Load Balancers Need In-Band Feedback Control

Bhavana Vannarth Shobhana, Srinivas Narayana, and Badri Nath
Rutgers University, USA

ABSTRACT

Server load balancers (LBs) are critical components of interactive services, routing client requests to servers in a pool. LBs improve service performance and increase availability by spreading the request load evenly across servers.

It is time to rethink what LBs can do for applications. As application compute becomes increasingly granular (e.g., microservices), request-processing latencies at servers will be ever more impacted by software and system variability at small time scales (e.g., 100μs–1ms). Beyond balancing load, we argue that LBs must actively optimize application response time, by adapting request-routing to quickly-varying server performance.

Specifically, we advocate for *in-band feedback control:* LBs should adapt the request-routing policy using purely local observations of server performance, derived from requests traversing the LB. A key challenge to designing such feedback controllers is that high-speed LBs only see the requests, not the responses. We present the design of an LB that adapts to a server latency inflation of 1 ms and reduces tail latencies in milliseconds, while observing only client-to-server traffic.

CCS CONCEPTS

• **Networks** → **Middle boxes**; **Network measurement**;

KEYWORDS

Load balancers, feedback control, passive measurement

ACM Reference Format:
Bhavana Vannarth Shobhana, Srinivas Narayana, and Badri Nath *Rutgers University, USA*. 2022. Load Balancers Need In-Band Feedback Control. In *The 21st ACM Workshop on Hot Topics in Networks (HotNets '22), November 14–15, 2022, Austin, TX, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3563766.3564094

1 INTRODUCTION

Server 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 [10, 49, 51, 62, 89, 94]. 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 [7, 11, 12, 14, 26, 30, 40, 43, 56, 57]. An LB may use either a request's layer-4 (connection 4-tuple) or layer-7 identifiers (e.g., HTTP object path) to route the request to a server. Typical request-routing policies aim to balance the request load evenly among servers in the pool [49, 62, 89].

Emerging trends in how interactive services are designed require us to rethink the role of LBs in applications. With the advent of microservices, serverless, and rack-scale computing [25, 35, 38, 69, 72, 74, 80, 83, 86, 109], application compute tasks are becoming 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 [6], and request duplication [60] will not work at these time scales. LBs, however, are in a unique position to mitigate high server variability: instead of simply balancing load, LBs may adapt request-routing to actively optimize service performance.

Adapting request-routing requires the design of feedback controllers that observe and react quickly to changes in server performance. However, shipping performance data from applications to centralized controllers or even the LBs themselves presents significant challenges in application instrumentation, data collection, and data freshness (§2.3).

We argue that each LB must implement *in-band feedback control,* reacting to the performance of remote servers using purely local observations derived from server traffic traversing the LB. Such an approach can improve application performance even without co-opting servers, clients, applications, or the network. We take inspiration from the long history of feedback control in our community, e.g., for TCP congestion [70, 77, 87] and wide-area traffic engineering [63, 76].

However, measuring 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 cut down CPU consumption and reduce response latency [15, 94].

HotNets '22, November 14–15, 2022, Austin, TX, USA

ACM ISBN 978-1-4503-9899-2/22/11…$15.00
https://doi.org/10.1145/3563766.3564094

This paper takes a first step towards in-band feedback control at LBs by presenting a technique to measure end-to-end response latency without observing responses (§3). End-to-end latency is a good indicator of a server's request-processing delay when a client is "close" to the LB, for example in tier-to-tier LBs and CDN/edge clusters. 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 end-to-end latencies. We present a simple controller that adapts request-routing using these 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 WHY IN-BAND FEEDBACK CONTROL?

2.1 Granularity and Network Delays

Modern user-facing services break complex application logic into loosely-coupled components, termed microservices [35, 72], 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 [4, 13, 16, 23], with the slowest microservice dominating response time [60]. 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 [25, 38], rack-scale [69, 74, 80, 83, 86, 109], 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 [69, 91]. 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 ignore the effects of network paths except at coarse spatial granularities [28, 29].

Further, the rate of load-balancing decisions increases with finer compute granularity. Hence, it is critical to get server selection "right" for each request, to support high 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 [33, 42, 44, 55] 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* [18]) and the service mesh [8, 47, 50]. These additional layers support translation between container and provider network addresses [39], access control policies [9], and authentication between containers [24, 36]. Each network message between containers may traverse the software network stack twice as many times as packets between baremetal machines [20, 110].

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 [54, 58, 74, 82]). Increasing the time spent by messages in the network stack also amplifies the impact of background tasks such as compaction and garbage collection [2, 60, 90] on processing latency. Recent works that improve operating system scheduling to shrink tail latencies [64, 74, 86, 96] use user-space networking stacks, which coexist poorly with multi-tenancy [97]. As such, they cannot support deployment in shared clusters.

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 [60]. 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 [22]. Automatic scaling [55] to spin up new VMs and containers may take tens of seconds to take effect [6, 31]. Compared to sending the request to a fast server in the first place, timeout-based request duplication [60] 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. Beyond merely balancing connections across servers [10, 62, 94] as many LBs aim to do, LBs 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 for all the requests arriving within this duration. However, to adapt to changing server performance, LBs must first observe it—a challenging task that we discuss below.

2.3 Avoiding App Modification

If servers could supply LBs with signals of local application performance out-of-band, perhaps LBs could use those signals to adapt how they route requests to the servers. For example, applications may publish the occupancy of software queues or CPU and memory utilization to external monitoring systems, or even directly to LBs [1, 3, 27, 28, 66, 79, 103]. Alternatively, centralized controllers [46, 95] may consume

such information from servers and perform control actions to update request-routing 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 [101]. Any degree of heterogeneity compounds the challenges of instrumenting source code [75, 93, 99]. 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 deployment.

If performance signals could indeed be collected from servers and applications, the efficacy of adaptive request-routing would depend on how quickly LBs can access fresh performance data or updated control actions. Designing a pub-sub system or implementing fast RPCs to propagate signals from large numbers of servers to LBs before the signals get stale (§2.2) will entail significant complexity and cost.

2.4 Minimizing Traffic Footprint

To avoid the staleness and complexity of out-of-band signaling, it is appealing to ask whether LBs can measure server performance *in-band* using data traffic traversing the LBs.

Unfortunately, this is not easy to do. Strictly speaking, LBs are just "infrastructure", moving data to and from application components. Yet, they must be designed to scale to large request loads and avoid additional latency on the critical request-processing path. Taming the CPU utilization of software LBs is a significant operational concern, both for frontend and tier-to-tier LBs [15, 59, 67, 94, 100]. It is especially critical for frontend LBs since they handle every packet sent to a service, including volumetric DDoS attacks.

Specifically, many LBs implement *direct server return (DSR)*, an optimization that enables servers to send response traffic directly to clients bypassing the LB [30, 32, 40, 94]. 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 request-processing delay or rate directly at the LB. The assumption of observing both directions of traffic is ubiquitous in measurement works that aim to passively measure round-trip times of connections from an intermediate vantage point [52, 68, 71, 73, 84, 85, 92, 98, 106–108].

Today, LBs exist that leverage server performance to adapt request-routing. They fall into two classes. The first requires terminating TCP connections on both sides, hence seeing both requests and responses [7, 14, 21, 26, 37, 41, 43, 79]. TCP connection termination is CPU- and memory-expensive, and often infeasible, e.g., frontend LBs. The second class uses out-of-band signaling [1, 3, 53, 78, 103], creating other challenges (§2.3). Neither approach is general or scalable.

2.5 Goals for Next-Generation LBs

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

- incorporate network and server processing delays into request-routing decisions (§2.1);
- react quickly to server performance variation (100μs–1ms) and on an ongoing basis (§2.2);
- use purely local observations, avoiding the need for application modification or external storage (§2.3);
- operate under direct server return, observing only one direction of traffic, going from client to server (§2.4);
- meet standard LB requirements such as connection-to-server affinity and minimize connection-breaking due to churn in the set of LBs and servers [51, 62, 89].

3 DESIGN

As a first step towards in-band feedback control at LBs, we present a design that optimizes *end-to-end* response latencies.

The end-to-end response latency is the sum of four components: (i) the delay for a request to travel from client to LB, (ii) then from LB to server, (iii) the delay for the server to process the request, and (iv) the delay for the response to travel from server to client (skipping LB). Ideally, an LB should measure and react just to the components that it can control with request-routing—the server-side delays (ii) and (iii). When clients are "close" to LBs, e.g., in tier-to-tier LBs and in CDN/edge clusters, the end-to-end response latency closely matches the controllable components of the delay.

In the rest of this section, we present a novel measurement technique to estimate the end-to-end response latency under direct server return (§2.4), and a simple control algorithm that adapts request-routing. Our measurement technique may also apply more generally to passive round-trip time measurements with asymmetric routing [48].

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, and the proxy measurement T_{LB} has the error $T_{LB} - T_{client} = O_3 - O_1 + T_{trigger}$. 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 for the client to trigger the next 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 [48, 81, 88, 104]. However, triggered packets are much more common and general. Other examples of triggered packets 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 [5, 17, 19]; and request-reply transactions serialized to respect data dependencies and ordering requirements in microservices [45, 65]. 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.

However, identifying packets 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 packet is 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 server response breaks the pause in transmissions by re-opening the flow control quota.

Separating packets into batches using pauses is reminiscent of *flowlet switching,* i.e., load-balancing batches of packets in a TCP connection that are close together in time, an idea that has been harnessed for in-network load balancing [102, 105]. 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 (batch).

One could identify triggered transmissions in a manner similar to identifying flowlets. The time gap between the first packets of successive batches provides a running estimate of the response latency of the connection, $\hat{T}_{LB}$. 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, δ
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, $\hat{T}_{LB}$, if a new sample is produced, else $undef$

```
1 T̂_LB = undef
2 if now − f.time_last_pkt > δ then
      ▷ New batch: record response latency.
3     T̂_LB = now − f.time_last_batch
4     f.time_last_batch = now
5 end
6 f.time_last_pkt = now
7 return T̂_LB
```

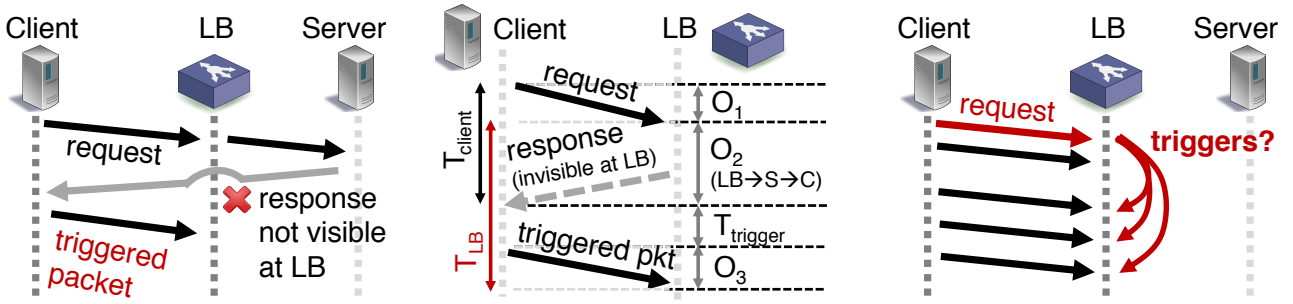
FIXEDTIMEOUT shown in Algorithm 1 implements this approach. It must be executed upon the arrival of each packet belonging to flow f at an LB. The algorithm separates packets into batches and estimates response latency for flow f.

However, setting the inter-batch timeout δ is nontrivial. Packets within a single batch 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. The timeout depends on the propagation delay between the client and the server, 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), and the pattern of packet transmissions at the client (i.e., how flow control is implemented by the server and client). These factors change with the deployment and over time, and as such, it is challenging to use a standard value in all scenarios.

Using ensemble estimation and sample cliffs. 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 $\hat{T}_{LB}$ is not $undef$) for any timeout δ, provides crucial information.

Suppose the true round-trip time (RTT) is fixed at T_{LB} over the duration of the epoch. If the timeout δ were in fact close to the (unknown) ideal timeout δ_{opt}, the number of samples obtained by FIXEDTIMEOUT will equal the number of true



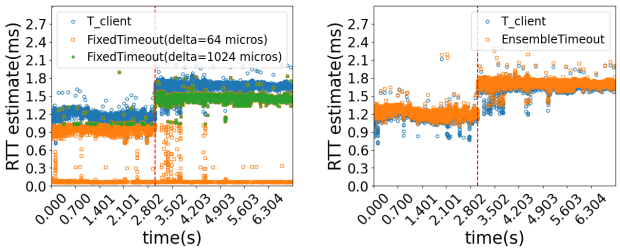
(a) Proxy measurement. (b) Errors in measurement. (c) Identifying triggered packets is hard.

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 $\delta_1, \delta_2, \cdots, \delta_k$
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 δ_i
Input: The last time a packet arrived for flow f, $f.time\_last\_pkt$
Input: Number of samples so far corresponding to δ_i this epoch, N_i
Input: Epoch length, E
Input: Timeout chosen for current epoch, δ_e
Output: An estimate of flow f's round trip time, $\hat{T}_{LB}$
Output: A new timeout for the next epoch, δ_e

for $i \leftarrow 1$ ***to*** k **do**
  ▷ For each timeout value
  $\hat{T}_{LB,i}$ = FixedTimeout () with timeout δ_i
  if $\hat{T}_{LB,i}$ *not undef* **then**
    Increment sample count N_i for timeout δ_i
  end
end
if *current packet is the first of a new epoch* **then**
  ▷ Detect sample cliff
  Pick $m = argmax_i(\frac{N_i}{N_{i+1}})$
  ▷ Reset all sample counters for next epoch
  Set $N_i \leftarrow 0$ for all i
  ▷ For next epoch, use timeout δ_m
  $\delta_e \leftarrow \delta_m$
end
return $\hat{T}_{LB,e}, \delta_e$



(a) (b)

Figure 2: Timeout-based RTT estimate $\hat{T}_{LB}$ compared against ground truth T_{client} (§3): (a) With FixedTimeout, using a low timeout δ produces too many low estimates. Too high a timeout results in too few large estimates. (b) EnsembleTimeout finds the best timeout δ_m using sample cliffs, tracking changes in the true RTT.

RTTs within the epoch, i.e., $\frac{E}{T_{LB}}$. If $\delta < \delta_{opt}$, FixedTimeout will still separate packets from different RTTs into different batches. However, FixedTimeout may also produce additional erroneous (low) outputs of $\hat{T}_{LB}$, incorrectly separating some packets from the same RTT into different batches. If $\delta > \delta_{opt}$, each output $\hat{T}_{LB}$ will span several true RTTs, and the algorithm will produce far fewer than $\frac{E}{T_{LB}}$ outputs.

Fig.2(a) compares the outputs from FixedTimeout ($\hat{T}_{LB}$) against the ground truth measured at the client (T_{client}), when observing a backlogged TCP flow between two endpoints at an LB. Throughout the experiment, an incorrect low timeout $\delta = 64\mu s$ produces many erroneously low $\hat{T}_{LB}$ outputs (see horizontal band near RTT $64\mu s$). The true RTT increases at $t = 3s$ (vertical dashed line). Before the increase, the timeout $\delta = 1024\mu s$ is too large. FixedTimeout produces a small number of erroneously large outputs.

Our key insight is to look for a drastic reduction in the number of samples collected with increasing timeouts δ_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 $\delta_1, \delta_2, \cdots, \delta_k$ (lines 1–6). The timeouts δ_i could be exponentially spaced to span a sufficiently large range of δ_{opt} values. We use $\delta_1 = 64\mu s, \delta_2 = 128\mu s, \cdots, \delta_7 = 4ms$. 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 δ_m. EnsembleTimeout returns response latencies estimated using δ_m over the next epoch. Fig.2(b) shows how EnsembleTimeout adapts its timeout δ_m dynamically to track the ground truth T_{client} closely in the same experiment where fixed timeouts δ produce erroneous outputs (Fig.2(a)).

Simple load balancing strategy. Inspired by gradient-based methods used in traffic engineering [63, 76], we use a simple load-balancing strategy that redistributes a fixed fraction α of total traffic from the server with the highest latency (as measured by EnsembleTimeout) equally over all other servers. We use $\alpha = 10\%$. 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 [57], which implements the Maglev hash function [62] 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 [61].

The requests are generated using the memtier benchmark tool [34]. 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 ($t = 100s$), we injected an artificial delay of 1 ms along the path from the LB to one of the servers. Fig.3 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 statistics every few seconds).

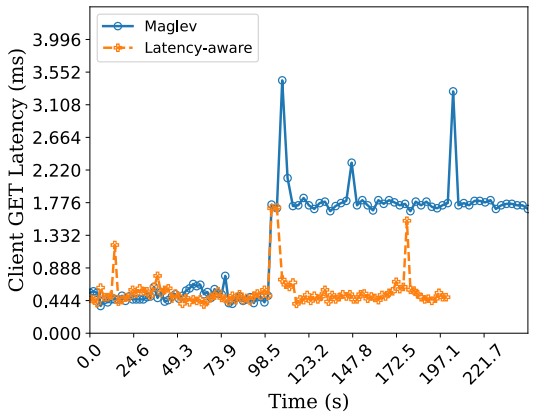


Figure 3: Evolution of the 95th percentile latency for GET requests in a load-balanced two-node memcached cluster. A delay of 1 ms is injected at one of the servers at $t = 100s$, increasing the tail latency for a regular Maglev LB. However, a latency-aware approach (§3) shifts traffic and reduces tail latencies in milliseconds.

5 OPEN RESEARCH QUESTIONS

(1) Dealing with far, non-equidistant clients. The LB's decisions do not control the client-to-LB path. Hence, the end-to-end round-trip time (RTT) of a client request is not always representative of the delays that an LB can control. Could an LB identify connections which can, in fact, see a performance benefit using performance-aware feedback control at the LB? How should an LB measure just the components of the RTT that are under the LB's control?

(2) Handling general packet timing behaviors. The techniques in this paper rely on clients sending bursts of packets and triggering subsequent packets "soon" upon responses. LBs must identify and handle violations of such timing assumptions: (1) application-limited clients, (2) network protocol behaviors with delayed transmission (e.g., TCP delayed ACKs), and (3) packet pacing.

(3) Handling application dependencies. How should an LB recognize that a server appears to be slow not because it is slow but one of its downstream dependencies is slow? How should an LB shift traffic if a dependency is slow?

(4) Designing more sophisticated control loops. Could we design control loops to minimize tail latency, while converging fast, without thundering-herd problems, with many LBs?

Conclusion. In this paper, we have argued that LBs must go beyond just balancing load, implementing in-band feedback control to actively improve application performance. We call upon the community to build on the techniques in this paper.

Acknowledgments. We thank the anonymous HotNets reviewers, Vig Sachidananda, Balaji Prabhakar, Sandip Shah, and Seungjoon Lee for helpful discussions on this paper. This work was funded in part by NSF grant CC* 1925482 and the Rutgers School of Arts and Sciences start-up fund.

REFERENCES

[1] 2006. Dynamic Feedback Load Balancing Scheduling. [Online, Retrieved Oct 14, 2022.] http://kb.linuxvirtualserver.org/wiki/Dynamic_Feedback_Load_Balancing_Scheduling. (2006).

[2] 2012. 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] 2013. HAProxy load balancer feedback agent check. [Online, Retrieved Oct 14, 2022.] https://www.loadbalancer.org/blog/open-source-windows-service-for-reporting-server-load-back-to-haproxy-load-balancer-feedback-agent/. (2013).

[4] 2015. 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).

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

[6] 2016. Autoscaling in Kubernetes. [Online, Retrieved Jun 12, 2022.] https://kubernetes.io/blog/2016/07/autoscaling-in-kubernetes/. (2016).

[7] 2017. gRPC load balancing. [Online, Retrieved Jun 12, 2022.] https://grpc.io/blog/grpc-load-balancing/. (2017).

[8] 2017. 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).

[9] 2018. Introduction to HAProxy ACLs. [Online, Retrieved Jun 12, 2022.] https://www.haproxy.com/blog/introduction-to-haproxy-acls/. (2018).

[10] 2018. 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).

[11] 2019. Cilium: Socket-based load balancing. [Online, Retrieved Jun 12, 2022.] https://cilium.io/blog/2019/08/20/cilium-16#hostservices. (2019).

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

[13] 2019. Managing Uber's data workflows at scale. [Online, Retrieved Jun 12, 2022.] https://eng.uber.com/managing-data-workflows-at-scale/. (2019).

[14] 2019. Microsoft RPC load balancing. [Online, Retrieved Jun 12, 2022.] https://docs.microsoft.com/en-us/windows/win32/rpc/rpc-load-balancing. (2019).

[15] 2020. Kube-proxy replacement with Direct Server Return. [Online, Retrieved Oct 14, 2022.] https://cilium.io/blog/2020/02/18/cilium-17/#kubeproxy-removal. (2020).

[16] 2020. 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).

[17] 2020. RFC 9000: QUIC: flow control. [Online, Retrieved Jun 12, 2022.] https://www.rfc-editor.org/rfc/rfc9000.html#flow-control. (2020).

[18] 2021. 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).

[19] 2021. gRPC performance best practices. [Online, Retrieved Jun 12, 2022.] https://grpc.io/docs/guides/performance/. (2021).

[20] 2021. How eBPF will solve Service Mesh - Goodbye Sidecars. [Online, Retrieved Jun 12, 2022.] https://isovalent.com/blog/post/2021-12-08-ebpf-servicemesh/. (2021).

[21] 2021. Load balancing algorithms. [Online, Retrieved Oct 14, 2022.] https://docs.citrix.com/en-us/citrix-adc/current-release/load-balancing/load-balancing-customizing-algorithms.html. (2021).

[22] 2021. 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).

[23] 2021. The Human Side of Airbnb's Microservice Architecture. [Online, Retrieved Jun 12, 2022.] https://www.infoq.com/presentations/airbnb-culture-soa/. (2021).

[24] 2022. Automatic mTLS. [Online, Retrieved Jun 12, 2022.] https://linkerd.io/2.11/features/automatic-mtls/. (2022).

[25] 2022. AWS Lambda. [Online, Retrieved Jun 12, 2022.] https://aws.amazon.com/lambda/. (2022).

[26] 2022. 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).

[27] 2022. Google cloud: Load balancing mode. [Online, Retrieved Jun 12, 2022.] https://cloud.google.com/load-balancing/docs/backend-service#balancing-mode. (2022).

[28] 2022. Google cloud: Traffic policies. [Online, Retrieved Jun 12, 2022.] https://cloud.google.com/load-balancing/docs/l7-internal/traffic-management#traffic_policies. (2022).

[29] 2022. Istio: Locality load balancing. [Online, Retrieved Jun 12, 2022.] https://istio.io/latest/docs/tasks/traffic-management/locality-load-balancing/. (2022).

[30] 2022. Kubernetes Networking: Load Balancer and Network Load Balancer. [Online, Retrieved Jun 12, 2022.] https://ibm.github.io/kubernetes-networking/services/loadbalancer/. (2022).

[31] 2022. Kubernetes scheduler. [Online, Retrieved Jun 12, 2022.] https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/. (2022).

[32] 2022. Kubernetes without Kube-Proxy. [Online, Retrieved Jun 12, 2022.] https://docs.cilium.io/en/stable/gettingstarted/kubeproxy-free/. (2022).

[33] 2022. Linux container and virtualization tools. [Online, Retrieved Jun 12, 2022.] https://linuxcontainers.org/. (2022).

[34] 2022. memtier_benchmark. [Online, Retrieved Jun 12, 2022.] https://github.com/RedisLabs/memtier_benchmark/. (2022).

[35] 2022. Microservices and Microservices architecture. [Online, Retrieved Jun 12, 2022.] https://www.intel.com/content/www/us/en/cloud-computing/microservices.html. (2022).

[36] 2022. Next-generation mutual authentication with Cilium service mesh. [Online, Retrieved Jun 12, 2022.] https://isovalent.com/blog/post/2022-05-03-servicemesh-security/. (2022).

[37] 2022. NGINX Plus Feature: Load Balancing. [Online, Retrieved Jun 12, 2022.] https://www.nginx.com/products/nginx/load-balancing. (2022).

[38] 2022. Serverless on AWS. [Online, Retrieved Jun 12, 2022.] https://aws.amazon.com/serverless/. (2022).

[39] 2022. The Kubernetes network model. [Online, Retrieved Jun 12, 2022.] https://kubernetes.io/docs/concepts/services-networking/. (2022).

[40] 2022. The Kubernetes Networking Guide: NodePort. [Online, Retrieved Jun 12, 2022.] https://www.tkng.io/services/nodeport/. (2022).

[41] 2022. There are several Load Balancing Methods. Which one is best for your environment? [Online, Retrieved Oct 14, 2022.] https://support.f5.com/csp/article/K42275060. (2022).

[42] 2022. Use containers to Build, Share and Run your applications. [Online, Retrieved Jun 12, 2022.] https://www.docker.com/resources/what-container. (2022).

[43] 2022. Using nginx as HTTP load balancer. [Online, Retrieved Jun 12, 2022.] https://nginx.org/en/docs/http/load_balancing.html. (2022).

[44] 2022. What is a container? [Online, Retrieved Jun 12, 2022.] https://azure.microsoft.com/en-us/overview/what-is-a-container/#overview. (2022).

[45] 2022. ZeroMQ: Advanced request-reply patterns. [Online, Retrieved Jun 12, 2022.] https://zguide.zeromq.org/docs/chapter3/. (2022).

[46] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, Amin Vahdat, et al. 2010. Hedera: dynamic flow scheduling for data center networks.. In *Nsdi*, Vol. 10. San Jose, USA, 89–92.

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

[48] Maria Apostolaki, Ankit Singla, and Laurent Vanbever. 2021. *Performance-Driven Internet Path Selection*. Association for Computing Machinery, New York, NY, USA, 41–53. https://doi.org/10.1145/3482898.3483366

[49] João Taveira Araújo, Lorenzo Saino, Lennert Buytenhek, and Raul Landa. 2018. Balancing on the edge: Transport affinity without network state. In *Usenix Symposium on Networked Systems Design and Implementation (NSDI)*.

[50] Sachin Ashok, P Brighten Godfrey, and Radhika Mittal. 2021. Leveraging Service Meshes as a New Network Layer. In *Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks*. 229–236.

[51] 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 *Usenix Symposium on Networked Systems Design and Implementation (NSDI)*.

[52] Paul Barford and Mark Crovella. 2000. Critical path analysis of TCP transactions. In *ACM SIGCOMM*.

[53] Brandon Williams. 2012. 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).

[54] Daniel Bristot de Oliveira, Daniel Casini, Rômulo Oliveira, and Tommaso Cucinotta. 2020. Demystifying the Real-Time Linux Scheduling Latency. In *ECRTS*. https://doi.org/10.4230/LIPIcs.ECRTS.2020.9

[55] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. 2016. Borg, Omega, and Kubernetes: Lessons learned from three container-management systems over a decade. *Queue* 14, 1 (2016), 70–93.

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

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

[58] Daniel Borkmann. 2022. 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).

[59] Daniel Borkmann and Martynas Pumputis. 2020. K8s Service Load Balancing with BPF & XDP. https://lpc.events/event/7/contributions/674/attachments/568/1002/plumbers_2020_cilium_load_balancer.pdf. In *Linux Plumbers Conference*.

[60] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (2013), 74–80.

[61] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel hh0Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. https://www.flux.utah.edu/paper/duplyakin-atc19

[62] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. 2016. Maglev: A Fast and Reliable Software Network Load Balancer. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*.

[63] Anwar Elwalid, Cheng Jin, Steven Low, and Indra Widjaja. 2001. 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)*, Vol. 3. IEEE, 1300–1309.

[64] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. 2020. Caladan: Mitigating interference at microsecond timescales. In *Usenix Symposium on Operating Systems Design and Implementation (OSDI)*.

[65] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. 2019. 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*. 3–18.

[66] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. 2019. 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*. 19–33.

[67] Rohan Gandhi, Hongqiang Harry Liu, Y. Charlie Hu, Guohan Lu, Jitendra Padhye, Lihua Yuan, and Ming Zhang. 2014. Duet: Cloud Scale Load Balancing with Hardware and Software. In *Proceedings of the 2014 ACM Conference on SIGCOMM (SIGCOMM '14)*. Association for Computing Machinery, New York, NY, USA, 27–38. https://doi.org/10.1145/2619239.2626317

[68] Mojgan Ghasemi, Theophilus Benson, and Jennifer Rexford. 2017. Dapper: Data plane performance diagnosis of tcp. In *Proceedings of the Symposium on SDN Research (SOSR)*.

[69] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Changhoon Kim, and Nick McKeown. 2021. The nanoPU: A Nanosecond Network Stack for Datacenters. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 239–256. https://www.usenix.org/conference/osdi21/presentation/ibanez

[70] Van Jacobson and Michael J. Karels. 1988. Congestion Avoidance and Control. In *SIGCOMM 1988*. Stanford, CA.

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

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

[73] Hao Jiang and Constantinos Dovrolis. 2002. Passive Estimation of TCP Round-Trip Times. *SIGCOMM Comput. Commun. Rev.* 32 (2002), 75–88.

[74] 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)*. 345–360.

[75] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, et al. 2017. Canopy: An end-to-end performance tracing and analysis system. In *Proceedings of the 26th symposium on operating systems principles*. 34–50.

[76] Srikanth Kandula, Dina Katabi, Bruce Davie, and Anna Charny. 2005. Walking the tightrope: Responsive yet stable traffic engineering. *ACM SIGCOMM Computer Communication Review* (2005).

[77] Dina Katabi, Mark Handley, and Charlie Rohrs. 2002. Congestion control for high bandwidth-delay product networks. In *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*. 89–102.

[78] Jeremy Kerr. 2003. Using Dynamic Feedback to Optimise Load Balancing Decisions. In *Australian Linux Conference*.

[79] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. 2019. {R2P2}: Making {RPCs} first-class datacenter citizens. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 863–880.

[80] Collin Lee and John Ousterhout. 2019. Granular computing. In *Proceedings of the Workshop on Hot Topics in Operating Systems*. 149–154.

[81] Uichin Lee, Joon-Sang Park, MY Sanadidi, Mario Gerla, et al. 2005. Flowbased dynamic load balancing for passive network monitoring.. In *Communications and Computer Networks (CCN)*. 357–362.

[82] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. 2014. Tales of the Tail: Hardware, OS, and Application-Level Sources of Tail Latency. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '14)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/2670979.2670988

[83] Yilong Li, Seo Jin Park, and John Ousterhout. 2021. MilliSort and MilliQuery: Large-Scale Data-Intensive Computing in Milliseconds. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 593–611. https://www.usenix.org/conference/nsdi21/presentation/li-yilong

[84] Zaoxing Liu, Samson Zhou, Ori Rottenstreich, Vladimir Braverman, and Jennifer Rexford. [n. d.]. *Memory-Efficient Performance Monitoring on Programmable Switches with Lean Algorithms*. 31–44. https://doi.org/10.1137/1.9781611976021.3 arXiv:https://epubs.siam.org/doi/pdf/10.1137/1.9781611976021.3

[85] Guohan Lu and Xing Li. 2003. On the correspondency between TCP acknowledgment packet and data packet. In *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement (IMC)*.

[86] Sarah McClure, Amy Ousterhout, Scott Shenker, and Sylvia Ratnasamy. 2022. Efficient Scheduling Policies for Microsecond-Scale Tasks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 1–18. https://www.usenix.org/conference/nsdi22/presentation/mcclure

[87] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. 2015. TIMELY: RTT-based congestion control for the datacenter. *ACM SIGCOMM Computer Communication Review* 45, 4 (2015), 537–550.

[88] Satoru Ohta and Ryuichi Andou. 2009. WWW server load balancing technique based on passive performance measurement. 884 – 887. https://doi.org/10.1109/ECTICON.2009.5137187

[89] Vladimir Olteanu, Alexandru Agache, Andrei Voinescu, and Costin Raiciu. 2018. Stateless datacenter load-balancing with beamer. In *Usenix Symposium on Networked Systems Design and Implementation (NSDI)*.

[90] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. 2015. Making sense of performance in data analytics frameworks. In *Usenix Symposium on Networked Systems Design and Implementation (NSDI)*.

[91] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. 2013. Sparrow: Distributed, Low Latency Scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 69–84. https://doi.org/10.1145/2517349.2522716

[92] Jitendra Pahdye and Sally Floyd. 2001. On inferring TCP behavior. In *ACM SIGCOMM*.

[93] Austin Parker, Daniel Spoonhower, Jonathan Mace, Ben Sigelman, and Rebecca Isaacs. 2020. *Distributed tracing in practice: Instrumenting, analyzing, and debugging microservices*. O'Reilly Media.

[94] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A. Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, Changhoon Kim, and Naveen Karri. 2013. Ananta: Cloud Scale Load Balancing. *SIGCOMM Comput. Commun. Rev.* 43 (2013), 207–218.

[95] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. 2014. Fastpass: A centralized" zero-queue" datacenter network. In *Proceedings of the 2014 ACM conference on SIGCOMM*. 307–318.

[96] 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

[97] Hugo Sadok, Zhipeng Zhao, Valerie Choung, Nirav Atre, Daniel S Berger, James C Hoe, Aurojit Panda, and Justine Sherry. 2021. We need kernel interposition over the network dataplane. In *Proceedings of the Workshop on Hot Topics in Operating Systems*. 152–158.

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

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

[100] Nikita V. Shirokov. 2018. XDP: 1.5 years in production. Evolution and lessons learned.. http://vger.kernel.org/lpc_net2018_talks/LPC_XDP_Shirokov_v2.pdf. In *Linux Plumbers Conference*.

[101] Benjamin H Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. Dapper, a large-scale distributed systems tracing infrastructure. (2010).

[102] Shan Sinha, Srikanth Kandula, and Dina Katabi. 2004. Harnessing TCP's burstiness with flowlet switching. In *Proc. 3rd ACM Workshop on Hot Topics in Networks (Hotnets-III)*.

[103] Lalith Suresh, Marco Canini, Stefan Schmid, and Anja Feldmann. 2015. C3: Cutting Tail Latency in Cloud Data Stores via Adaptive Replica Selection. In *Usenix Symposium on Networked Systems Design and Implementation (NSDI)*. 513–527.

[104] Michal Szymaniak, David Presotto, Guillaume Pierre, and Maarten van Steen. 2008. Practical large-scale latency estimation. *Computer Networks* 52, 7 (2008), 1343–1364. https://doi.org/10.1016/j.comnet.2007.11.022

[105] Erico Vanini, Rong Pan, Mohammad Alizadeh, Parvin Taheri, and Tom Edsall. 2017. Let it flow: Resilient asymmetric load balancing with flowlet switching. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 407–420.

[106] Bryan Veal, Kang Li, and David Lowenthal. 2005. New methods for passive estimation of TCP round-trip times. In *International workshop on passive and active network measurement*. Springer, 121–134.

[107] Wenfei Wu, Guohui Wang, Aditya Akella, and Anees Shaikh. 2013. Virtual network diagnosis as a service. In *Proceedings of the 4th annual Symposium on Cloud Computing*. 1–15.

[108] Yin Zhang, Lee Breslau, Vern Paxson, and Scott Shenker. 2002. On the characteristics and origins of internet flow rates. In *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*. 309–322.

[109] Hang Zhu, Kostis Kaffes, Zixu Chen, Zhenming Liu, Christos Kozyrakis, Ion Stoica, and Xin Jin. 2020. RackSched: A Microsecond-Scale Scheduler for Rack-Scale Computers. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association.

[110] Danyang Zhuo, Kaiyuan Zhang, Yibo Zhu, Hongqiang Harry Liu, Matthew Rockett, Arvind Krishnamurthy, and Thomas Anderson. 2019. Slim: OS kernel support for a low-overhead container overlay network. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 331–344.