch, to help set the correct timeout for the next epoch. We call this *sample cliff* detection. Over each epoch E, algorithm ENSEMBLETIMEOUT (Algorithm 2) implements k instances of FIXEDTIMEOUT with timeout values $T_1, T_2, \cdots, T_k$ (lines 1–6). The timeouts T_i could be exponentially spaced to span a sufficiently large range of T_{opt} values. We use $T_1 = 64\mu s, T_2 = 128\mu s, \cdots, T_7 = 4ms$. At the end of each epoch, ENSEMBLETIMEOUT determines the largest reduction in the number of samples between adjacent timeouts (sorted from smallest to largest timeouts, see line 8). We pick a timeout corresponding to a sample cliff; suppose this timeout is T_m. ENSEMBLETIMEOUT returns response latencies estimated using T_m over the next epoch.

Simple load balancing strategy. Inspired by gradient-based methods used in traffic engineering [61, 77], we use a simple load-balancing strategy that redistributes a fixed fraction δ of total traffic from the server with the highest latency (as measured by ENSEMBLETIMEOUT) equally over all other servers. We use $\delta = 10\%$. The traffic shift may occur every time the LB receives a new sample of response latency, e.g., every round-trip time of each connection. We leave more sophisticated strategies to future work.

4 Preliminary Evaluation

This section provides a preliminary demonstration of how response latencies measured locally at LBs can aid in designing reactive load-balancing strategies. We implemented the measurement and control strategies described in §3 in the context of Cilium's XDP load balancer [55], which implements the Maglev hash function [60] to map connections to servers. In our setup, the LB balances requests arriving towards two memcached Kubernetes pods, each running on its own baremetal server on CloudLab [59].

The requests are generated using the memtier benchmark tool [33]. The client establishes multiple TCP connections, sends several requests over each connection, closes, and reopens the connections, and repeats over the duration of the experiment. Sending multiple requests over each connection allows the LB to observe response latencies per server. Reestablishing connections from time to time allows the LB to make fresh request-routing decisions using the learned server latencies. We used a 50-50 mix of GET and SET requests.

The LB is initialized with the default Maglev hash function, i.e., 50% of the slots in the LB's hash table point to each of the pods. However, in the middle of the experiment, we injected an artificial delay of 1 ms along the path from the LB to one of the servers. Fig.2 compares the 95th percentile GET response latency of the latency-aware design (§3) and the regular Maglev LB. The latency-aware design can react much faster: our instrumentation of the LB's hash table shows that the updates incorporate the latency inflation in milliseconds (the client only provides performance statistics every few seconds).

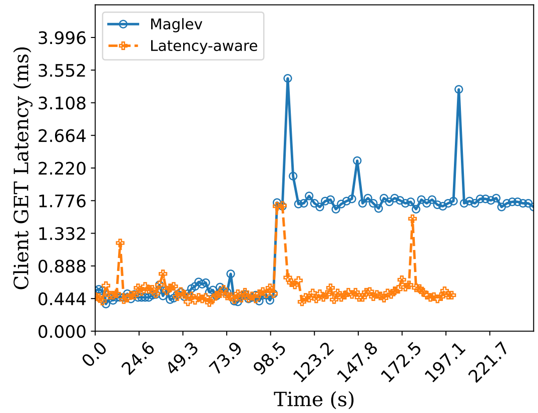


Figure 2: Evolution of the 95th percentile latency for GET requests in a load-balanced two-node memcached cluster. An artificial delay of 1 ms is injected at one of the servers, resulting in high tail latencies for a regular Maglev LB. However, a latency-aware approach (§3) can shift traffic to reduce tail latencies in milliseconds.

5 Open Research Questions

We outline open research questions pertaining to better in-band measurement and control at LBs.

(1) Handling scenarios without causal triggering. Responses from servers trigger subsequent transmissions if the client-server flow is bottlenecked by flow control (at the application or transport layers). However, data transfers may have other bottlenecks. Examples include application-limited flows and algorithms inducing delayed packet transmission in the network stack, e.g., TCP delayed ACKs.

(2) Addressing smooth inter-packet gaps. Even if causally-triggered transmissions are occurring, waiting for "long" pauses between packets may be insufficient to detect such transmissions. Examples of such scenarios include (i) *paced* packet transmissions, where each inter-packet gap looks similar to the next one (by design); and (ii) flows transmitting at the full rate of the bottleneck link between client and LB, so the inter-packet gaps all equal the link's transmission delay.

(3) Dealing with non-equidistant clients. The LB's decisions require aggregating flow-level response latencies into server-level assessments of performance. Such aggregation can be challenging when clients are distributed geographically, e.g., for frontend LBs. Here, the flow's response latency depends on the network paths from the client to the LB and the server to the client, which are outside the LB's control and impervious to its decisions. It is necessary to tease out just the components of the flow latency that can, in fact, be controlled through load balancing.

(4) Disambiguating poor performance due to load from other causes. Server load is only loosely correlated with high processing latency. Even at high load, an application's queue may be nearly empty [64], leading to low latency. Further, poor performance may manifest even at low loads due to "fail-slow" hardware faults occurring at scale [67, 106, 69, 68, 31]. Redirecting requests away from a a server presenting high response latencies may actually *worsen* tail latencies if other servers are overloaded in the process. LBs must learn to categorize the cause of poor server performance.

(5) Designing more sophisticated control loops. There are important open questions in designing control loops that optimize tail latency, while converging fast, yet avoiding thundering-herd problems when multiple LBs are reacting.

6 Conclusion

Load balancers (LBs) are critical components of interactive applications. In this paper, we have argued for in-band feedback control at LBs, and shown techniques to measure and react to server response latencies. We call on the community to research designs for novel performance-aware LBs.

7 References

[1] AWS S3 vs Google Cloud vs Azure: Cloud Storage Performance. [Online, Retrieved Jun 12, 2022.] http://blog.zachbjornson.com/2015/12/29/cloud-storage-performance.html, 2011.

[2] Send Hints to Dynamic Snitch when Compaction or repair is going on for a node. [Online, Retrieved Jun 12, 2022.] https://issues.apache.org/jira/browse/CASSANDRA-3722, 2012.

[3] Adopting Microservices at Netflix: Lessons for Architectural Design. [Online, Retrieved Jun 12, 2022.] https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/, 2015.

[4] RFC 7540 HTTP/2: Streams and Multiplexing. [Online, Retrieved Jun 12, 2022.] https://www.rfc-editor.org/rfc/rfc7540.html#section-5, 2015.

[5] Autoscaling in Kubernetes. [Online, Retrieved Jun 12, 2022.] https://kubernetes.io/blog/2016/07/autoscaling-in-kubernetes/, 2016.

[6] gRPC load balancing. [Online, Retrieved Jun 12, 2022.] https://grpc.io/blog/grpc-load-balancing/, 2017.

[7] What's a service mesh and why do I need one? [Online, Retrieved Jun 12, 2022.] https://linkerd.io/2017/04/25/whats-a-service-mesh-and-why-do-i-need-one/, 2017.

[8] Introduction to HAProxy ACLs. [Online, Retrieved Jun 12, 2022.] https://www.haproxy.com/blog/introduction-to-haproxy-acls/, 2018.

[9] Open-sourcing Katran, a scalable load balancer. [Online, Retrieved Jun 12, 2022.] https://engineering.fb.com/2018/05/22/open-source/open-sourcing-katran-a-scalable-network-load-balancer/, 2018.

[10] Cilium: Socket-based load balancing. [Online, Retrieved Jun 12, 2022.] https://cilium.io/blog/2019/08/20/cilium-16#hostservices, 2019.

[11] Deploying load balancing. [Online, Retrieved Jun 12, 2022.] https://docs.microsoft.com/en-us/windows/win32/rpc/deploying-load-balancing, 2019.

[12] Managing Uber's data workflows at scale. [Online, Retrieved Jun 12, 2022.] https://eng.uber.com/managing-data-workflows-at-scale/, 2019.

[13] Microsoft RPC load balancing. [Online, Retrieved Jun 12, 2022.] https://docs.microsoft.com/en-us/windows/win32/rpc/rpc-load-balancing, 2019.

[14] Rebuilding Twitter's public API. [Online, Retrieved Jun 12, 2022.] https://blog.twitter.com/engineering/en_us/topics/infrastructure/2020/rebuild_twitter_public_api_2020, 2020.

[15] RFC 9000: QUIC: flow control. [Online, Retrieved Jun 12, 2022.] https://www.rfc-editor.org/rfc/rfc9000.html#flow-control, 2020.

[16] Comparing Kubernetes Container Network Interface (CNI) providers. [Online, Retrieved Jun 12, 2022.] https://kubevious.io/blog/post/comparing-kubernetes-container-network-interface-cni-providers, 2021.

[17] gRPC performance best practices. [Online, Retrieved Jun 12, 2022.] https://grpc.io/docs/guides/performance/, 2021.

[18] How eBPF will solve Service Mesh - Goodbye Sidecars. [Online, Retrieved Jun 12, 2022.] https://isovalent.com/blog/post/2021-12-08-ebpf-servicemesh/, 2021.

[19] The Cost of Cloud, a Trillion Dollar Paradox. [Online, Retrieved Jun 12, 2022.] https://a16z.com/2021/05/27/cost-of-cloud-paradox-market-cap-cloud-lifecycle-scale-growth-repatriation-optimization/, 2021.

[20] The Human Side of Airbnb's Microservice Architecture. [Online, Retrieved Jun 12, 2022.] https://www.infoq.com/presentations/airbnb-culture-soa/, 2021.

[21] Automatic mTLS. [Online, Retrieved Jun 12, 2022.] https://linkerd.io/2.11/features/automatic-mtls/, 2022.

[22] AWS Lambda. [Online, Retrieved Jun 12, 2022.] https://aws.amazon.com/lambda/, 2022.

[23] Configure disks to meet performance requirements. [Online, Retrieved Jun 12, 2022.] https://cloud.google.com/compute/docs/disks/performance, 2022.

[24] Envoy: supported load balancers. [Online, Retrieved Jun 12, 2022.] https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/upstream/load_balancing/load_balancers, 2022.

[25] Google cloud: Load balancing mode. [Online, Retrieved Jun 12, 2022.] https://cloud.google.com/load-balancing/docs/backend-service#balancing-mode, 2022.

[26] Google cloud: Traffic policies. [Online, Retrieved Jun 12, 2022.] https://cloud.google.com/load-balancing/docs/l7-internal/traffic-management#traffic_policies, 2022.

[27] Istio: Locality load balancing. [Online, Retrieved Jun 12, 2022.] https://istio.io/latest/docs/tasks/traffic-management/locality-load-balancing/, 2022.

[28] Kubernetes Networking: Load Balancer and Network Load Balancer. [Online, Retrieved Jun 12, 2022.] https://ibm.github.io/kubernetes-networking/services/loadbalancer/, 2022.

[29] Kubernetes scheduler. [Online, Retrieved Jun 12, 2022.] https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/, 2022.

[30] Kubernetes without Kube-Proxy. [Online, Retrieved Jun 12, 2022.] https://docs.cilium.io/en/stable/gettingstarted/kubeproxy-free/, 2022.

[31] Latency Sensei Gallery. [Online, Retrieved Jun 12, 2022.] https://sensei.clockwork.io/user/gallery/, 2022.

[32] Linux container and virtualization tools. [Online, Retrieved Jun 12, 2022.] https://linuxcontainers.org/, 2022.

[33] memtier_benchmark. [Online, Retrieved Jun 12, 2022.] https://github.com/RedisLabs/memtier_benchmark/, 2022.

[34] Microservices and Microservices architecture. [Online, Retrieved Jun 12, 2022.] https://www.intel.com/content/www/us/en/cloud-computing/microservices.html, 2022.

[35] Next-generation mutual authentication with Cilium service mesh. [Online, Retrieved Jun 12, 2022.] https://isovalent.com/blog/post/2022-05-03-servicemesh-security/, 2022.

[36] NGINX Plus Feature: Load Balancing. [Online, Retrieved Jun 12, 2022.] https://www.nginx.com/products/nginx/load-balancing, 2022.

[37] Serverless on AWS. [Online, Retrieved Jun 12, 2022.] https://aws.amazon.com/serverless/, 2022.

[38] The Kubernetes network model. [Online, Retrieved Jun 12, 2022.] https://kubernetes.io/docs/concepts/services-networking/, 2022.

[39] The Kubernetes Networking Guide: NodePort. [Online, Retrieved Jun 12, 2022.] https://www.tkng.io/services/nodeport/, 2022.

[40] Use containers to Build, Share and Run your applications. [Online, Retrieved Jun 12, 2022.] https://www.docker.com/resources/what-container, 2022.

[41] Using nginx as HTTP load balancer. [Online, Retrieved Jun 12, 2022.] https://nginx.org/en/docs/http/load_balancing.html, 2022.

[42] What is a container? [Online, Retrieved Jun 12, 2022.] https://azure.microsoft.com/en-us/overview/what-is-a-container/#overview, 2022.

[43] ZeroMQ: Advanced request-reply patterns. [Online, Retrieved Jun 12, 2022.]

https://zguide.zeromq.org/docs/chapter3/, 2022.

[44] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, A. Vahdat, et al. Hedera: dynamic flow scheduling for data center networks. In *Nsdi*, volume 10, pages 89–92. San Jose, USA, 2010.

[45] G. Antichi and G. Rétvári. Full-stack SDN: The next big challenge? In *Proceedings of the Symposium on SDN Research*, pages 48–54, 2020.

[46] M. Apostolaki, A. Singla, and L. Vanbever. *Performance-Driven Internet Path Selection*, page 41–53. Association for Computing Machinery, New York, NY, USA, 2021.

[47] J. T. Araújo, L. Saino, L. Buytenhek, and R. Landa. Balancing on the edge: Transport affinity without network state. In *Usenix Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.

[48] S. Ashok, P. B. Godfrey, and R. Mittal. Leveraging service meshes as a new network layer. In *Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks*, pages 229–236, 2021.

[49] T. Barbette, C. Tang, H. Yao, D. Kostić, G. Q. Maguire Jr, P. Papadimitratos, and M. Chiesa. A high-speed load-balancer design with guaranteed per-connection-consistency. In *Usenix Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

[50] P. Barford and M. Crovella. Critical path analysis of tcp transactions. In *ACM SIGCOMM*, 2000.

[51] Brandon Williams. Dynamic snitching in Cassandra: past, present, and future. [Online, Retrieved Jun 12, 2022.] https://www.datastax.com/blog/dynamic-snitching-cassandra-past-present-and-future, 2012.

[52] D. Bristot de Oliveira, D. Casini, R. Oliveira, and T. Cucinotta. Demystifying the real-time linux scheduling latency. In *ECRTS*, 07 2020.

[53] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes. Borg, Omega, and Kubernetes: Lessons learned from three container-management systems over a decade. *Queue*, 14(1):70–93, 2016.

[54] Carson Anderson. Kubernetes deconstructed. [Online, Retrieved Jun 12, 2022.] https://vimeo.com/245778144/4d1d597c5e, 2017.

[55] Daniel Borkmann. Kube-proxy replacement at the XDP layer. [Online, Retrieved Jun 12, 2022.] https://cilium.io/blog/2020/06/22/cilium-18#kubeproxy-removal, 2020.

[56] Daniel Borkmann. Cilium & BPF: a fundamentally better dataplane. [Online, Retrieved Jun 12, 2022.] https://guild42.ch/wp-content/uploads/2021/12/Guild42.ch-BPF-Borkmann.pdf, 2022.

[57] Daniel Borkmann and Martynas Pumputis. K8s Service Load Balancing with BPF & XDP. In *Linux Plumbers Conference*, 2020.

[58] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[59] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. hh0Cecchet, S. Kar, and P. Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2019.

[60] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein. Maglev: A fast and reliable software network load balancer. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, 2016.

[61] A. Elwalid, C. Jin, S. Low, and I. Widjaja. Mate: Mpls adaptive traffic engineering. In *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No. 01CH37213)*, volume 3, pages 1300–1309. IEEE, 2001.

[62] J. Fried, Z. Ruan, A. Ousterhout, and A. Belay. Caladan: Mitigating interference at microsecond timescales. In *Usenix Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[63] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18, 2019.

[64] Y. Gan, Y. Zhang, K. Hu, D. Cheng, Y. He, M. Pancholi, and C. Delimitrou. Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices. In *Proceedings of the twenty-fourth international conference on architectural support for programming languages and operating systems*, pages 19–33, 2019.

[65] R. Gandhi, H. H. Liu, Y. C. Hu, G. Lu, J. Padhye, L. Yuan, and M. Zhang. Duet: Cloud scale load balancing with hardware and software. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, page 27–38, New York, NY, USA, 2014. Association for Computing Machinery.

[66] M. Ghasemi, T. Benson, and J. Rexford. Dapper: Data plane performance diagnosis of tcp. In *Proceedings of the Symposium on SDN Research (SOSR)*, 2017.

[67] H. S. Gunawi, R. O. Suminto, R. Sears, C. Golliher, S. Sundararaman, X. Lin, T. Emami, W. Sheng, N. Bidokhti, C. McCaffrey, D. Srinivasan, B. Panda, A. Baptist, G. Grider, P. M. Fields, K. Harms, R. B. Ross, A. Jacobson, R. Ricci, K. Webb, P. Alvaro, H. B. Runesha, M. Hao, and H. Li. Fail-slow at scale: Evidence of hardware performance faults in large production systems. *ACM Trans. Storage*, 14, 2018.

[68] P. Helland. Fail-fast Is Failing... Fast! Changes in compute environments are placing pressure on tried-and-true distributed-systems solutions. *Queue*, 19(1):5–15, 2021.

[69] P. H. Hochschild, P. J. Turner, J. C. Mogul, R. K. Govindaraju, P. Ranganathan, D. E. Culler, and A. Vahdat. Cores that don't count. In *Proc. 18th Workshop on Hot Topics in Operating Systems (HotOS 2021)*, 2021.

[70] S. Ibanez, A. Mallery, S. Arslan, T. Jepsen, M. Shahbaz, C. Kim, and N. McKeown. The nanopu: A nanosecond network stack for datacenters. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 239–256. USENIX Association, July 2021.

[71] V. Jacobson and M. J. Karels. Congestion avoidance and control. In *SIGCOMM 1988*, Stanford, CA, Aug. 1988.

[72] S. Jaiswal, G. Iannaccone, C. Diot, J. Kurose, and D. Towsley. Inferring tcp connection characteristics through passive measurements. In *IEEE INFOCOM 2004*, volume 3, pages 1582–1592 vol.3, 2004.

[73] James Lewis and Martin Fowler. Microservices: a definition of this new architectural term. [Online, Retrieved Jun 12, 2022.] https://martinfowler.com/articles/microservices.html, 2014.

[74] H. Jiang and C. Dovrolis. Passive estimation of tcp round-trip times. *SIGCOMM Comput. Commun. Rev.*, 32:75–88, 2002.

[75] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis. Shinjuku: Preemptive scheduling for {μsecond-scale} tail latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 345–360, 2019.

[76] J. Kaldor, J. Mace, M. Bejda, E. Gao, W. Kuropatwa, J. O'Neill, K. W. Ong, B. Schaller, P. Shan, B. Viscomi, et al. Canopy: An end-to-end performance tracing and analysis system. In *Proceedings of the 26th symposium on operating systems principles*, pages 34–50, 2017.

[77] S. Kandula, D. Katabi, B. Davie, and A. Charny. Walking the tightrope: Responsive yet stable traffic engineering. *ACM SIGCOMM Computer Communication Review*, 2005.

[78] D. Katabi, M. Handley, and C. Rohrs. Congestion control for high bandwidth-delay product networks. In *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 89–102, 2002.

[79] C. Lee and J. Ousterhout. Granular computing. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 149–154, 2019.

[80] J. Li, N. K. Sharma, D. R. K. Ports, and S. D. Gribble. Tales of the tail: Hardware, os, and application-level sources of tail latency. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC

'14, page 1–14, New York, NY, USA, 2014. Association for Computing Machinery.

[81] Y. Li, S. J. Park, and J. Ousterhout. MilliSort and MilliQuery: Large-Scale Data-Intensive computing in milliseconds. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 593–611. USENIX Association, Apr. 2021.

[82] Z. Liu, S. Zhou, O. Rottenstreich, V. Braverman, and J. Rexford. *Memory-Efficient Performance Monitoring on Programmable Switches with Lean Algorithms*, pages 31–44.

[83] G. Lu and X. Li. On the correspondency between tcp acknowledgment packet and data packet. In *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement (IMC)*, 2003.

[84] S. McClure, A. Ousterhout, S. Shenker, and S. Ratnasamy. Efficient scheduling policies for Microsecond-Scale tasks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1–18, Renton, WA, Apr. 2022. USENIX Association.

[85] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 15–28, 2017.

[86] R. Mittal, V. T. Lam, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, and D. Zats. Timely: Rtt-based congestion control for the datacenter. *ACM SIGCOMM Computer Communication Review*, 45(4):537–550, 2015.

[87] V. Olteanu, A. Agache, A. Voinescu, and C. Raiciu. Stateless datacenter load-balancing with beamer. In *Usenix Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.

[88] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun. Making sense of performance in data analytics frameworks. In *Usenix Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[89] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 69–84, New York, NY, USA, 2013. Association for Computing Machinery.

[90] J. Pahdye and S. Floyd. On inferring tcp behavior. In *ACM SIGCOMM*, 2001.

[91] A. Parker, D. Spoonhower, J. Mace, B. Sigelman, and R. Isaacs. *Distributed tracing in practice: Instrumenting, analyzing, and debugging microservices*. O'Reilly Media, 2020.

[92] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, C. Kim, and N. Karri. Ananta: Cloud scale load balancing. *SIGCOMM Comput. Commun. Rev.*, 43:207–218, 2013.

[93] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. Fastpass: A centralized" zero-queue" datacenter network. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 307–318, 2014.

[94] G. Prekas, M. Kogias, and E. Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 325–341, New York, NY, USA, 2017. Association for Computing Machinery.

[95] H. Sadok, Z. Zhao, V. Choung, N. Atre, D. S. Berger, J. C. Hoe, A. Panda, and J. Sherry. We need kernel interposition over the network dataplane. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 152–158, 2021.

[96] Satadal Sengupta and Hyojoon Kim and Jennifer Rexford. Continuous In-Network Round-Trip Time Monitoring. In *SIGCOMM 2022*, Aug. 2022.

[97] N. Serrino. Horizontal Pod Autoscaling with Custom Metrics in Kubernetes. [Online, Retrieved Jun 12, 2022.] `https://blog.px.dev/autoscaling-custom-k8s-metric/`, 2021.

[98] N. V. Shirokov. XDP: 1.5 years in production. Evolution and lessons learned. In *Linux Plumbers Conference*, 2018.

[99] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. 2010.

[100] S. Sinha, S. Kandula, and D. Katabi. Harnessing tcp's burstiness with flowlet switching. In *Proc. 3rd ACM Workshop on Hot Topics in Networks (Hotnets-III)*, 2004.

[101] L. Suresh, M. Canini, S. Schmid, and A. Feldmann. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *Usenix Symposium on Networked Systems Design and Implementation (NSDI)*, pages 513–527, 2015.

[102] M. Szymaniak, D. Presotto, G. Pierre, and M. van Steen. Practical large-scale latency estimation. *Computer Networks*, 52(7):1343–1364, 2008.

[103] E. Vanini, R. Pan, M. Alizadeh, P. Taheri, and T. Edsall. Let it flow: Resilient asymmetric load balancing with flowlet switching. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 407–420, 2017.

[104] B. Veal, K. Li, and D. Lowenthal. New methods for passive estimation of tcp round-trip times. In *International workshop on passive and active network measurement*, pages 121–134. Springer, 2005.

[105] W. Wu, G. Wang, A. Akella, and A. Shaikh. Virtual network diagnosis as a service. In *Proceedings of the 4th annual Symposium on Cloud Computing*, pages 1–15, 2013.

[106] A. Yoo, Y. Wang, R. Sinha, S. Mu, and T. Xu. Fail-slow fault tolerance needs programming support. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, page 228–235, New York, NY, USA, 2021. Association for Computing Machinery.

[107] C. Zeng, L. Luo, T. Zhang, Z. Wang, L. Li, W. Han, N. Chen, L. Wan, L. Liu, Z. Ding, X. Geng, T. Feng, F. Ning, K. Chen, and C. Guo. Tiara: A scalable and efficient hardware acceleration architecture for stateful layer-4 load balancing. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1345–1358, Renton, WA, Apr. 2022. USENIX Association.

[108] Y. Zhang, L. Breslau, V. Paxson, and S. Shenker. On the characteristics and origins of internet flow rates. In *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 309–322, 2002.

[109] H. Zhu, K. Kaffes, Z. Chen, Z. Liu, C. Kozyrakis, I. Stoica, and X. Jin. RackSched: A Microsecond-Scale scheduler for Rack-Scale computers. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1225–1240. USENIX Association, Nov. 2020.

[110] D. Zhuo, K. Zhang, Y. Zhu, H. H. Liu, M. Rockett, A. Krishnamurthy, and T. Anderson. Slim: OS kernel support for a low-overhead container overlay network. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 331–344, 2019.