Measuring Response Latencies Under Asymmetric Routing

Bhavana Vannarth Shobhana, Yen-Lin Chien, Mark Doughten, Jonathan Diamant, Badri Nath, Shir Landau-Feibish, and Srinivas Narayana

ABSTRACT

Latency is a key indicator of Internet service performance. Continuously tracking the latency of client requests enables service operators to quickly identify bottlenecks, perform adaptive resource allocation or routing, and mitigate attacks. Passively measuring the response latency at intermediate vantage points is attractive since it provides insight into the experience of real clients without requiring client instrumentation or incurring probing overheads.

We argue that existing passive measurement techniques have not caught up with recent trends in service deployments, specifically, the increasing prevalence of obscured or encrypted transport headers, and the use of asymmetric routing by design. Existing methods are inapplicable, inaccurate, or inefficient.

This paper presents Pirate, a passive approach to measure response latencies when only the client-to-server traffic is visible, even when transport headers are invisible. Pirate estimates the time gap between *causal pairs*—two requests such that the response to the first triggered the second—as a proxy for the client-side response latency. Our experiments with a realistic web application show that Pirate can accurately estimate the response latencies measured at the client application layer. A Pirate-enhanced layer-4 load balancer (with DSR) cuts tail latencies by 37%.

1 INTRODUCTION

Latency is a key indicator of the performance and quality of interactive Internet services. For developers of such services, it is well known that smaller client-visible latencies drive better user engagement [5, 7, 14, 19, 89]. Given its primacy, accurate measurements of latency can feed important decisions in designing and adapting networked systems. For example, operators of some large content delivery services use the latency between content servers and users to determine which servers to redirect users to [91, 119]. Autonomous Systems (ASes) on the wide-area Internet may use high or variable latencies experienced by transiting connections to identify pathologies such as persistent link congestion [61, 63, 116, 117] or interdomain route hijack [11, 23, 51], and take corrective actions to fix routing configurations or provision capacity [95]. Within a data center, server latency may be used to implement performance-optimized replica selection in load balancers [53, 72] or remote procedure call (RPC) clients [118, 127].

Continuous measurement of client-visible latency is crucial since the latency may change over the lifetime of a connection [48, 54, 64, 78]. For example, time-varying bursts and packet losses in the network [57, 74], or server variability due to noisy neighbors, load, and resource scheduling [54, 56, 71, 96], can significantly change the latency perceived by the same client connection over time.

Consequently, the community has developed several techniques and systems for continuous latency measurement. Broadly, *active approaches* send explicit probes that observe latency, for example, by running ICMP pings between servers [77]. *Passive approaches* observe latency at strategic locations that can be controlled [119, 130, 132], possibly at intermediate vantage points outside the client and the server. Passive approaches do not require instrumentation or self-reporting from clients which may be untrustworthy or not easily changed [119]. If passive measurement is possible, it can observe real and representative client connections [54], avoiding the typical downsides of active approaches that incur compute and network resources for probing [60, 90].

This Paper. We are interested in passively and continuously measuring *response latencies,* which we define as the time between when an application-layer request is sent and the last byte of the response is delivered to the client application, *i.e.* the per-object time to last byte. For RPCs within data centers, response latency corresponds to the RPC completion time at the application layer [83, 94]. For web-based applications, the response latencies of specific objects (*e.g.* those first painted or largest on the user's screen) are highly correlated with several quality of experience metrics [27, 35, 38, 55]. As we elaborate in §2.1, to be applicable to a broad range of scenarios, we seek techniques to passively measure response latency while meeting the following additional requirements:

(1) handle missing or encrypted transport headers;
(2) handle routing asymmetry;
(3) generalize across transport dynamics;
(4) support efficient deployment on software middleboxes.

Application-layer response latency is most closely related to the transport-layer round-trip time (RTT), whose passive measurement is widely studied, *e.g.* [59, 64, 75, 81, 82, 111, 112, 123]. However, response latency is distinct from the

transport-layer RTT, since the server may return a transport-layer acknowledgment well before the full application-layer response. As far as we are aware, no existing RTT measurement technique can meet the additional requirements above, and we know of no passive techniques to directly measure response latency (§2.2).

Our Key Ideas. This paper uses three key ideas to meet our measurement goals (§3).

Our first idea is to leverage the closed-loop nature of Internet protocols and applications. When application requests depend on the contents of prior responses—for example, a web object embeds other objects—the reception of a prior response generates the subsequent request. Flow-controlled applications (transports) mandate that new requests (packets) are only transmitted when prior responses (acknowledgments) arrive. Such closed-loop packet transmission behavior enables estimating the response latency by proxy: the vantage point can measure the time delay between a request and a subsequent request that was triggered by the reception of the response to the first request. We call the latter request a *causally-triggered request,* and the pair of requests a *causal pair.* Many latency-sensitive applications exhibit cross-request dependencies (*e.g.* web [101], RPC [30, 71, 97, 114]) and flow control (*e.g.* key-value store [103], web [13, 20]).

It is not obvious how to identify causal pairs. At any given time, many concurrent requests may be in flight. Connection persistence and stream multiplexing is standardized and widely deployed in the HTTP protocols [1, 12, 24]. Two consecutive requests arriving at a vantage point from the same connection do not necessarily form a causal pair.

Our second key idea is to leverage the time gaps between packet arrivals to identify causal pairs. Once a client has sent a few requests, its transmission of subsequent requests is typically blocked by dependencies or flow control. Hence, causally-triggered requests arrive after a time gap that is longer than the inter-arrival times of the packets just prior. We call this the *prominent packet gap assumption.* By choosing an appropriate threshold on packet inter-arrival times, a vantage point can classify any packet arriving with a time gap exceeding this threshold as a causally-triggered request. The time interval between two consecutive causally-triggered requests provides one estimate of the response latency.

But what should the time threshold be set to? How can we make this setting robust across deployments (*e.g.* wide area *vs* data center), network load, transport dynamics (*e.g.* TCP flavors), and applications (*e.g.* web *vs* key-value store)?

Our third key idea is the following: While a single prominent packet gap is subject to (unknown) network, application, and server conditions, observing packet inter-arrivals *over a period of time* provides a more robust picture. In this paper, we leverage the probability distribution of inter-arrival times to identify prominent packet gaps, by designing a lightweight construction to measure a coarse histogram of all packet inter-arrival times within a connection. We also devise a procedure to estimate the average response latency over the (configurable) time period when the distribution was measured.

We call our algorithm, a synthesis of the three ideas above, Pirate. We show how to use measurements from Pirate in a feedback control loop to adapt a layer-4 load balancer. The load balancer may use direct server return [105] to avoid processing the responses. Since Pirate works under routing asymmetry, it is possible to make latency-aware decisions by unilaterally changing the load balancer, leaving clients, servers, application software, and the network unmodified.

Results. In §4, we evaluate Pirate under a realistic web workload derived from the Alexa top-100 web sites, with a web server whose CPU availability varies according to a real CPU utilization trace [124]. Across all monitored responses, Pirate achieves a median relative error of 0.63% relative to the response latency measured at the client application, and gets to within ±15% error 90% of the time. Our results also show that techniques to measure transport-layer RTT do not faithfully measure application-layer response latency, even when such techniques can take advantage of bidirectional traffic visibility. We integrated Pirate into Katran [16], an open-source layer-4 load balancer based on Maglev hashing [67]. Latency-aware Katran cuts the 99th percentile latency in our experimental setup by 37% on average across loads, relative to vanilla Katran.

We outline caveats when interpreting our results in §5. A shorter version of this paper appeared previously at a workshop venue (citation elided for anonymity). In comparison, this paper incorporates new techniques to make estimation more robust, significantly expands evaluations of accuracy and overhead under realistic settings, and designs a feedback controller that operates on real-time measurements. **This paper does not raise any ethical issues.**

2 MOTIVATION AND BACKGROUND

Monitoring latency is a fundamental requirement for the design, maintenance, and optimization of interactive Internet services. This paper studies the problem of measuring *response latency*, which we define as the time between when an application client sends out the request and when it receives the last byte of the corresponding response at the application layer. The client may be a user device contacting a web server or an RPC client in one tier of a distributed multi-tiered application.

2.1 Our Goals

We seek a passive measurement approach to continuously measure response latency at a vantage point which can lie outside the client and the server, along the path from the client to the server. To make it more broadly applicable, we also seek to meet the following more specific goals.

G1. Handle missing or obscured transport headers. Encryption of application-layer payloads is ubiquitous on the web and inside compute clusters [26, 44–46]. Modern network-layer security goes one level further, obscuring transport-layer headers. Deployments of network-layer encrypted tunnels (*e.g.* WireGuard [43], IPSec [4]) have been trending upward [41]. Even when transport is unencrypted, overlay networks and IP fragmentation could result in transport headers missing from packets. We desire a method that does not require visibility into the transport layer headers.

G2. Work under routing asymmetry. Nodes conducting passive measurement may be at locations which do not have access to both directions of traffic flowing between the client and the server. The prevalence of routing asymmetry is well established in the wide-area Internet [106] and also in data centers [131]. Further, many network deployments use asymmetric routing *by design*. For example, layer-4 load balancers employ direct server return [67, 105], bypassing the load balancer for response packets. BGP policies on the Internet (*e.g.* stub networks) may choose different ingress and egress border routers for a given connection [86, 120].

G3. Generalize across transport dynamics. The design and deployment of new transport algorithms for congestion control and loss recovery is an active and rapidly evolving area [68]. We seek techniques that avoid relying on specific behaviors of the transport protocol, and sidestep the need for new extensions to transport protocols.

G4. Efficient enough to run in software middleboxes. Measurement and monitoring devices are frequently deployed in the form of virtual network functions managed through a software-defined measurement infrastructure, *e.g.* [42, 104]. We aim for techniques that impose low compute and memory overheads in software deployments, *e.g.* middleboxes running high-speed packet processing.

2.2 Prior Work and Its Applicability

In the academic literature, the problem most closely related to passively estimating response latency is the passive measurement of round-trip times (RTTs) at the transport layer.

RTT estimation approaches. The basic idea of passive RTT estimation is relating data packets to their corresponding transport-layer acknowledgments (ACKs) and measuring the time difference between this pair of packets. Many prior works on RTT estimation determine this relationship by connecting the sequence numbers or timestamps on data packets to the corresponding information echoed on ACK packets [64, 75, 81, 111, 112, 123]. Implementing this basic approach requires visibility into transport headers in the clear (**G1**), and seeing packets in both directions (**G2**).

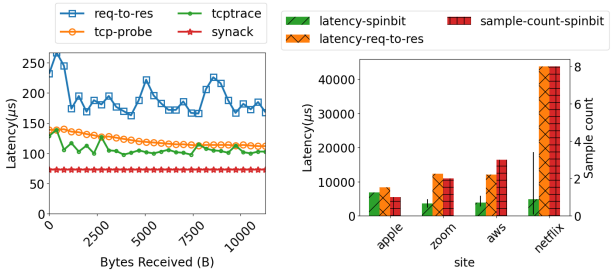


Figure 1: Transport round-trip times (RTTs) do not match response latency, with clients running (a) TCP HTTP/1.1 and (b) UDP QUIC. See discussion in §2.2.

We are aware of only a small number of techniques which apply when the passive observer sees packet traffic in only one direction. An early approach from Jiang and Dovrolis [82] estimates RTT through the time interval between the SYN and the ACK packet of the 3-way handshake, and also between small packet bursts during the early rounds of slow start. These techniques are customized to the specific dynamics of the TCP protocol (**G3**) and do not measure connections beyond the beginning. Another set of prior works estimate RTTs by computing frequency spectra over the time series of packet arrivals [59, 123]. Such computations require significant memory to hold packet timings (**G4**), and can be complex to tune [64]. Most recently, researchers have proposed the spin bit as a protocol extension (**G3**) to passively measure RTTs [62]. It has been incorporated as an optional mechanism in QUIC [17], but its deployment may be limited [92].

Response latency is dissimilar to the RTT. A high RTT implies a high response latency, but a low RTT does not necessarily imply a low response latency. In principle, the response latency can significantly differ from the transport-layer RTT. The server can transmit a transport-layer ACK to a client prior to transmitting any response data. Further, responses, typically larger than requests, may span multiple packets, widening the gap between the transport-layer ACK and the last byte of the response. When the client and server use a protocol with pipelined requests (*e.g.* HTTP/1.1), multiple application requests may be transmitted in a single packet from client to server. The transport-layer ACK only corresponds to the first response.

Experimentally, we find that RTT and response latency differ even when there is only one small application-layer request in flight. We ran a web benchmark where a TCP

client maintains a single outstanding request at a time with the server. Our full experimental setup is described in §4.1. On one client thread, Figure 1 (a) shows the time series of response latencies (marked `req-to-res`) against three techniques to measure transport RTT: (i) reading statistics from the kernel network stack through `tcp-probe` [33]; (ii) estimating RTT from a packet trace (`tcptrace` [112]; and (iii) syn-ack [82]. Not only is the response latency different from the RTT, but it is also uncorrelated.

We ran a similar experiment to compare the RTT of a QUIC connection measured using the spin protocol extension [62] with the application-level response latency, measured with a headless browser on a real web page. Figure 1 (b) shows that the two values diverge, especially when there are multiple RTT samples corresponding to a single response latency. The error bars show the minimum and maximum RTT reported through spin.

Other related work. Given the emerging prevalence of encrypted transports, some recent work aims to passively infer quality of experience metrics specifically for web applications [79, 125]. These metrics compose the response latencies of multiple objects, *e.g.* to estimate overall web experience [39], through machine learning and prediction. In contrast, we are concerned with continuous measurement of response latencies of individual objects throughout the lifetime of a connection. There is also significant prior work on passively measuring the latency of specific segments of a network, *e.g.* [57, 74, 90, 93] and algorithms to estimate latencies over end-to-end network paths using previously measured delays, *e.g.* [98, 102]. Plenty of active approaches exist to measure end-to-end network latency characteristics, *e.g.* [116, 117, 129]. This paper seeks continuous passive measurement techniques for end-to-end response latency.

In summary, we believe that passive continuous measurement of response latencies under modern transports and practical deployment constraints (§2.1) is as yet unsolved.

3 PIRATE

This section introduces an algorithm to measure response latency passively and continuously for interactive applications (§2.1). We assume that the vantage point of measurement lies on the path from the client to the server.

3.1 Causal Pairs

Instead of measuring the time interval between a request and its response, our first key idea is to measure a proxy time interval—the time between a request and a packet transmitted by the client due to the reception of the corresponding response. We call the latter packet a *causally-triggered request*, and the pair of packets a *causal pair*.

Causally-triggered requests exist due to several reasons: (1) *Cross-request dependencies:* Many interactive applications issue subsequent requests only when responses to previous requests have been received and processed by the client application. For example, web clients and RPC clients issue follow-up requests to fetch objects based on previous responses. (2) *Flow control:* Clients frequently cap the number of application or transport data that are outstanding at the server. Web browsers, memcached clients, and RPC client libraries are all known to subject clients to such flow control [13, 20, 32]. (3) *Acknowledgments.* A response may trigger an acknowledgment at the transport or application layer.

If a client application maintains exactly one request in flight, with the next request only issued once the response to the first one arrives, the two requests form a causal pair, and measuring the time interval between the two requests provides an estimate of the response latency. Causal pairs are a generalization of syn-ack estimation [82].

There are two caveats to using causal pairs to estimate response latency. First, a client application may incur additional time to process a response prior to generating a follow-up request, due to considerations such as parsing or delays due to process or thread scheduling. We call this the *client think time.* In experiments on lightly-loaded machines, we have measured client think times of a few microseconds to a few tens of microseconds.Second, a response latency measurement is only available for one request per causal pair, not all the requests transmitted between the two requests on the same connection.

A key challenge arises when we go to settings where multiple requests or packets are concurrently in flight. Two consecutive requests observed at a vantage point from the same connection need not form a causal pair. Accurate knowledge of the client's ongoing window size (at the transport or application layer) may help identify packets that are not causally related with each other, *e.g.* if they belong to the same window. However, inferring the window size typically itself requires assumptions on the dynamics of the transport protocol in question [75, 81]. This violates goal **G3** (§2.1).

3.2 Prominent Packet Gap Assumption

Our second key idea is to leverage the timings of packet arrivals to identify causal pairs when a client has many concurrent requests in flight. The reasons that produce causally-triggered requests (cross-request dependencies, flow control, acknowledgments) result in clients transmitting a burst of data in one shot, and then pausing request transmission until a response arrives. The sender side of transport often transmits in bursts to leverage hardware batching optimizations (*e.g.* TSO [47]), while receivers typically optimize I/O and CPU by batching ACK processing [21, 47]. The prevalence of

burst-then-pause behavior is well documented: TCP senders that use window-based transmission send bursts of packets, termed flowlets [115, 121], separated by a pause, termed the flowlet gap.

We exploit the observation that in many practical scenarios, the pause is noticeably longer and more prominent than the packet inter-arrival times in the burst, since the former is subject to delays from the network and server processing, while the latter is determined only by the ability of the client to transmit requests. Hence, the first packet arriving after the pause must be a causally-triggered request. The time interval between two consecutive causally-triggered requests provides an estimate of the response latency.

The assumption of a prominent packet gap does not hold universally. Packets arriving at the vantage point may be uniformly paced either by the sender's transport, *e.g.* [58, 76] or a bottleneck link or policer [70, 85]. We conjecture that latency-sensitive applications transmitting small amounts of bursty traffic may not be widely subject to pacing, capacity bottlenecks, or policing.

Given a fixed time threshold δ, packets which arrive with an inter-arrival time gap of more than δ are considered to be a part of different bursts. We estimate the response latency by the time between the first packets of successive bursts. This algorithm is shown in Algorithm 1 in Appendix A.

How should we choose the time threshold δ? The threshold substantially impacts the accuracy of our estimate. Choosing too large a δ will miss legitimate causal pairs, only identifying long idle periods in the connection. Choosing too small a δ can make the estimation vulnerable to noisy gaps between packets transmitted within the "burst," for example due to scheduling or processing delays at the client's application or transport layer.

Fundamentally, the duration of the pause depends on several factors, such as the timing of writes from the client application into the transport layer, the client transport's scheduling of packet transmission (*i.e.* flow and congestion control), scheduling at the network stack's traffic control layer or the NIC [110], cross traffic competing with the connection before packets arrive at the vantage point, the network round-trip time, and server processing delays. The combination of these factors makes it unlikely that a fixed threshold δ can work across different scenarios, or across time even for a single connection in a specific network. Approaches that select time thresholds for flowlet-based network load balancing [49, 84, 115, 121] do so using the differences between expected path latencies, or through an empirically-tuned value that achieves a desired balance among paths. These approaches are unsuitable for a passive observer attempting to measure latency in the first place.

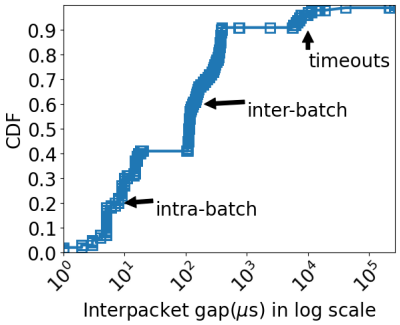


Figure 2: Modes in the empirical distribution of inter-packet gaps (IPGs). Modes carry useful information about phenomena of interest occurring over the measurement epoch.

3.3 Choosing a Packet Gap Threshold

Our third key idea is that observing the timings of packet arrivals over time can provide meaningful clues to a good value for the pause time threshold δ (§3.2). This observation is inspired by prior work that motives a longitudinal view of noisy data as a way to design robust measurement and feedback control [73, 87, 109, 126]. Instead of producing an estimate instantaneously, we can produce an average over a configurable time epoch.

Specifically, given an empirical distribution of inter-packet time gaps (IPGs) observed over the lifetime of a connection, the significant *modes* of the probability distribution carry information about specific events occurring on that connection. In Figure 2, we show an empirical distribution of IPGs in a simple experimental scenario where a TCP client has a roughly constant response latency whenever the response arrives, but experiences occasional packet losses and idle periods (where no data is either transmitted or received). The IPG distribution includes a batch of packets within a burst (smallest-valued modes); IPGs across bursts of causally-dependent packets; loss timeouts (typically set larger than response latencies); and idle periods (the largest modes).

Through knowledge of standard parameters used for retransmission timeouts in transport stacks (publicly known for several standard TCP and QUIC configurations [8, 22, 122]), it is possible to eliminate the modes corresponding to retransmission timeouts and idle periods, leaving behind the de-noised distribution of IPGs, which only includes gaps between request packets that are either (i) back-to-back transmissions without causal dependencies, or (ii) causally-triggered requests sent after a prominent packet gap.

Computing a proportional mode sum. We devise a simple estimation procedure that assumes that (i) the IPG distribution is representative of phenomena over the epoch where the distribution is maintained; and (ii) the largest mode in the

de-noised IPG distribution is the IPG preceding the arrival of a causally-triggered request. We call this IPG the *inter-batch gap (IBG)*. The average response delay can be estimated by summing up the modes smaller than the IBG, weighted by their frequency relative to the IBG. For example, suppose the IPG distribution has three modes m_1 = 100 μs, m_2 = 150 μs, and m_3 = 250 μs (after de-noising), with corresponding probabilities $Pr(m_1) = 0.4, Pr(m_2) = 0.2$, and $Pr(m_3) = 0.1$. We assume that m_3 (the largest-valued mode) is the prominent packet gap (§3.2). Corresponding to each occurrence of m_3, there are $Pr(m_1)/Pr(m_3) = 4$ occurrences of m_1 and $Pr(m_2)/Pr(m_3) = 2$ occurrences of m_2. We estimate the average response latency as $4 * m_1 + 2 * m_2 + m_3 = 950\mu$s. More generally, the proportional mode sum estimate can be summarized by the expression $\sum_{m_i \leq IBG} \frac{Pr(m_i)}{Pr(IBG)} * m_i$.

Our use of IPG distributions is distinct from prior techniques that probe the network using packet trains and use IPG distributions to estimate bottleneck link bandwidth, *e.g.* [65, 108]. The average response latency is the sum of some number of packet gaps; identifying which packet gaps to combine and in what proportion is the core challenge of latency measurement, which prior work does not tackle.

While IPG distributions provide useful information, computing distributions by maintaining the full list of IPGs (say, on a software middlebox) for each active connection is prohibitively memory-expensive. Maintaining histograms is also expensive: With a sufficiently fine-grained time resolution, say, 10 μs (the RTT value in some data centers [34, 69]), an epoch length of 100 ms and a 2-byte counter per bucket of the histogram, one connection would require 20 KBytes of memory. For a vantage point that sees 10K active connections, the memory consumption of the histogram touches 100 MBytes. This value is larger than the L2 caches on many server architectures, and could imply a substantial slowdown in packet-processing performance. Using a coarse time resolution will sacrifice accuracy when response latencies are small, preventing the algorithm from distinguishing modes finer than the time resolution used.

3.4 Maintaining Efficient Histograms

To maintain an IPG distribution with memory efficiency, we design an algorithm that maintains a small number of buckets by dynamically varying the resolution of each bucket according to the IPGs observed on that connection. Dynamic bucket resolution has the advantages that (1) the histograms of different connections may freely span different ranges of IPGs; (2) we can avoid the overhead of bucket counters for IPG values that do not occur; and instead, (3) focus the available memory on IPG values that are actually observed.

Our algorithm works as follows. The histogram, M, includes a (configurable) maximum number of buckets (modes) N. Initially, all buckets are uninitialized. Each bucket (after initialization) specifies the minimum, maximum, count, and sum of all the IPGs observed within that bucket. For each observed IPG g, the algorithm either (i) counts g into an existing bucket whose (min, max) includes g, or (ii) if g is "close enough" to an existing bucket, extends the bucket or merges two buckets to produce a new, larger bucket that now counts g, or (iii) puts g into its own new bucket if there is an available uninitialized bucket, or (iv) discards g. As few as $N = 10$ buckets (maximum number of modes) per connection prove sufficient to capture all IPG distributions in our experiments. At the end of each epoch, the proportional mode sum (§3.3) is computed over the average mode values of M to emit a response latency averaged over the epoch. Our complete algorithm is shown in Algorithm 2 in Appendix B.

Similar to ours, there exist storage-efficient algorithms in the streaming setting, where a dynamic bucket size may be used to maintain modes efficiently, *e.g.* [100]. We leave the adaptation of such algorithms to our scenario and a quantitative comparison to future work, since our experiments (§4) show that our current algorithm is practical.

3.5 A Latency-Aware DSR Layer-4 Load Balancer

To showcase the utility of real-time response latency measurement under asymmetric routing, we show the design of a latency-optimizing layer-4 load balancer. Layer-4 load balancers implement direct server return (DSR), a mechanism to allow backend servers to send responses directly to the client, bypassing the load balancer [105] on the path from the server to the client. To our knowledge, this work is the first to enable the availability of real-time latency signals at DSR load balancers unilaterally without modifying the client, server, application, or the network. We do not claim any novelty in our algorithm design; there is a significant body of algorithmic work on performance-aware request load balancing, *e.g.* [31, 31, 53, 72, 88, 99, 118, 127]. Below, we describe one way in which a feedback controller can be designed around latency measurements.

Our design augments a Maglev-hashing-based load balancer [67]. We assume that the load balancer provides mechanisms to assign weights to distribute the load across servers. We use three key ideas to assign the weights. First, we take away weights from servers with latencies larger than a high watermark, and place them on servers with latencies smaller than a low watermark. Second, we limit the addition of weight to servers that have insufficient recency ("freshness") in their latency measurement. Third, we regress to the mean by slowly equalizing weights across servers when latencies have not changed recently. Appendix C presents a full discussion of these ideas and our design.

4 EVALUATION

In this section, we empirically evaluate PIRATE under the experimental settings described in §4.1. We ask the following questions:
(§4.2) How accurate is PIRATE in measuring application-level response latencies under realistic applications and settings? How does PIRATE compare to transport RTT estimators?
(§4.3) What overheads does PIRATE impose on a software middlebox?
(§4.4) Can measurements from PIRATE enable real-time robust feedback control?
(§4.5) How well does each core idea from PIRATE work in realistic settings?
(§4.6) Is PIRATE robust to network and load variability?

4.1 Experimental Setup

***Implementation of* PIRATE.** PIRATE runs as a kernel bypass program developed in Linux eBPF, which attaches to the express data path (XDP [80]) hook, which resides in the network device driver in the kernel. We use a standalone XDP forwarder that implements the PIRATE algorithm for the accuracy experiments. For our evaluation of the feedback controller, we instrumented the Katran layer-4 load balancer [113], developed and open sourced by Meta. PIRATE runs as a program that calls Katran using a BPF tail call. Katran implements direct server return [105]. The measurement component of PIRATE constitutes approximately 500 lines of C code.

Web benchmarking framework. We use the WebPolygraph suite [3] as our HTTP/1.1 client and server for benchmarking. WebPolygraph allows detailed configuration of workload characteristics such as (1) the number of benchmarking threads, offered load (requests per second) initiated from each benchmarking client thread, and the number of concurrent requests on persistent pipelined HTTP connections over TCP; (2) the types of objects (*e.g.*jpeg, *etc.*) and their prevalence (*e.g.* 20%) among the requested objects; (3) properties of the dependency tree of objects requested starting from the root URL requested by the client (*e.g.* an object of type T includes N_a objects of type a, N_b objects of type b, and so on, where the values of the $N_{(.)}$ can be drawn from a custom probability distribution; and (4) the probability distributions of the sizes of each object type. The WebPolygraph server is single-threaded. With our client workload (see below) and server machine, each client-to-server connection saturates a single CPU core at a load of 4K requests/second.

Client workload. We populate web page parameters (types, sizes, object dependencies) into WebPolygraph by creating a dataset of web objects from the home pages of a random sample of 20 unique web sites from the Alexa top-100. Each HTTP request issued by the client fetches one of 4 kinds of objects (HTML, CSS, JS, image), with the response size drawn from the empirical size distribution of the corresponding object type in our dataset (CSS: median 11.8 KByte, max 1.2 MByte; JS: median 19.6 KByte, max 4.05 MByte; HTML: median 39.5 KByte, max 2.65 MByte; image: median 20.8 KByte, max 8.86 MByte). Unless specified otherwise, our client runs on one machine with 20 threads and 200 connections total, driving an aggregate workload of 2K requests/sec. To simulate dependent HTTP requests created as a result of parsing HTTP responses (*e.g.* images embedded in HTML), each response returned by the server includes references to different kinds of embedded objects drawn from the corresponding empirical probability distribution in our dataset. Connections are persistent and pipelined. Unless stated otherwise, each WebPolygraph client connection maintains at most 4 outstanding requests, comparable to default limits in the HTTP standard and in popular browsers [1, 2, 6, 9, 15]. Each active connection may fetch between 1–7000 requests, with a median of 2094 HTTP objects fetched per connection.

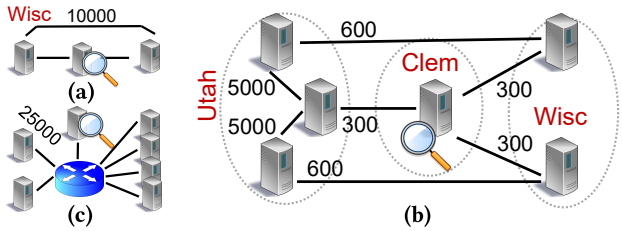


Figure 3: The experimental topologies we use on CloudLab [66]. Most experiments use the single cluster "triangle" topology (a). We also run experiments on a wide-area topology (b) and a star topology (c). Traffic flows from the clients (left) to servers (right) via the measurement vantage point (magnifying glass). Server to client traffic bypasses the vantage point. Link bandwidths (Mbit/s) are shown.

Server performance variability. We derive a time series of CPU allocations for our benchmark web server from a real CPU utilization trace of Google's Borg cluster jobs [18, 124] (2019 trace). The trace provides histograms of CPU usage over a 5-minute period, where each sample collected in the original data is over one second. We select the time series of the job with the largest average CPU utilization in the trace, sample from the histograms, and enforce the resulting CPU allocation on the WebPolygraph server using Linux cgroups.

Topology. We set up our implementation of PIRATE and the client/server workloads on CloudLab [66].

Most experiments use three machines in a single cluster connected in a triangle topology (Figure 3 (a)). The three machines consist of a client, a measurement vantage point, and a server. Each machine includes two Intel E5-2630 8-core

CPUs, 128GB memory, and a dual-port 10Gb NIC. Requests are routed from the client to the server via the vantage point. Responses go directly from server to client, bypassing the vantage point. Without any network load, the round-trip time between the client and the server in this setup is 227μs.

We also evaluate Pirate in wide-area settings with the topology in Figure 3 (b). A client machine and a contending traffic source reside in the CloudLab Utah site, and send traffic over a bottlenecked wide-area link to the Clemson site, which runs the measurement vantage point. Corresponding servers reside in the Wisconsin site. Servers return traffic to the client directly, bypassing the measurement vantage point. This network exhibits a propagation round-trip time of 70 milliseconds. Link bandwidths (Mbit/s) are labeled in the figure. The bandwidths along the forward and reverse path are asymmetric.

To test load balancing, we also employ a star topology (Figure 3 (c)), with two clients connecting to four servers via a measurement vantage point. The seven server machines include an Intel E5-2640v4 10-core CPU, 64GB memory, and dual-port 25Gb NIC. A Dell S4048 switch interconnects the machines. We configured the switch and all the servers to forward packets from clients to servers via the vantage point, and server-to-client packets directly via the switch, mimicking a direct server return configuration [29]. The Pirate algorithm runs on the switch-facing ingress interface of the vantage point.

Techniques compared.

(1) Ground truth response latency (`req-to-res`): We instrumented the WebPolygraph client to compute the time delay between the transmission of each request and the reception of the complete response object for that request at the application layer. This is our ground truth for response latency, labeled `req-to-res` on the graphs.

(2) Causal pair delay (`req-to-req`): Even when the reception of a complete response could trigger another request (*i.e.* a causally-triggered request), a client application may incur additional time to transmit the latter, due to processing (*e.g.* parsing) and scheduling delays. Since Pirate approximates the time delay between the causal pair, for reference, we also show the time between the request and the triggered request, labeled `req-to-req` on the graphs.

(3) Transport RTT measurement: We compare Pirate against transport-level RTT estimators, specifically (1) `tcptrace` [112], an open-source tool that analyzes pcap traces, (2) `tcp_probe` [33], which emits the sample RTTs maintained by the Linux network stack; and (3) syn-ack delay [82, 119], the time between the SYN and the ACK packets in the TCP 3-way handshake.

Syn-ack only produces one measurement per connection. We ensure that the values reported by the rest of the techniques above are directly comparable, by aligning the transport-layer sequence numbers represented in the measurements. All techniques except Pirate and syn-ack can see both directions of network traffic.

Accuracy metrics. When we compare a measurement technique T against a baseline technique B, we report the absolute error $latency_B - latency_T$ and the relative error $(latency_B - latency_T)/latency_B$ as a percentage.

4.2 Accuracy of Latency Estimation

How close does Pirate get to the ground truth response latency at the client application (`req-to-res`)?

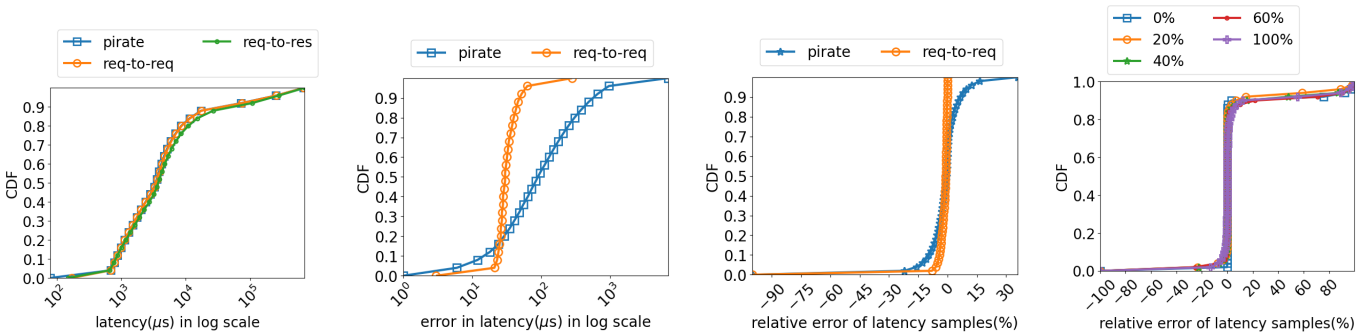
Single-cluster setting. In a triangle network topology (Figure 3 (a)), our client offers a load of 2K requests/second in total to the server across all threads and connections.

Figure 4 (a) shows the CDF of the response latencies at the client (`req-to-res`) and the vantage point (Pirate) across all objects over all connections. We also show the time delay between the causal pair (`req-to-req`), which Pirate approximates. The three distributions are closely aligned, providing confidence that (i) the time delay between causal pairs is a good estimator of response latency, and (ii) that Pirate can estimate the latter with high accuracy, in realistic settings where the client maintains persistent and pipelined connections with multiple concurrent application-layer requests over each connection (§4.1).

Figure 4 (b) shows the CDF of the absolute error of Pirate and `req-to-req` against `req-to-res`, while Figure 4 (c) shows the CDF of the relative errors. The causal pair delay is always an overestimate of the response latency, showing a relative error between −10% to 0%. Pirate's relative error is less than 1% at the median, but stretches beyond ±10% outside $10^{th} - 90^{th}$ pc, and beyond ±15% at the tails beyond $5^{th} - 95^{th}$ pc. In a state of the art where response latency measurement is nonexistent under asymmetric routing, an estimator that gets within a 15% error 90% of the time is useful (§3.5; §4.4).

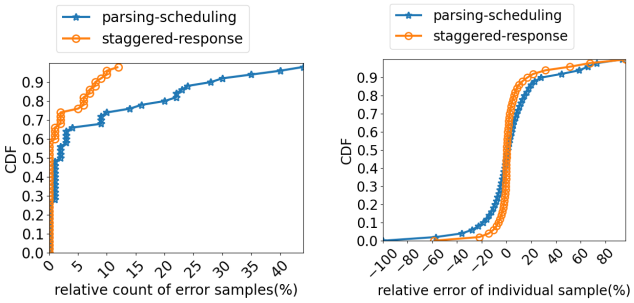
Why does* Pirate *make errors? An investigation of when and how significantly Pirate miscomputes response latencies in the experiment above reveals two kinds of errors. Figure 5 measures their prevalence and magnitude.

The first kind of error, labeled `parsing-scheduling`, arises when a client has a request to transmit, and also has a chance to transmit (*i.e.* not subject to flow control or pipeline concurrency limit), and yet takes additional time to transmit. This is due to response processing (*i.e.* parsing) and delays in scheduling the client process. Pirate correctly identifies the causal pairs but the latter delay is not representative of response latency.



(a) All samples (single cluster) (b) Abs. error (single cluster) (c) Rel. error (single cluster) (d) Rel. error (wide area)

Figure 4: Accuracy in estimating the response latency (`req-to-res`) under realistic settings. Subfigures (a)–(c) show distributions within a single cluster; (d) shows Pirate's relative error in a wide-area network with different degrees of cross traffic.



(a) Error prevalence (b) Error magnitude

Figure 5: Classifying Pirate's errors in estimating response latency, showing prevalence (a) and magnitude (b) across connections. Discussion in §4.2 (why does Pirate make errors?)

The second kind of error, labeled `staggered-response`, occurs when a series of large response objects (each spanning multiple network RTTs) causes successive requests to be transmitted well apart in time. Pirate mistakes the pauses between the causally-unrelated successive requests to be pauses between causally-related ones. Here, Pirate misidentifies the causal pairs in the first place.

Figure 5 (a) shows the CDF of the fraction of erroneous samples per connection. The parsing-scheduling error is more prevalent. Roughly 70% of the connections have fewer than 5% erroneous samples of either kind. Figure 5 (b) shows the CDF of the average relative error of the erroneous samples per connection. The magnitude of these errors is consistent with the overall distribution (Figure 4 (c)). Further, the magnitudes of the two kinds of errors are comparable.

Wide-area setting. We run the client workload from §4.1 on a wide-area network spanning three sites (Figure 3 (b)). The network includes cross traffic contending with the measured client-server connection. The cross traffic consists of a mix of short- and long-lived TCP flows whose sizes are sampled from a distribution measured in a production network at Microsoft [50]. Flows arrive and depart throughout the lifetime of the experiment. The arrival times of new flows are sampled from a Poisson distribution that is parameterized by the average aggregate rate of the cross traffic, a value we configure as a fraction of the bottleneck link capacity of the client-to-server path. Figure 4 (d) shows the relative error of Pirate for different degrees of cross traffic shown as a % of bottleneck capacity. Pirate's error is close to 0% for more than 90% of the measurements across all configurations of cross traffic. Comparing Figure 4 (c) and Figure 4 (d), Pirate exhibits a higher accuracy in larger RTT, wide-area settings, than in smaller RTT, single-cluster settings.

Comparison Against RTT estimators. To perform a faithful comparison of Pirate against transport-layer RTT estimators (§2.2), we run a simplified web workload where the clients from §4.1 only hold a single request in flight on each connection. A majority of response objects in the workload are small enough to fit into a single packet, which increases the likelihood that transport RTTs can faithfully model response latency. We use the single cluster setup (Figure 3 (a)). All compared techniques except syn-ack and Pirate (`tcp-probe`, `tcptrace`, `req-to-req`, and `req-to-res`) have visibility into both directions of network traffic.

Figure 6 (a) compares the CDF of the latencies measured by each technique over all objects across all connections. Even with a simplified version of a realistic workload, RTT measures consistently underestimate response latency, since they are more closely aligned with the time to the first byte of the response, as opposed to the last byte. Figure 6 (b) and (c) show measurements reported over a small interval of data transfer. Syn-ack estimation produces only one measurement per connection, which does not capture the variability of latency over the lifetime of the connection.

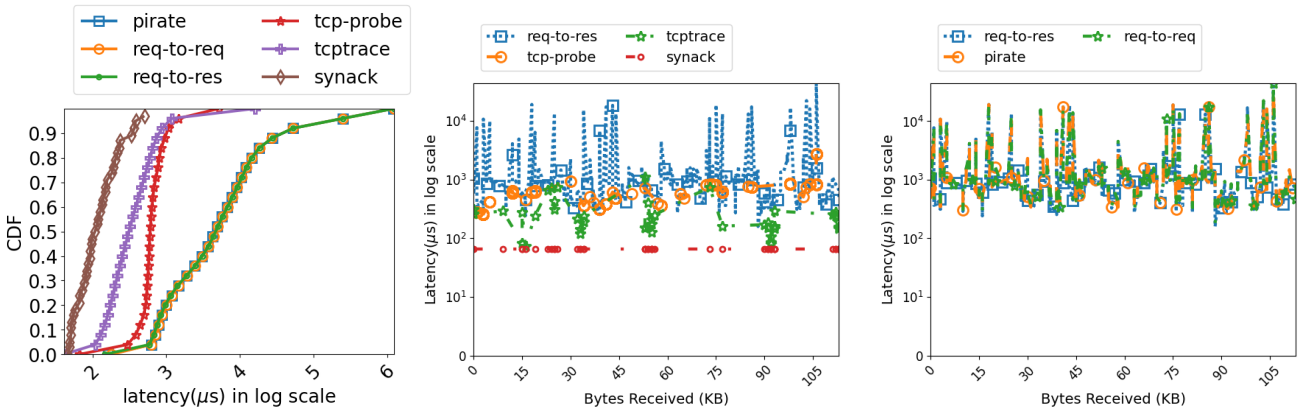


Figure 6: (a) CDF of all measurements throughout the lifetime of all connections. (b) and (c): Time evolution of the latency of a single (randomly chosen) connection during a small window of data transfer.

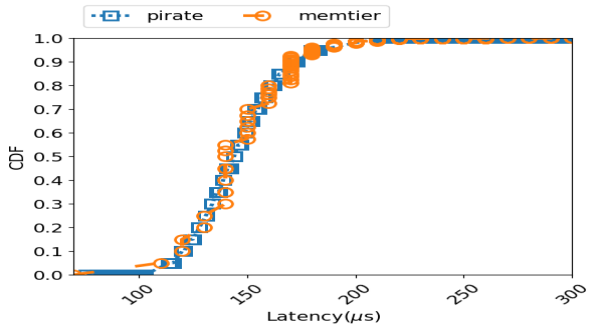


Figure 7: Latencies from memtier (ground truth) and Pirate.

Measuring API traffic from memcached. To evaluate the efficacy of Pirate for API traffic, we run a memcached benchmarking client, `memtier` [10] in the single-cluster topology (Figure 3 (a)). The client is run with 10 threads and one client connection per thread. The proportion of get and set requests is 1:1. The data size for responses is configured in the range 40–10000 bytes. The benchmark was run for 2 minutes. Figure 7 shows the CDF of the response latency measured at the application layer by `memtier` (ground truth) against estimates from Pirate. Moving beyond web workloads, Pirate may be useful to estimate response latencies for delay-sensitive applications more generally, when such applications exhibit flow control or request-level dependencies.

4.3 Software Overheads

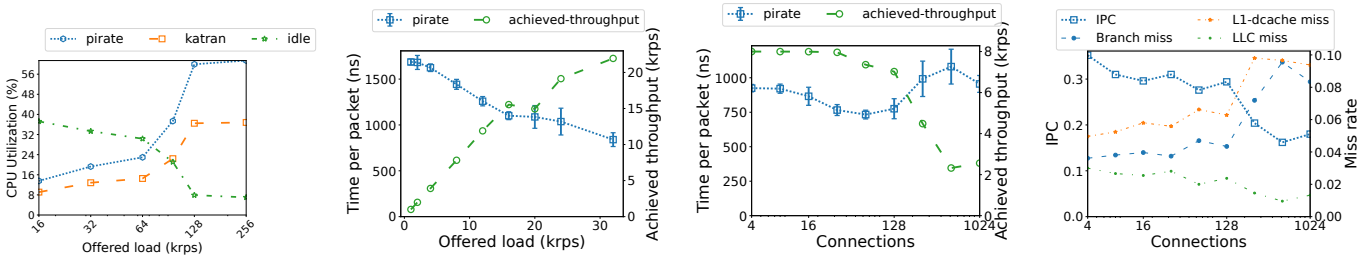
We evaluate the CPU overheads of the Pirate algorithm and its XDP implementation in the context of another XDP packet-processing application, Katran [16], a layer-4 load balancer used in production at Meta. On our single-cluster setup (Figure 3 (a)), the vantage point serves as the device under test, running Pirate and Katran chained through an eBPF tail call [36]. The client runs `wrk2`, a web benchmarking client, with 4 threads and a configurable offered load (requests/sec) and connection count. For simplicity of interpretation, we direct all requests to a single fixed CPU core using Linux Receive Side Scaling (RSS [40]).

In Figure 8 (a) and (b), we fix the number of connections to 64, and vary the offered load (Krequests/sec). Subfigure (a) reports the CPU utilization measured using `perf` events [37], in comparison to Katran and the CPU idle thread. The CPU usage of Pirate is comparable to but slightly higher than that of Katran. Figure 8 (b) shows the time per packet within Pirate across loads. To put this in context, on our experimental system, Katran incurs 1115 ns per packet, while a simple forwarding/redirect program incurs 870 ns.

In Figure 8 (c) and (d), we fix the total offered load to 8000 requests/sec, and vary the number of active client connections. Subfigure (c) shows that Pirate offers stable packet-processing times and throughput upto 128 connections, but degrades beyond this point. An investigation using system performance counters [37], reported in subfigure (d), shows that the instructions per cycle and data cache miss rates rise sharply with additional connections beyond 128.

The results above have been measured with all packet processing executed on a single CPU core. Practical systems scale single-core throughput and connections nearly linearly with additional CPU cores [40, 52, 107, 128]. Hence, the total throughput and active connections supported in a real deployment would depend on the CPU parallelism available.

XDP requires provisioning memory ahead of time for lookup data structures. By default, Katran provisions a hash



(a) CPU util. vs load (b) Latency vs load (c) Latency vs # conns. (d) `perf` vs # conns.

Figure 8: The CPU and latency overheads of PIRATE in the context of a layer-4 load balancer. See discussion in §4.3.

table for 64K connections, incurring 458 MBytes. In comparison, supporting the same number of connections in PIRATE requires 12 MBytes (154 bytes/connection).

4.4 Feedback Control

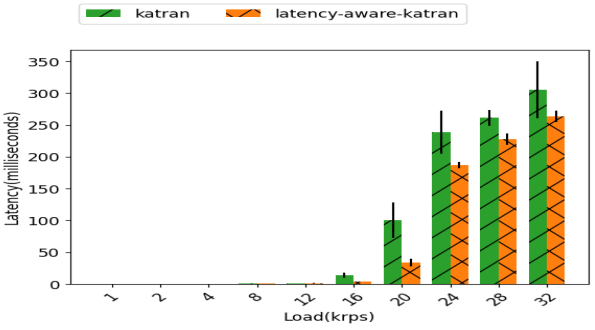


Figure 9: Comparison of 99th percentile tail latency of an unmodified Katran load balancer and our latency-aware one.

Can measurements from PIRATE enable more robust real-time feedback control? To answer this question, we evaluate the use of measurements from PIRATE within a layer-4 load balancer, Katran [16], implementing direct server return [105], using the approach discussed in §3.5. We use the star topology in Figure 3 (c), where two clients connect to four servers via a vantage point that uses the measurements to balance server load. We simplified the web workload from §4.1 to have each client connection maintain one request in flight. Figure 9 compares the 99th percentile tail response latency of the latency-aware load balancer against the vanilla version, at varying offered loads (Krequests/sec). Latency awareness produces on average a 37% reduction in tail latency across loads. Moreover, latency awareness also results in more predictable tail latencies (smaller error bars).

4.5 Benefits of Key Ideas

We evaluate the viability of the core ideas in PIRATE through microbenchmarks. Here are the key questions we are interested in: (i) Does the time gap between causally-related packets reflect the application-level response latency (§3.1)? (ii) How likely is a packet following a prominent time gap (*i.e.* a time gap "significantly" larger than other inter-packet time gaps) to be a causally-triggered request (§3.2)? (iii) Are prominent time gaps captured accurately by maintaining distributions in PIRATE (§3.3)?

We run our web benchmarking client and server (§4.1) within a single cluster (Figure 3 (a)). To simplify the experiment, we leave server performance unconstrained by interfering CPU allocations (§4.1), and collect measurements over one long-lived client-server connection. We measure: (1) the ground truth response latency (`req-to-res`), and also three successively coarse approximations of it: (2) causal pair delay (`req-to-req`, §3.1); (3) an approach that uses a threshold of 0.6 × the true response latency, to separate packets into batches of causally-related packets (§3.2), labeled `prominent-gap`; and (4) PIRATE (§3.3). We align the transport-level sequence numbers represented in the measurements across techniques, so that estimates can be compared directly against each other. In our experiments, the number of concurrent requests in flight is a key factor that impacted measurement accuracy, hence we evaluate three pipeline depths (4, 8, 16).

Figure 10 shows the CDF of relative error of each approximate method above with respect to the method before it. Subfigure (a) shows that `req-to-req` is a good overapproximation of `req-to-res`. Subfigure (b) shows that `prominent-gap` is slightly erroneous but is still a good approximation of `req-to-req`. Subfigure (c) shows that PIRATE is a similarly good approximation of `prominent-gap`. We also note the errors in the tail worsen as pipeline depth increases.

4.6 Robustness

Packet Loss and Reordering. We evaluate the impact of packet loss and reordering, both of which affect the packet

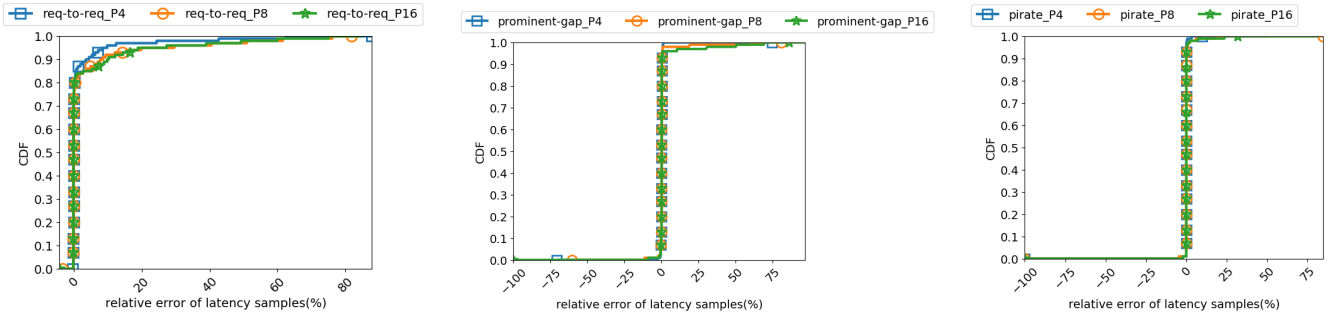


Figure 10: Viability of the approximations in Pirate. See discussion in §4.5. The graphs show the relative errors of (a) `req-to-req` over `req-to-res`; (ii) `prominent-gap` over `req-to-req`; and (iii) Pirate over `prominent-gap`, at three pipeline depths.

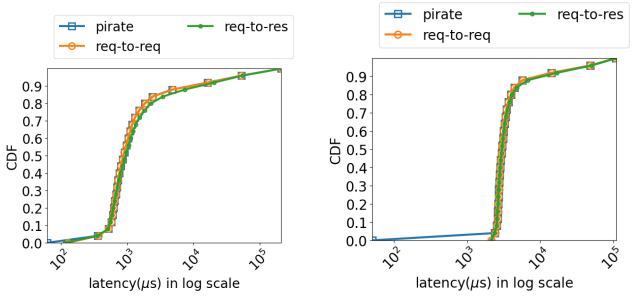


Figure 11: (a) CDF of latencies for a loss rate of 1% (b) CDF of latencies for a packet reorder rate of 25%

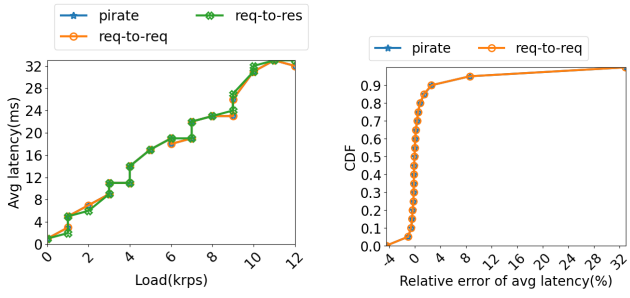


Figure 12: (a) Average response latency across all connections vs. offered load. (b) Relative error at 12K requests/sec.

inter-arrival times due to transport adaptation, on the estimation in Pirate. In our single-cluster setup (Figure 3 (a), we induce loss and reordering of varying rates over packets from the client towards the vantage point. (Loss on the server-to-client path triggers retransmission from the server to the client, which is not observed by the vantage point, hence we do not evaluate this.) Figure 11 (a) shows the CDF of latencies measured by Pirate under a loss rate of 1%, while Figure 11 (b) shows the CDF of latencies under a high packet reorder rate of 25%. Pirate produces robust estimates of response latency under both conditions. The estimation accuracies were similar under other loss and reorder rates that we evaluated.

Robustness to Offered Load. We evaluated the accuracy of latency estimation as the load offered to the server varies. In this experiment, we simplify the web workload from §4.1 to only transmit one request in flight, and varied the offered server load (requests/sec). Figure 12 (a) shows the average latency across all connections across load. At a load of 12K requests/second, Figure 12 (b) shows the CDF of relative error (averaged per connection) across the techniques. Pirate shows strong agreement with the response latency.

5 DISCUSSION

This paper presented Pirate, an algorithm that passively and continuously measures response latencies under routing asymmetry, even when transport headers are missing due to encryption or fragmentation. Pirate leverages the idea of causal pairs, two requests the second of which is triggered by the response to the first. In realistic experiments, Pirate shows high accuracy and enables robust feedback control.

Pirate has two significant limitations, both stemming from the estimator's assumptions. First, applications with significant client think times suffer from sizable overestimations of response latency (*e.g.* DASH video streams). Second, the estimator assumes that successive batches of packets are causally related. However, large response objects (§4.2), and the presence of arbitrary reordering in server responses (relative to requests) both violate this assumption. The latter challenges generalizing our results to HTTP/2 and QUIC.

Despite these limitations, Pirate is robust for HTTP/1.1-like web and fast-growing API traffic [25, 28], in the presence of multiple pipelined concurrent requests, server fluctuations, and network variability. These results may have broader implications to the design of reactive feedback control systems relying on passive continuous measurement.

REFERENCES

[1] 1999. HTTP/1.1 Persistent Connections (RFC 2616). [Online, Retrieved Jan 26, 2025.] https://www.rfc-editor.org/rfc/rfc2616#section-8.1. (1999).

[2] 2001. Network.http.max-connections-per-server. [Online, Retrieved Jan 31, 2026.] https://kb.mozillazine.org/Network.http.max-connections-per-server. (2001).

[3] 2004. Web Polygraph. [Online, Retrieved Jan 26, 2025.] https://www.web-polygraph.org/. (2004).

[4] 2005. IP Encapsulating Security Payload (ESP). [Online, Retrieved Jan 31, 2026.] https://www.rfc-editor.org/rfc/rfc4303. (2005).

[5] 2006. Marissa Mayer at Web 2.0. [Online, Retrieved Jan 26, 2025.] https://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html. (2006).

[6] 2008. Raising network.http.max-persistent-connections-per-server? [Online, Retrieved Jan 31, 2026.] https://groups.google.com/g/mozilla.dev.apps.firefox/c/Xuu4xlw0w8Y?pli=1. (2008).

[7] 2009. The cost of latency. [Online, Retrieved Jan 26, 2025.] https://perspectives.mvdirona.com/2009/10/the-cost-of-latency/. (2009).

[8] 2011. Computing TCP's retransmission timer. [Online, Retrieved Jan 26, 2025.] https://www.rfc-editor.org/rfc/rfc6298. (2011).

[9] 2011. Firefox 200 websocket connections limit. [Online, Retrieved Jan 31, 2026.] https://hg-edge.mozilla.org/mozilla-central/rev/43f88c89f4cb. (2011).

[10] 2013. memtier_benchmark: A High-Throughput Benchmarking Tool for Redis and Memcached. [Online, Retrieved Jan 26, 2025.] https://redis.io/blog/memtier_benchmark-a-high-throughput-benchmarking-tool-for-redis-memcached/. (2013).

[11] 2013. Someone's been siphoning data through a huge security hole in the Internet. [Online, Retrieved Jan 26, 2025.] https://www.wired.com/2013/12/bgp-hijacking-belarus-iceland/. (2013).

[12] 2015. HTTP/2 Streams and Multiplexing (RFC 7540). [Online, Retrieved Jan 26, 2025.] https://www.rfc-editor.org/rfc/rfc7540#section-5. (2015).

[13] 2015. RFC 7540 HTTP/2: Streams and Multiplexing. [Online, Retrieved Jun 12, 2022.] https://www.rfc-editor.org/rfc/rfc7540.html#section-5. (2015).

[14] 2017. Mobile site speed playbook. [Online, Retrieved Jan 26, 2025.] https://www.thinkwithgoogle.com/_qs/documents/4290/c676a_Google_MobileSiteSpeed_Playbook_v2.1_digital_4JWkGQT.pdf. (2017).

[15] 2018. Chrome defaults to 6 connections per host. [Online, Retrieved Jan 31, 2026.] https://chromium.googlesource.com/chromium/src/net/%2B/master/socket/client_socket_pool_manager.cc#45. (2018).

[16] 2018. Open-sourcing Katran, a scalable network load balancer. [Online, Retrieved Jan 26, 2025.] https://engineering.fb.com/2018/05/22/open-source/open-sourcing-katran-a-scalable-network-load-balancer/. (2018).

[17] 2018. The Latency Spin Bit: draft-trammell-quic-spin-01. [Online, Retrieved Jan 26, 2025.] https://www.ietf.org/proceedings/101/slides/slides-101-quic-the-latency-spin-bit-00.pdf. (2018).

[18] 2019. Google cluster data 2019. [Online, Retrieved Jan 26, 2025.] https://github.com/google/cluster-data/blob/master/ClusterData2019.md. (2019).

[19] 2020. Milliseconds make millions. [Online, Retrieved Jan 26, 2025.] https://www.deloitte.com/ie/en/services/consulting/research/milliseconds-make-millions.html. (2020).

[20] 2020. RFC 9000: QUIC: flow control. [Online, Retrieved Jun 12, 2022.] https://www.rfc-editor.org/rfc/rfc9000.html#flow-control. (2020).

[21] 2021. QUIC: Generating acknowledgments. [Online, Retrieved Jan 31, 2026.] https://www.rfc-editor.org/rfc/rfc9000.html#name-generating-acknowledgments. (2021).

[22] 2021. QUIC probe timeout replaces RTO and TLP. [Online, Retrieved Jan 26, 2025.] https://quicwg.org/base-drafts/rfc9002.html#name-probe-timeout-replaces-rto-. (2021).

[23] 2021. When BGP Routes Accidentally Get Hijacked: A Lesson in Internet Vulnerability. [Online, Retrieved Jan 26, 2025.] https://www.thousandeyes.com/blog/internet-report-episode-44. (2021).

[24] 2022. HTTP/3 stream mapping and usage (RFC 9114). [Online, Retrieved Jan 26, 2025.] https://www.rfc-editor.org/rfc/rfc9114.html#name-stream-mapping-and-usage. (2022).

[25] 2022. Landscape of API Traffic. [Online, Retrieved Jan 31, 2026.] https://blog.cloudflare.com/landscape-of-api-traffic/. (2022).

[26] 2023. 21 SSL Statistics that Show Why Security Matters so Much. [Online, Retrieved Jan 26, 2025.] https://webtribunal.net/blog/ssl-stats. (2023).

[27] 2023. Are you measuring what matters? A fresh look at Time To First Byte. [Online, Retrieved Jan 26, 2025.] https://blog.cloudflare.com/ttfb-is-not-what-it-used-to-be/. (2023).

[28] 2023. Examining HTTP/3 usage one year on: API request distribution by HTTP version. [Online, Retrieved Jan 31, 2026.] https://blog.cloudflare.com/http3-usage-one-year-on/#api-request-distribution-by-http-version. (2023).

[29] 2023. Katran example setup. [Online, Retrieved Jan 26, 2025.] https://github.com/facebookincubator/katran/blob/main/EXAMPLE.md. (2023).

[30] 2024. Connection concurrency (Performance Best Practices with gRPC). [Online, Retrieved Jan 26, 2025.] https://learn.microsoft.com/en-us/aspnet/core/grpc/performance?view=aspnetcore-9.0#connection-concurrency. (2024).

[31] 2024. HTTP Load Balancing. [Online, Retrieved Jan 26, 2025.] https://docs.nginx.com/nginx/admin-guide/load-balancer/http-load-balancer/. (2024).

[32] 2024. Performance best practices with gRPC. [Online, Retrieved Jan 26, 2025.] https://learn.microsoft.com/en-us/aspnet/core/grpc/performance?view=aspnetcore-9.0. (2024).

[33] 2024. tcp_probe Linux kernel tracepoint. [Online, Retrieved Jan 26, 2025.] https://elixir.bootlin.com/linux/v6.8-rc3/source/include/trace/events/tcp.h#L238. (2024).

[34] 2025. Azure network round-trip latency statistics. [Online, Retrieved Apr 25, 2025.] https://learn.microsoft.com/en-us/azure/networking/azure-network-latency?tabs=Americas%2CWestUS. (2025).

[35] 2025. Core Web Vitals (CWV). [Online, Retrieved Jan 26, 2025.] https://www.cloudflare.com/en-gb/learning/performance/what-are-core-web-vitals/. (2025).

[36] 2025. eBPF docs: Tail calls. [Online, Retrieved Jan 31, 2026.] https://docs.ebpf.io/linux/concepts/tail-calls/. (2025).

[37] 2025. Getting started with perf. [Online, Retrieved Sep 18, 2025.] https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/10/html/monitoring_and_managing_system_status_and_performance/getting-started-with-perf. (2025).

[38] 2025. Largest Contentful Paint (LCP). [Online, Retrieved Jan 26, 2025.] https://web.dev/articles/lcp. (2025).

[39] 2025. Pagespeed insights. [Online, Retrieved Jan 26, 2025.] https://pagespeed.web.dev/. (2025).

[40] 2025. Scaling in the Linux networking stack. [Online, Retrieved Nov 6, 2025.] https://www.kernel.org/doc/html/latest/networking/scaling.html. (2025).

[41] 2025. The Top 25 VPN Statistics, Facts & Trends for 2025. [Online, Retrieved Jan 26, 2025.] https://www.cloudwards.net/vpn-statistics/. (2025).

[42] 2025. ThousandEyes: How to Set Up the Virtual Appliance. [Online, Retrieved Jan 26, 2025.] https://docs.thousandeyes.com/product-documentation/global-vantage-points/enterprise-agents/installing/appliances/how-to-set-up-the-virtual-appliance. (2025).

[43] 2025. WireGuard: Fast, modern, secure VPN tunnel. [Online, Retrieved Jan 31, 2026.] https://www.wireguard.com/. (2025).

[44] 2026. Cilium mutual authentication. [Online, Retrieved Jan 31, 2026.] https://docs.cilium.io/en/latest/network/servicemesh/mutual-authentication/mutual-authentication/. (2026).

[45] 2026. Linkerd: Automatic mTLS. [Online, Retrieved Jan 31, 2026.] https://linkerd.io/2-edge/features/automatic-mtls/. (2026).

[46] 2026. Mutual TLS Migration. [Online, Retrieved Jan 31, 2026.] https://istio.io/latest/docs/tasks/security/authentication/mtls-migration/. (2026).

[47] 2026. TCP segmentation offload. [Online, Retrieved Jan 31, 2026.] https://docs.kernel.org/networking/segmentation-offloads.html. (2026).

[48] Jay Aikat, Jasleen Kaur, F Donelson Smith, and Kevin Jeffay. 2003. Variability in TCP round-trip times. In *ACM Internet Measurement Conference (IMC)*.

[49] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, et al. 2014. CONGA: Distributed congestion-aware load balancing for datacenters. In *ACM SIGCOMM*.

[50] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data center TCP (DCTCP). In *ACM SIGCOMM*.

[51] Axel Arnbak and Sharon Goldberg. 2015. Loopholes for Circumventing the Constitution: Unrestrained Bulk Surveillance on Americans by Collecting Network Traffic Abroad. *Michigan Telecommunications and Technology Law Review* (2015).

[52] Tom Barbette, Georgios P. Katsikas, Gerald Q. Maguire, and Dejan Kostić. 2019. RSS++: Load and State-Aware Receive Side Scaling. In *Conference on Emerging Networking Experiments And Technologies (CoNEXT)*.

[53] 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)*.

[54] Paul Barford and Mark Crovella. 1999. Measuring web performance in the wide area. *ACM SIGMETRICS Performance Evaluation Review* (1999).

[55] Enrico Bocchi, Luca De Cicco, and Dario Rossi. 2016. Measuring the quality of experience of web users. *ACM SIGCOMM Computer Communication Review (CCR)* (2016).

[56] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. 2021. Understanding host network stack overheads. In *ACM SIGCOMM*.

[57] Christopher Canel, Balasubramanian Madhavan, Srikanth Sundaresan, Neil Spring, Prashanth Kannan, Ying Zhang, Kevin Lin, and Srinivasan Seshan. 2024. Understanding Incast Bursts in Modern Datacenters. In *ACM Internet Measurement Conference (IMC)*.

[58] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2016. BBR: Congestion-Based Congestion Control. *ACM Queue* (2016).

[59] Damiano Carra, Konstantin Avrachenkov, Sara Alouf, Alberto Blanc, Philippe Nain, and Georg Post. 2010. Passive online RTT estimation for flow-aware routers using one-way traffic. In *NETWORKING*. Springer.

[60] David Choffnes and Fabián E Bustamante. 2009. On the effectiveness of measurement reuse for performance-based detouring. In *INFOCOM 2009*.

[61] Baek-Young Choi, Sue Moon, Zhi-Li Zhang, Konstantina Papagiannaki, and Christophe Diot. 2007. Analysis of point-to-point packet delay in an operational network. *Computer networks* (2007).

[62] Piet De Vaere, Tobias Bühler, Mirja Kühlewind, and Brian Trammell. 2018. Three bits suffice: Explicit support for passive measurement of internet latency in quic and tcp. In *Internet Measurement Conference (IMC)*.

[63] Amogh Dhamdhere, David D Clark, Alexander Gamero-Garrido, Matthew Luckie, Ricky KP Mok, Gautam Akiwate, Kabir Gogia, Vaibhav Bajpai, Alex C Snoeren, and Kc Claffy. 2018. Inferring persistent interdomain congestion. In *ACM SIGCOMM*.

[64] Hao Ding and Michael Rabinovich. 2015. TCP stretch acknowledgements and timestamps: findings and implications for passive RTT measurement. *ACM SIGCOMM Computer Communication Review* (2015).

[65] C. Dovrolis, P. Ramanathan, and D. Moore. 2001. What do packet dispersion techniques measure?. In *IEEE INFOCOM*.

[66] 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 *USENIX Annual Technical Conference (ATC)*.

[67] 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)*.

[68] Margarida Ferreira, Ranysha Ware, Yash Kothari, Inês Lynce, Ruben Martins, Akshay Narayan, and Justine Sherry. 2024. Reverse-Engineering Congestion Control Algorithm Behavior. In *ACM Internet Measurement Conference (IMC)*.

[69] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. 2018. Azure accelerated networking:{SmartNICs} in the public cloud. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[70] Tobias Flach, Pavlos Papageorge, Andreas Terzis, Luis Pedrosa, Yuchung Cheng, Tayeb Karim, Ethan Katz-Bassett, and Ramesh Govindan. 2016. An Internet-Wide Analysis of Traffic Policing. In *ACM SIGCOMM*.

[71] 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 *Conference on architectural support for programming languages and operating systems (ASPLOS)*.

[72] Rohan Gandhi and Srinivas Narayana. 2025. KnapsackLB: Enabling Performance-Aware Layer-4 Load Balancing. In *ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT)*.

[73] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. 2018. Exploiting a natural network effect for scalable, fine-grained clock synchronization. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[74] Ehab Ghabashneh, Yimeng Zhao, Cristian Lumezanu, Neil Spring, Srikanth Sundaresan, and Sanjay Rao. 2022. A microscopic view of bursts, buffer contention, and loss in data centers. In *ACM Internet Measurement Conference (IMC)*.

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

[76] Prateesh Goyal, Akshay Narayan, Frank Cangialosi, Srinivas Narayana, Mohammad Alizadeh, and Hari Balakrishnan. 2022. Elasticity detection: A building block for internet congestion control. In *ACM SIGCOMM*.

[77] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, et al. 2015. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *ACM SIGCOMM*.

[78] Toke Høiland-Jørgensen, Bengt Ahlgren, Per Hurtig, and Anna Brunstrom. 2016. Measuring latency variation in the internet. In *ACM CoNEXT*.

[79] Alexis Huet, Antoine Saverimoutou, Zied Ben Houidi, Hao Shi, Shengming Cai, Jinchun Xu, Bertrand Mathieu, and Dario Rossi. 2020. Revealing QoE of Web Users from Encrypted Network Traffic. In *IFIP Networking Conference (Networking)*.

[80] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. 2018. The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel. In *Conference on emerging Networking EXperiments and Technologies (CoNEXT)*.

[81] Sharad Jaiswal, Gianluca Iannaccone, Christophe Diot, Jim Kurose, and Don Towsley. 2004. Inferring TCP connection characteristics through passive measurements. In *INFOCOM 2004*.

[82] Hao Jiang and Constantinos Dovrolis. 2002. Passive estimation of TCP round-trip times. *ACM SIGCOMM Computer Communication Review (CCR)* (2002).

[83] Anuj Kalia, Michael Kaminsky, and David Andersen. 2019. Datacenter {RPCs} can be general and fast. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[84] Srikanth Kandula, Dina Katabi, Shantanu Sinha, and Arthur Berger. 2007. Dynamic load balancing without packet reordering. *SIGCOMM Computer Communication Review (CCR)* (2007).

[85] Partha Kanuparthy and Constantine Dovrolis. 2011. ShaperProbe: end-to-end detection of ISP traffic shaping using active methods. In *ACM Internet Measurement Conference (IMC)*.

[86] Ethan Katz-Bassett, Harsha V Madhyastha, Vijay Kumar Adhikari, Colin Scott, Justine Sherry, Peter Van Wesep, Thomas E Anderson, and Arvind Krishnamurthy. 2010. Reverse traceroute.. In *Usenix Symposium on Networked Systems Design and Implementation (NSDI)*.

[87] Frank Kelly, Gaurav Raina, and Thomas Voice. 2008. Stability and fairness of explicit congestion control with small buffers. *ACM SIGCOMM Computer Communication Review* (2008).

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

[89] Ron Kohavi, Randal M Henne, and Dan Sommerfield. 2007. Practical guide to controlled experiments on the web: listen to your customers not to the hippo. In *ACM SIGKDD international conference on Knowledge discovery and data mining*.

[90] Ramana Rao Kompella, Kirill Levchenko, Alex C Snoeren, and George Varghese. 2009. Every microsecond counts: tracking fine-grain latencies with a lossy difference aggregator. In *ACM SIGCOMM*.

[91] Rupa Krishnan, Harsha V Madhyastha, Sridhar Srinivasan, Sushant Jain, Arvind Krishnamurthy, Thomas Anderson, and Jie Gao. 2009. Moving beyond end-to-end path information to optimize CDN performance. In *ACM Internet Measurement Conference (IMC)*.

[92] Ike Kunze, Constantin Sander, and Klaus Wehrle. 2023. Does It Spin? On the Adoption and Use of QUIC's Spin Bit. In *Proceedings of the 2023 ACM on Internet Measurement Conference*. 554–560.

[93] Myungjin Lee, Nick Duffield, and Ramana Rao Kompella. 2010. Not all microseconds are equal: Fine-grained per-flow measurements with reference latency interpolation. In *Proceedings of the ACM SIGCOMM 2010 conference*. 27–38.

[94] Jialin Li, Naveen Kr Sharma, Dan RK Ports, and Steven D Gribble. 2014. Tales of the tail: Hardware, os, and application-level sources of tail latency. In *ACM Symposium on Cloud Computing (SOCC)*.

[95] Shihan Lin, Yi Zhou, Xiao Zhang, Todd Arnold, Ramesh Govindan, and Xiaowei Yang. 2025. Latency-Aware Inter-domain Routing. (2025). arXiv:2410.13019 https://arxiv.org/abs/2410.13019

[96] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: Improving resource efficiency at scale. In *International Symposium on Computer Architecture (ISCA)*.

[97] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. 2021. Characterizing Microservice Dependency and Performance: Alibaba Trace Analysis. In *ACM Symposium on Cloud Computing (SoCC)*.

[98] Harsha V Madhyastha, Thomas Anderson, Arvind Krishnamurthy, Neil Spring, and Arun Venkataramani. 2006. A structural approach to latency prediction. In *ACM Internet Measurement Conference (IMC)*.

[99] M. Mitzenmacher. 2001. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems* (2001).

[100] Michael Mitzenmacher, Thomas Steinke, and Justin Thaler. 2012. Hierarchical heavy hitters with the space saving algorithm. In *SIAM Workshop on Algorithm Engineering and Experiments (ALENEX)*.

[101] Ravi Netravali, Ameesh Goyal, James Mickens, and Hari Balakrishnan. 2016. Polaris: Faster Page Loads Using Fine-grained Dependency Tracking. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[102] TS Eugene Ng and Hui Zhang. 2002. Predicting Internet network distance with coordinates-based approaches. In *IEEE Infocom*.

[103] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. 2013. Scaling memcache at facebook. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[104] Palo Alto. 2025. VM-Series Deployment Guide. [Online, Retrieved Jan 26, 2025.] https://docs.paloaltonetworks.com/vm-series/10-2/vm-series-deployment/about-the-vm-series-firewall. (2025).

[105] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, et al. 2013. Ananta: Cloud scale load balancing. *ACM SIGCOMM*.

[106] Vern Paxson. 1997. *Measurements and Analysis of End-to-End Internet Dynamics*. Ph.D. Dissertation. University of California, Berkeley.

[107] Francisco Pereira, Fernando M.V. Ramos, and Luis Pedrosa. 2024. Automatic Parallelization of Software Network Functions. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[108] R. Prasad, C. Dovrolis, M. Murray, and K. Claffy. 2003. Bandwidth estimation: metrics, measurement techniques, and tools. *IEEE Network* (2003).

[109] Gaurav Raina, Don Towsley, and Damon Wischik. 2005. Part II: Control theory for buffer sizing. *ACM SIGCOMM Computer Communication Review* (2005).

[110] Ahmed Saeed, Nandita Dukkipati, Vytautas Valancius, Vinh The Lam, Carlo Contavalli, and Amin Vahdat. 2017. Carousel: Scalable traffic shaping at end hosts. In *ACM SIGCOMM*.

[111] Satadal Sengupta, Hyojoon Kim, and Jennifer Rexford. 2022. Continuous in-network round-trip time monitoring. In *ACM SIGCOMM*.

[112] Shawn Ostermann. [n. d.]. tcptrace. [Online, Retrieved Jan 26, 2025.] https://www.tcptrace.org/index.shtml. ([n. d.]).

[113] 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*.

[114] Benjamin H Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. Dapper, a large-scale distributed systems tracing infrastructure. (2010).
[115] Sanjay Sinha, Srikanth Kandula, and Dina Katabi. 2004. Harnessing TCP's Burstiness with Flowlet Switching. In *ACM Workshop on Hot Topics in Networks (HotNets)*.
[116] Joel Sommers, Paul Barford, Nick Duffield, and Amos Ron. 2007. Accurate and efficient SLA compliance monitoring. In *ACM SIGCOMM*.
[117] Srikanth Sundaresan, Mark Allman, Amogh Dhamdhere, and Kc Claffy. 2017. TCP congestion signatures. In *ACM Internet Measurement Conference (IMC)*.
[118] 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)*.
[119] Michal Szymaniak, David Presotto, Guillaume Pierre, and Maarten van Steen. 2008. Practical large-scale latency estimation. *Computer Networks* (2008).
[120] Steve Uhlig and Olivier Bonaventure. 2004. Designing BGP-based outbound traffic engineering techniques for stub ASes. *ACM SIGCOMM Computer Communication Review* 34, 5 (2004), 89–106.
[121] Erico Vanini, Rong Pan, Mohammad Alizadeh, Parvin Taheri, and Tom Edsall. 2017. Let It Flow: Resilient Asymmetric Load Balancing with Flowlet Switching. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.
[122] Vijay Vasudevan, Amar Phanishayee, Hiral Shah, Elie Krevat, David G Andersen, Gregory R Ganger, Garth A Gibson, and Brian Mueller. 2009. Safe and effective fine-grained TCP retransmissions for datacenter communication. *ACM SIGCOMM computer communication review* (2009).
[123] Bryan Veal, Kang Li, and David Lowenthal. 2005. New methods for passive estimation of TCP round-trip times. In *Passive and Active Network Measurement (PAM)*.
[124] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *European Conference on Computer Systems (EuroSys)*.
[125] Nikolas Wehner, Michael Seufert, Joshua Schuler, Sarah Wassermann, Pedro Casas, and Tobias Hossfeld. 2021. Improving Web QoE Monitoring for Encrypted Network Traffic through Time Series Modeling. *SIGMETRICS Perform. Eval. Rev.* (2021).
[126] Damon Wischik and Nick McKeown. 2005. Part I: Buffer sizes for core routers. *ACM SIGCOMM Computer Communication Review* (2005).
[127] Bartek Wydrowski, Robert Kleinberg, Stephen M. Rumble, and Aaron Archer. 2024. Load is not what you should balance: Introducing Prequal. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.
[128] Qiongwen Xu, Sebastiano Miano, Xiangyu Gao, Tao Wang, Adithya Murugadass, Songyuan Zhang, Anirudh Sivaraman, Gianni Antichi, and Srinivas Narayana. 2025. State-compute replication: parallelizing high-speed stateful packet processing. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.
[129] Curtis Yu, Cristian Lumezanu, Abhishek Sharma, Qiang Xu, Guofei Jiang, and Harsha V Madhyastha. 2015. Software-defined latency monitoring in data center networks. In *Passive and Active Measurement*.
[130] Minlan Yu, Albert Greenberg, Dave Maltz, Jennifer Rexford, Lihua Yuan, Srikanth Kandula, and Changhoon Kim. 2011. Profiling network performance for multi-tier data center applications. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.
[131] Jiao Zhang, F Richard Yu, Shuo Wang, Tao Huang, Zengyi Liu, and Yunjie Liu. 2018. Load balancing in data center networks: A survey. *IEEE Communications Surveys & Tutorials* (2018).
[132] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y Zhao, et al. 2015. Packet-level telemetry in large datacenter networks. In *ACM SIGCOMM*.

A AN ESTIMATOR BASED ON PROMINENT PACKET GAPS

See Algorithm 1.

Algorithm 1 Track causally-triggered requests using a fixed time threshold δ at the vantage point. The algorithm is executed upon receiving each packet of a flow f.

Require: Fixed threshold on inter-packet gaps, δ
Require: Timestamp of the current packet's arrival, now
Require: The last time a new batch arrived for flow f, $f.time\_last\_batch$
Require: The last time a packet arrived for flow f, $f.time\_last\_pkt$
Ensure: An estimate of flow f's reponse latency, $\hat{T}_{LB}$, if a new sample is produced, else $undef$

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

B MAINTAINING EFFICIENT HISTOGRAMS

See Algorithm 2.

C DETAILED DESIGN OF THE LATENCY-AWARE LOAD BALANCER

Below, we present the details of the design of our latency-aware layer-4 load balancer, which builds on a Maglev-like load balancer [67] which permits the setting of weights to servers. The load balancer should measure response latency for a number of connections mapped to each server using the algorithms in §3, to produce a representative average latency for each server. We then use these average server latencies to adapt the weights assigned to the servers, using three key ideas.

First, we take away weights from servers which have average latencies larger than a high watermark, and place them on servers which have average latencies smaller than a low

Algorithm 2 Maintaining a small number of modes N from the empirical probability distribution of IPGs.

Require: New observation of an IPG g
Require: A representation of the empirical probability distribution, M, with N modes. For each $1 \leq i \leq N$, the i^{th} mode is a tuple $(min, max, count, sum)$, denoting the minimum, maximum, count, and sum of all the IPGs observed within the mode.
Ensure: M is updated with the additional IPG g

```
function UpdateModes(IPG g)
    for m ∈ M.get_modes() : do
                    ▷ Modes traversed in ascending order
        left = m.get_min()
        right = m.get_max()
        if left − ϵ ≤ g ≤ right + ϵ then
                ▷ IPG g lies within or proximal to mode m
            AddGapToMode(g, m)
            if left − ϵ ≤ g ≤ left then
                                ▷ g is proximal from below
                m.set_min(g)
                ConsiderMergeModes(m.get_prev(), m)
            else if right ≤ g ≤ right + ϵ then
                                ▷ g is proximal from above
                m.set_max(g)
                ConsiderMergeModes(m, m.get_next())
            end if
            return
        end if
    end for
    if M has fewer than N initialized modes then
                    ▷ Insert singleton mode containing g
        AddMode(M, g)
    else
        discarded += 1
    end if
end function
```

watermark. Our high (respectively, low) latency watermark is defined as α_{high} (respectively, α_{low}) times the latency of the server with the smallest average latency. We use $\alpha_{high} = 1.5$ and $\alpha_{low} = 1.2$. Moreover, the weight shifted from a high-latency server is proportional to its latency.

Second, we restrict the low-latency servers eligible for placing additional weight by limiting ourselves to servers that have sufficient *freshness* in their latency measurement. As prior work has observed [127], measured latencies are a good metric for the past performance of backend servers, but not their future. Instead, a different metric such as the number of requests in flight to a server is a leading indicator for the future performance of that server. Consequently, some prior works consider a combination of requests in flight and latency to balance load [118, 127]. A DSR load balancer cannot directly measure the number of requests in flight.

Instead, we measure the recency of the latency measurement of servers, by defining the freshness of a server latency measurement to be the ratio of the total number of requests received in the last measurement interval to the number of concurrent active connections at the end of that interval. This metric captures the intuition that a server on the verge of slowing down will have processed fewer requests per active connection. Conversely, a fast server must have processed more requests in the last measurement interval even if several of those connections have arrived and completed. We only consider a low-latency server eligible to take on additional weight if its freshness is at least as high as any high-latency server. Each such server receives an equal share of the total weight that is shifted away from the high-latency servers, subject to a cap on the per-server increment in weight, to avoid the thundering herd problem [103].

The third key idea is to regress to the mean: if latencies have not changed in the recent k measurement intervals, the weights are slowly equalized across servers (we use $k = 3$). Server and network performance can be highly variable. It is faster to improve performance from an operating point where no one server is assigned a disproportionately large weight.

D HEURISTICS IN PIRATE

To further improve the accuracy of Algorithm 2, we use two heuristics that eliminate IPG modes that we deem noisy. First, we coalesce IPGs corresponding to pure transport-layer acknowledgments into one IPG, by ignoring them from the stream of IPGs observed for a connection. The intuition is that pure ACKs do not represent the completion of an application-layer response, but rather signal partial completion. Mechanically, when transport headers are unencrypted, it is easy to identify pure ACK packets by inspecting the transport-layer headers, for example, the TCP ACK flag and packet size. When the transport layer is encrypted, this heuristic only applies if there is information that helps classify a packet as a pure ACK, *e.g.* payload sizes. Our second heuristic is to explicitly mark IPGs following a non-MTU packet as candidate modes to represent the IBG used in the proportional mode sum computation. The intuition is that clients typically send full MTU packets whenever packets can be transmitted independently and back to back.

Using our setup described in §4.1, we evaluate the benefits of the ACK-coalescing and MTU size heuristics (§3.4) in improving the accuracy of Pirate's estimation. Figure 13 (a) compares the estimated response latencies when turning off both heuristics, against Pirate. In (b), the time series of request-to-triggered-request delays is shown against Pirate

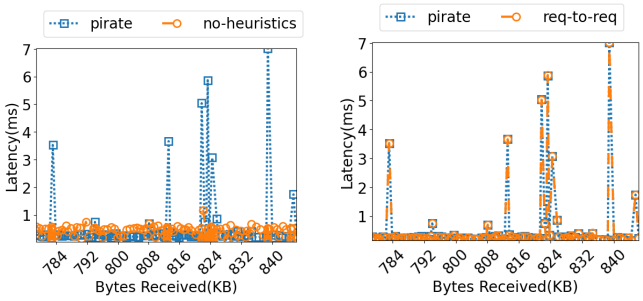


Figure 13: (a) Pirate *vs* Pirate without both heuristics. (b) Pirate *vs* the request-to-triggered request delay.

for reference. The heuristics provide a noticeable improvement to accuracy, especially in allowing Pirate to track fast fluctuations in latency.