Measuring Round-Trip Response Latencies Under Asymmetric Routing

Bhavana Vannarth Shobhana[1], *Yen-lin Chien*[1], *Jonathan Diamant*[2],
Badri Nath[1], *Shir Landau Feibish*[2], *Srinivas Narayana*[1]
[1] *Rutgers University, USA* [2] *The Open University of Israel, Israel*

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 uptake of encrypted transports (such as QUIC) 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 encrypted. 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 estimate the response latencies measured at the client application layer to within 1%. 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 [3, 4, 10, 14, 74]. 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 [76, 103]. 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 [47, 49, 100, 101] or interdomain route hijack [7, 18, 38], and take corrective actions to fix routing configurations or provision capacity [80]. Within a data center, server latency may be used to implement performance-optimized replica selection in load balancers [39, 57] or remote procedure call (RPC) clients [102, 111].

Continuous measurement of client-visible latency is crucial since the latency may change over the lifetime of a connection [36, 40, 50, 63]. For example, time-varying bursts and packet losses in the network [43, 59], or server variability due to noisy neighbors, load, and resource scheduling [40, 42, 56, 81], 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 [62]. *Passive approaches* observe latency at strategic locations that can be controlled [103, 113, 115], 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 [103]. If passive measurement is possible, it can observe real and representative client connections [40], avoiding the typical downsides of active approaches that incur compute and network resources for probing [46, 75].

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 [68, 79]. 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 [21, 29, 30, 41]. 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 encrypted transport and application headers;
2. handle routing asymmetry;
3. generalize across transports and transport algorithms; and
4. support efficient deployment on software middleboxes.

At first glance, it may appear that the response latency is closely related to the transport-layer round-trip time (RTT), whose passive measurement is widely studied, *e.g.* [45, 50, 60, 66, 67, 95, 96, 107]. 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 [86], RPC [23, 56, 82, 98]) and flow control (*e.g.* key-value store [88], web [9, 15]).

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, 8, 19]. 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), transports (*e.g.* TCP *vs* QUIC), 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 key ideas above, PIRATE. We show how to use measurements from PIRATE in a feedback control loop (§4), by adapting a layer-4 load balancer to leverage real-time response latencies. This design unilaterally changes the load balancer, leaving clients, servers, application software, and the network unmodified. The load balancer may use direct server return [90] to avoid processing the responses.

Results. In §5, we evaluate PIRATE under a realistic web workload derived from the Alexa top-100 web sites, with a web server whose CPU availability varies follows a real CPU utilization trace [108]. Across all monitored responses, PIRATE achieves a median relative error of 0.63% relative to the response latency measured at the client application. Our results affirm the hypothesis that the proxy request-to-request intervals estimated by PIRATE are sound approximations of client-application-visible response latencies. We have localized the causes of errors in PIRATE to two main reasons: client-side processing delays and staggered flights of response packets transmitted by servers. Our results further show that transport-layer RTT measurement cannot model the application-layer response latency accurately, even when such measurement can take advantage of bidirectional visibility into traffic (PIRATE only sees client-to-server traffic).

We integrated PIRATE into Katran [11], an open-source layer-4 load balancer based on Maglev hashing [53]. Latency-aware Katran cuts the 99th percentile latency in our experimental setup by 37% on average across loads, relative to (unmodified) Katran. Latency-aware Katran also shows smaller variability in tail latencies. The accuracy of PIRATE is robust to packet loss and reordering in the network. We also show that PIRATE accurately estimates response latencies over an encrypted transport (mvfst QUIC [31]). PIRATE imposes an average delay of 346 ns in the critical packet-processing path when forwarding packets using XDP [65], a fast packet-processing platform. For context, the Katran load balancer takes on average 1114 ns to process each packet.

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.

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. The discussions below apply either when the client is 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, but 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 encrypted transport headers. Encryption of application-layer payloads is ubiquitous on the web and in compute clusters [20]. Modern transport protocols and network-layer security go one level further, obscuring transport-layer headers. Deployments of HTTP/3 over QUIC [16,24] and network-layer encrypted tunnels [34], both of which encrypt transport headers, are trending upward. We desire a method that does not require visibility into transport layer headers.

G2. Handle 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 [91] and also in data centers [114]. Further, many network deployments use asymmetric routing *by design*. For example, layer-4 load balancers employ direct server return [53, 90]—bypassing the load balancer for response packets—to avoid redundant packet processing at the load balancer. BGP policies on the Internet (*e.g.* stub networks) may choose different ingress and egress border routers for a given connection [71, 104].

G3. Generalize across transports and transport algorithms. The design of new TCP and QUIC algorithms (*e.g.* congestion control, loss recovery) is an active and rapidly evolving area. We seek techniques that avoid relying on specific behaviors of the transport protocol, generalize across transports and transport algorithms, and sidestep the need for new extensions to transport protocols.

G4. Capable of being run "online," in particular on 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.* [35, 89]. 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

Measuring latency is a deeply studied problem in networking. In particular, significant prior work exists on passive measurement to observe round-trip times (RTTs) at the transport layer. Below, we survey the literature and the suitability of the ideas therein to attaining our measurement goals (§2.1).

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 [50, 60, 66, 95, 96, 107]. Implementing this basic approach requires visibility into transport headers in the clear (**G1**), and seeing packets in both directions (**G2**).

We are aware of only a small number of techniques which apply when the passive observor sees packet traffic in only one direction. An early approach from Jiang and Dovrolis [67] 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 [45, 107]. Such computations require significant memory to hold packet timings (**G4**), and can be complex to tune [50]. Most recently, researchers have proposed the spin bit as a protocol extension (**G3**) to passively measure RTTs [48]. It has been incorporated as an optional mechanism in QUIC [12], but its deployment may be limited [77].

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, HTTP/2, and QUIC), 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 the clients maintain a single outstanding request at a time with the server (our full experimental setup is described in §5.1). For one of the client threads, Figure 1 shows the time series of response latencies (marked `req-to-res`) against three techniques to measure RTT: (i) using the kernel network stack

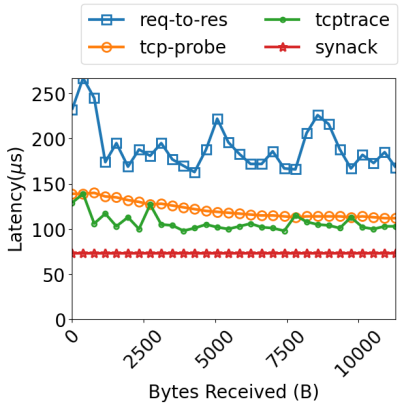

Figure 1: TCP Round-Trip Time (RTT) estimation from the network stack (`tcp-probe`), trace-based methods (`tcpTrace`), and syn-ack estimation [67] do not match the client-visible response latency.

through `tcp-probe` [27]; (ii) by estimation from a packet trace (`tcptrace` [96]; and (iii) syn-ack estimation [67]. Not only is the response latency different from the RTT, but it is also uncorrelated.

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 [64, 109]. These metrics compose the response latencies of multiple objects, *e.g.* to estimate overall web experience [32], 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.* [43, 59, 75, 78] and algorithms to estimate latencies over end-to-end network paths using previously measured delays, *e.g.* [83, 87]. Plenty of active approaches exist to measure end-to-end network latency characteristics, *e.g.* [100, 101, 112]. This paper seeks continuous passive measurement techniques for end-to-end response latency.

In summary, we believe that measuring response latencies passively and continuously under modern transports and practical deployment constraints (§2.1) is as yet an unsolved problem. In the next section, we discuss a measurement approach that addresses this problem.

3 PIRATE

This section introduces an algorithm to measure response latencies 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 [9, 15, 26]. (3) *Acknowledgments.* A response may trigger an acknowledgment at either the transport or the application layer.

In a simple setting, 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 [67].

There are two caveats to using causal pairs to estimate response latency. First, a key source of error arises from the additional time that the client application takes to process a response and generate the follow-up request, *i.e.* the *client think time.* In experiments on lightly-loaded clients, we have measured client think times of a few microseconds to a few tens of microseconds, and found its value to be independent of the true response latency. Second, a measurement is only available for one request per causal pair, not all the requests on the connection. Our experimental results (§5) show that the distribution of the samples of response latency we obtain through causal pairs is representative of the true distribution of the response latency across all requests.

A key challenge arises when we go to settings where multiple requests or packets may be 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 [60, 66] (see **G3** in §2.1).

3.2 Prominent Packet Gap Assumption

When a client has many requests and packets in flight, our key insight to identify causal pairs is to leverage the timings

of packet arrivals. The same 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. We exploit the observation that in many practical scenarios, the pause is noticeably longer than 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. Morever, the time interval between two consecutive causally-triggered requests provides an estimate of the response latency.

When a causal pair is separated by a burst-then-pause pattern of packet arrivals, we say that the connection satisfies the *prominent packet gap assumption.*

The prevalence of scenarios where the assumption holds is well documented: TCP senders that use window-based transmission send bursts of packets, termed flowlets [99, 105], separated by a pause, termed the flowlet gap. However, the assumption does not hold universally. Packets arriving at the vantage point may be uniformly paced either by the sender's transport, *e.g.* [44, 61] or a bottleneck link or policer [55, 70]. 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 Alg.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 $undef$

```
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 if
6: f.time_last_pkt ← now
7: return T̂_LB
```

How should we choose the time threshold δ? This question has a substantial impact to the accuracy of our response latency 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 [94], 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 [37, 69, 99, 105] 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 observor attempting to measure latency in the first place.

3.3 Choosing a Packet Gap Threshold

Producing a response latency instantaneously after each causal pair is subject to determining a hard-to-configure pause time threshold δ (§3.2 and Alg.1). In contrast, we observe that the timings of packet arrivals *over time* can provide meaningful clues. 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 [58, 72, 93, 110]. We estimate the *average response latency* of each connection 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 (typically the largest modes).

Through knowledge of standard parameters used for retransmission timeouts in transport stacks (publicly known for several standard TCP and QUIC configurations [5, 17, 106]), it is possible to eliminate the modes corresponding to retransmission timeouts and idle periods, leaving behind the

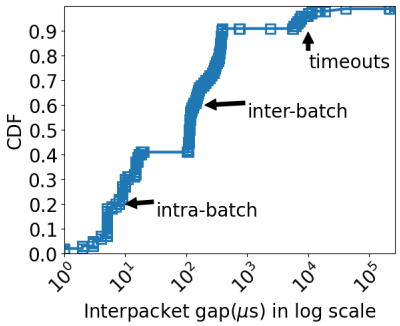

Figure 2: Modes in the empirical distribution of inter-packet gaps (IPGs). Modes carry useful information about phenomena of interest occurring over the measurement epoch.

de-noised distribution of IPGs, which only includes gaps between request packets that are either (i) back-to-back packets sent without dependencies among them, or (ii) triggered packets that are sent after a delay corresponding to the response latency of a packet from the previous batch.

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.* [51, 92]. 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 could also get expensive: With a sufficiently fine-grained time resolution, say, 10 μs (RTT in some real data centers [28, 54]), 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, larger than the L2 caches on many servers, implying 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 resolution used.

3.4 Maintaining Efficient Histograms

To maintain an IPG distribution with memory efficiency for each connection (§3.3), we design an algorithm that maintains a small number of buckets by dynamically varying the resolution of each bucket according to the observed IPGs on that connection. Dynamic bucket resolution provides some advantages: (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 for bucket counters on the IPGs that are indeed observed.

Our algorithm is shown in Alg.2. The histogram, M, includes a (configurable) maximum number of buckets (modes) N. Initially, all bucket 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. The assessment of whether a g is "close enough" to an existing bucket is to check whether $min - \varepsilon \leq IPG \leq max + \varepsilon$ for a fixed parameter ε. 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 histogram M to emit an average response latency over the epoch.

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.* [85]. We leave the adaptation of such algorithms to our scenario and a quantitative comparison to future work, since our experiments (§5) show that our current algorithm is practical.

To further improve the accuracy of Alg.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.

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

```
1:  function UpdateModes(IPG g)
2:      for m ∈ M.get_modes() : do
                          ▷ Modes traversed in ascending order
3:          left = m.get_min()
4:          right = m.get_max()
5:          if left − ε ≤ g ≤ right + ε then
                    ▷ IPG g lies within or proximal to mode m
6:              AddGapToMode(g, m)
7:              if left − ε ≤ g ≤ left then
                                    ▷ g is proximal from below
8:                  m.set_min(g)
9:                  ConsiderMergeModes(m.get_prev(), m)
10:             else if right ≤ g ≤ right + ε then
                                    ▷ g is proximal from above
11:                 m.set_max(g)
12:                 ConsiderMergeModes(m, m.get_next())
13:             end if
14:             return
15:         end if
16:     end for
17:     if M has fewer than N initialized modes then
                              ▷ Insert singleton mode containing g
18:         AddMode(M, g)
19:     else
20:         discarded += 1
21:     end if
22: end function
```

4 Use Case: DSR Layer-4 Load Balancer

To showcase the utility of real-time response latency measurement, we show the design of a latency-optimizing layer-4 load balancer, which assigns incoming client connections to backend servers based on real-time server performance. Such load balancers implement *direct server return (DSR),* a mechanism to allow backend servers to send responses directly to the client, bypassing the load balancer [90] on the path from the server to the client. 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.* [25, 25, 39, 57, 73, 84, 102, 111]. However, we do believe that a DSR load balancer is a compelling use case for using continuous and passively-measured response latencies to drive real-time feedback control under routing asymmetry and encrypted transports. The goal of this section is to describe one way in which such control can be designed.

Our design augments a Maglev-hashing-based load balancer [53]. We assume that the load balancer provides mechanisms to assign weights to distribute the load across 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 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 [111], 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 [102, 111]. 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 [88].

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.

5 Evaluation

In this section, we empirically evaluate PIRATE to answer the following questions:
(§5.2) How accurate is PIRATE in measuring response latencies under realistic applications and settings?
(§5.3) How does the accuracy of PIRATE compare to prior measurement approaches (§2.2)?
(§5.4) Is PIRATE robust to factors which might affect packet timings—packet loss, reordering, and request load?
(§5.5) What compute and memory overheads does PIRATE impose on a software packet-processing middlebox?
(§5.6) Does PIRATE produce fresh-enough response latency measurements to drive real-time feedback control systems?
(§5.7) How well does PIRATE generalize across transports and applications?
(§5.8) Do the heuristics (§3.4) help improve accuracy?

5.1 Experimental Setup

***Implementation of* PIRATE.** PIRATE runs as a kernel bypass program developed in Linux eBPF, which attaches to the express data path (XDP [65]) 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 [97], developed and open sourced by Meta. PIRATE runs as a program that calls Katran using a BPF tail call. Katran implements direct server return. The measurement component of PIRATE was developed in roughly 500 lines of C code.

Web benchmarking framework. We use the WebPolygraph suite [2] as our HTTP 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 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 provided probability distribution; and (4) the probability distributions of the sizes of each object type. The WebPolygraph server is single-threaded. Under the client workload we evaluate (see below), each client-to-server connection saturates at a load of 4K requests/second under 100% allocation of a CPU core. Each WebPolygraph client maintains at most 4 outstanding requests, compatible with default limits on many browsers.

Client workload. We populate web page parameters (types, sizes, object dependencies) into WebPolygraph by empirically measuring a sample of 20 of the Alexa top-100 sites.

Server CPU performance. 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 [13, 108] (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 [52]. Unless specified otherwise, we use a topology with three machines connected in a triangle topology. The three machines consist of a client, server, and a measurement vantage point. Each machine is a `c220g1` instance on Cloudlab, with two Intel E5-2630 8-core CPUs, 128GB memory, and dual-port 10Gb NIC. Packets are routed from the client to the server via the vantage point. Server-to-client packets are routed asymmetrically on the direct link between the server and the client. The PIRATE measurement XDP program is attached to the client-facing ingress interface on the vantage point.

Baselines. We instrumented the WebPolygraph client to compute the time series of response latencies for each object on each connection. This response latency is labeled `req-to-res` on our graphs. However, clients can inflate the time between causal pairs due to response processing on the client side. Hence, we also show the request-to-triggered-request delay measured at the client for reference, labeled `req-to-req` on our graphs. In situations where we compare PIRATE against RTT estimators, we have evaluated (1) `tcptrace` [96], an open-source tool that analyzes pcap traces to produce latency estimates, (2) `tcp_probe` [27], a tool that emits the sample RTT estimates maintained by the transport layer on the Linux network stack; and (3) syn-ack estimation [67, 103], which is the time between the SYN and the ACK in the TCP 3-way handshake. We compare the delays measured by the different techniques on the same set of sequence numbers at the transport layer, ensuring that our comparisons are "apples to apples" on the same data. The one exception is the syn-ack estimator, which only produces a single latency estimate for the whole connection.

Definition of accuracy. When we compare a technique T against a baseline (*e.g.* ground truth) technique B, we report

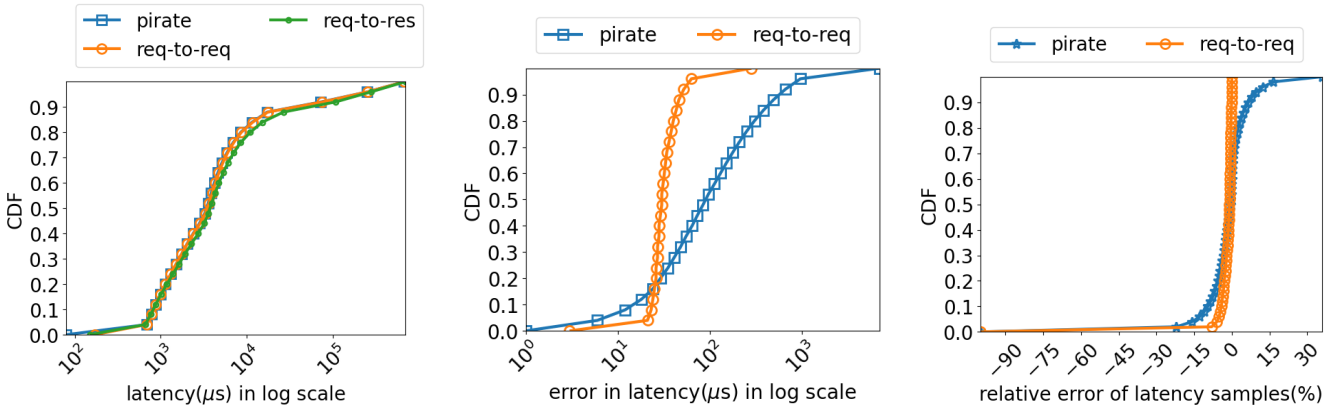

Figure 3: (a) CDF of observed latencies across all the connections for the web workload in §5.1. (b) CDF of absolute errors across all responses (relative to response latency). (c) CDF of relative errors across all responses (relative to response latency).

the absolute error $latency_B - latency_T$ and the relative error $\frac{latency_B - latency_T}{latency_B}$.

5.2 Accuracy of PIRATE

We evaluate the accuracy of PIRATE under the settings described in §5.1. In this experiment, we offer a load of 2K requests/second in total to the server from all client threads and connections. Figure 3 (a) shows the CDF of the observed latencies at the client and the vantage point across all objects across all connections. The three curves are closely aligned, showing that PIRATE, which estimates the request-to-triggered-request delay (labeled `req-to-req`), is able to match that and also the ground truth response latency (labeled `req-to-res`). Figure 3 (b) shows the CDF of the absolute error of PIRATE and the client-side request-to-request latency, against the ground truth response latency, while Figure 3 (c) shows the CDF of the relative errors. PIRATE's median relative error across all responses on all connections is less than 1%, but stretches beyond $\pm$15% at both tails.

We investigated where PIRATE makes errors (Figure 4). We observe two kinds of errors. First, when the client has a chance to transmit, but fails to transmit soon due to delays from process scheduling and response processing, PIRATE sometimes interprets the inter-packet gap as a pause (`parsing-scheduling`). Second, when a response is large or incurs a significant processing time at the server, the response packets may be spread over multiple network RTTs. Hence, what is normally one burst of requests is now broken into several "mini-flights" (say, just one request) which is released after a single previous response is fully received. PIRATE may mistake the pause between consecutive requests as a pause between batches, while it is only a pause between two adjacent requests which are not causally dependent on each other (`staggered-response`). Figure 4 (a) shows the CDF of

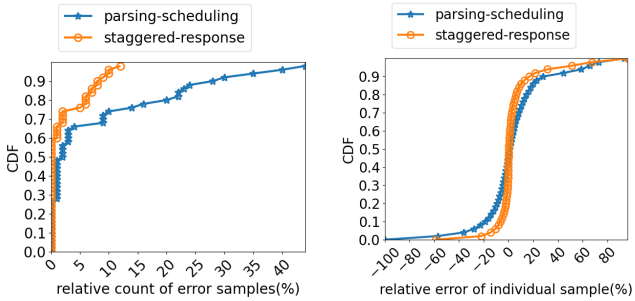

Figure 4: (a) CDF of samples that differ from request-to-request delay due to parsing/scheduling delays at client or processing delays at server (b) CDF of number of sample within each connection that differ from request-to-request delay due to parsing/scheduling delays at client or processing delays at server

the fraction of erroneous samples of response latency from PIRATE for each connection. Figure 4 (b) shows the CDF of the average relative error of the erroneous samples emitted by PIRATE relative to the `req-to-req` baseline.

5.3 Comparison Against RTT Measurement

To perform a faithful comparison of PIRATE against RTT measurement approaches (§2.2), in this subsection, we restrict the web workload from §5.1 to hold only a single request in flight at a time on a single connection. A majority of web objects (responses) are small enough to fit into a single packet, which increases the likelihood that RTTs can faithfully model the response latency. We compare PIRATE against the previously-used baselines (ground truth response latency, as well as request-to-triggered-request time), as well as RTT

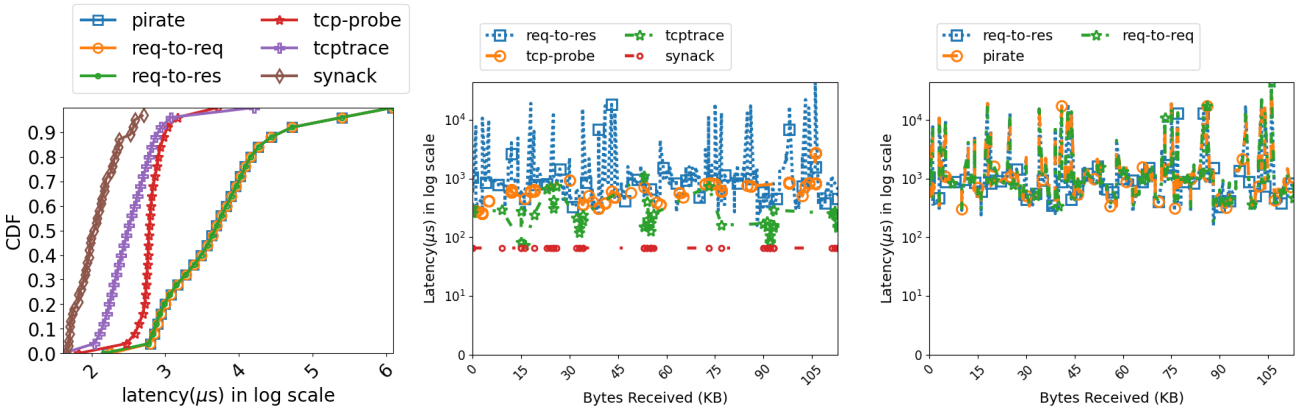

Figure 5: (a) CDF of all measurements throughout the lifetime of all connections. (b) and (c): Time evolution of the latency of a single (randomly chosen) connection during a small window of data transfer.

measurement baselines `tcp-probe`, `tcptrace`, and syn-ack estimation. Note that all techniques except PIRATE and syn-ack have visibility into both directions of packet traffic.

Figure 5 (a) compares the CDF of the latencies measured by these techniques over all objects and connections. TCP RTT measures consistently underestimate the response latency, since they are more closely aligned with the time to first byte of the response, as opposed to the last byte. Figure 5 (b) and (c) show latencies measured by these techniques over a small interval of data transfer. Syn-ack delay is not representative of response latency within a connection, as the latter varies significantly over the connection lifetime.

5.4 Robustness

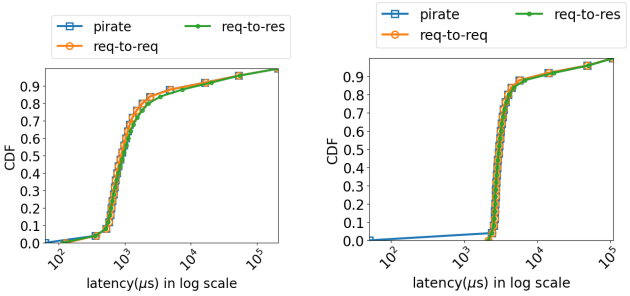

Figure 6: (a) CDF of latencies for a loss rate of 1% (b) CDF of latencies for a packet reorder rate of 25%

Packet Loss and Reordering. We evaluate the robustness of PIRATE to packet loss and reordering, which may affect packet inter-arrival times at the vantage point due to the transport layer's adaptation. 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 6 (a) shows the CDF of latencies measured by PIRATE under a loss rate of 1%, while Figure 6 (b) shows the CDF of latencies under a high packet reorder rate of 25%. PIRATE produces robust estimates of response latency under both conditions. Estimation accuracies were similar under other loss and reorder rates that we evaluated.

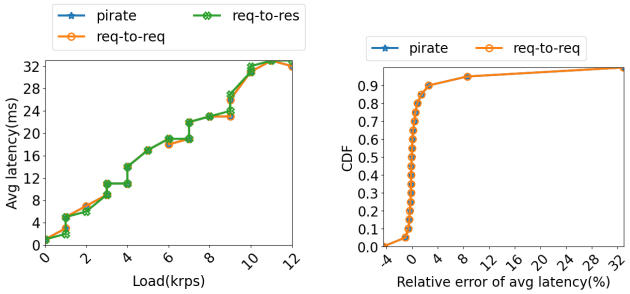

Figure 7: (a) Average response latency across all connections under different offered loads. (b) CDF of error relative to the response latency (averaged per connection) at 12K requests/second.

Robustness to Offered Load. We evaluated the accuracy of latency estimation as the load offered to the server varies. In this experiment, we use the simplified web workload from §5.3 to push the server load to higher request/second values. Figure 7 (a) shows the average latency across all connections at varying loads (requests/second). At a load of 12K requests/second, in Figure 7 (b), we show the CDF of relative error

(averaged per connection) across the techniques. In both cases, PIRATE agrees strongly with the response latency measured at the client.

5.5 Overheads of PIRATE

Feature	Memory	Latency
Katran	458MB	1114ns
Katran + tail	458MB	1125ns ($\delta = 11ns$)
Katran + measure	470MB ($\delta = 12MB$)	1443ns ($\delta = 329ns$)
Redirect	968 kB	871ns
Redirect + tail	968 kB	1022ns ($\delta = 151ns$)
Redirect + measure	11 MB ($\delta = 10MB$)	1396ns ($\delta = 525ns$)

Table 1: PIRATE's overheads, in relation to basic packet forwarding and the Katran load balancer.

We compare the run times and memory of executing our measurement algorithm (§3) within the XDP framework. To put these numbers in context, we provide numbers for basic packet redirection in XDP, and also the (unmodified) Katran load balancer. To chain PIRATE with Katran, we used the eBPF tail-call mechanism, which imposes a non-trivial compute overhead, so we show the overhead of using tail calls separately from using tail calls with our algorithm. Table 1 shows the compute time and memory numbers. Here, δ is the variance we measured across 5 runs. The extra latency incurred by PIRATE is in the range of 300-350 ns, while the memory overhead per connection is 154 Bytes/connection. (For 65K connections, this comes out to ~10MByte, which we believe is affordable on modern servers.) We believe that these overheads are overall reasonable, and can be shrunk further through careful optimization of the data structures and the memory accesses per packet.

5.6 Feedback Control

We set up a larger experimental topology specifically to evaluate our load balancer (§4). Here, we use seven machines—two clients connecting to four servers, and a machine for the vantage point. Each machine is a Cloudlab `xl170` instance, with 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 machines to forward packets from clients to servers via the vantage point, but server-to-client packets directly via the switch, mimicking a direct server return configuration [22]. The PIRATE algorithm runs on the switch-facing ingress interface of the vantage point.

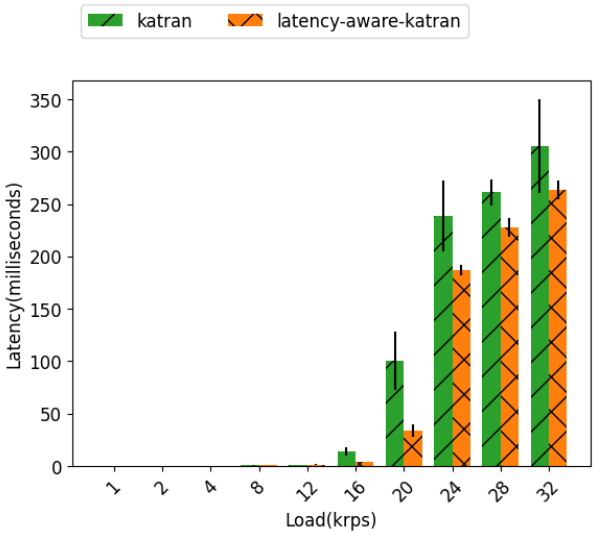

Figure 8: Comparison of 99th percentile tail latency of an unmodified Katran load balancer and our latency-aware one.

Figure 8 compares the 99th percentile tail response latency of our enhanced latency-aware Katran (§4) and the unmodified Katran. At varying loads, the latency-aware Katran is able to produce a sizable benefit in tail latency, and also leads to more predictable tail latencies (shorter error bars). The latency awareness provided by continuous and real-time response latency measurement enables reducing the 99th percentile tail response latency by 37% on average across loads.

5.7 Generalizing to Other Transports & Apps

Accuracy over QUIC connections. We set up an HTTP/3 QUIC client and server using Meta's mvfst and proxygen frameworks [31, 33]. As with our HTTP/TCP web client, we instrumented our QUIC client to emit response latencies as well as request-to-triggered-request delay for each response. The client workload is simpler than a full web-like workload (§5.1) in this experiment: we randomly choose response sizes for objects from a uniform distribution between two specified limits, and use a small static dependency tree of requests. We tested two scenarios: (1) when the responses of all requests in flight are small enough altogether to fit into a single packet; and (2) when the responses are large and trigger QUIC ACKs from the client. The response size in scenario 1 has a range of 30–130 bytes, while the response size in scenario 2 has a range of 1600–15100 bytes. In both cases, the HTTP header size is around 187 bytes. There are 1–16 multiplexed streams per connection.

PIRATE's ACK-coalescing heuristic (§3.4) requires differentiating pure ACK packets from those that contain data, a feature not readily supported by inspecting QUIC protocol headers. Hence, while our measurements for scenario (1) are

exact (labels ending with `-noack`), we simulated the correct identification of pure ACKs for scenario (2) (labels ending with `-ack`). Figure 9 compares the accuracy of PIRATE against the ground truth response latency and the request-to-triggered-request latency, for a connection running for over a minute. The result affirms the potential utility of PIRATE to measure multiplexed encrypted transports. In the future, we believe that either an extension to the protocol, or heuristics based on payload sizes, may be applied to accurately identify pure ACKs (scenario `-ack`) and practically realize these benefits.

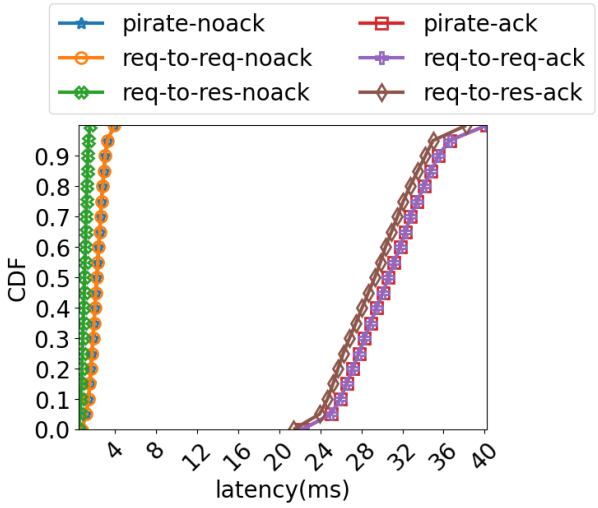

Figure 9: CDF of measured response latencies for a QUIC connection, in scenarios that do or do not generate pure ACK frames (`-ack` and `-noack`).

Accuracy for `memcached` ***clients.*** We set up a memcached benchmarking client using `memtier` [6], a noSQL traffic generator. The memtier benchmark is run with a total of 10 threads, with one client connection per thread. The proportion of get and set requests is 1:1. The data size for responses is in the range of 40–10000 bytes. The benchmark was run for 2 minutes. Figure 10 shows the CDF of the latency estimated by the various baselines. PIRATE closely follows the ground truth measured at the client. Similar to web clients and memtier, we believe that PIRATE may be useful to measure response latencies passively for other latency-sensitive applications that exhibit cross-request dependencies or flow control.

5.8 Benefits of Heuristics

Using our setup described in §5.1, we evaluate the benefits of the ACK-coalescing and MTU size heuristics (§3.4) in improving the accuracy of PIRATE's estimation. Figure 11 (a) compares the estimated response latencies when turning off both heuristics, against PIRATE. In (b), the time series of

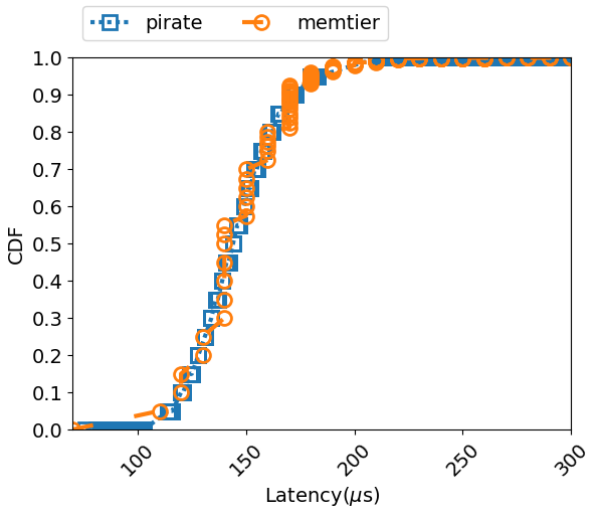

Figure 10: CDF of response latencies from memtier and PIRATE.

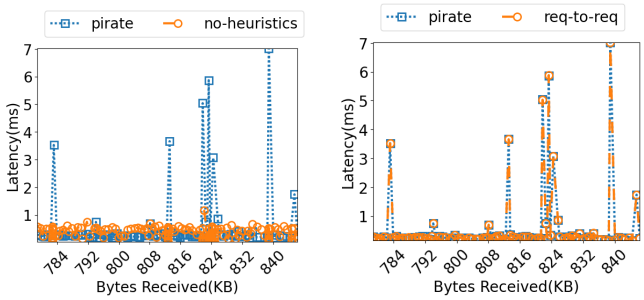

Figure 11: (a) PIRATE *vs* PIRATE without both heuristics. (b) PIRATE *vs* the request-to-triggered request delay.

request-to-triggered-request delays is shown against PIRATE for reference. The heuristics provide a noticeable improvement to accuracy, especially in allowing PIRATE to track fast fluctuations in latency.

6 Conclusion

This paper presented PIRATE, an algorithm that passively and continuously measures response latencies under routing asymmetry and under encrypted transport headers. PIRATE leverages the idea of causal pairs, two requests the second of which is triggered by the response to the first. In experimental evaluation with realistic client workload and server performance variations, PIRATE shows promising accuracy, and measures latency variations fast enough to enable real-time feedback control. Our results may have broader implications to the design of reactive feedback control systems relying on passive continuous measurement.

References

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

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

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

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

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

[6] 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.

[7] 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.

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

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

[10] 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.

[11] 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.

[12] 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.

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

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

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

[16] HTTP/3 adoption. [Online, Retrieved Jan 26, 2025.] https://almanac.httparchive.org/en/2021/http#http3-adoption, 2021.

[17] 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.

[18] 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.

[19] 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.

[20] 21 SSL Statistics that Show Why Security Matters so Much. [Online, Retrieved Jan 26, 2025.] https://webtribunal.net/blog/ssl-stats, 2023.

[21] 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.

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

[23] 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.

[24] Discovering HTTP/3 support (Web Almanac). [Online, Retrieved Jan 26, 2025.] https://almanac.httparchive.org/en/2024/http#discovering-http3-support, 2024.

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

[26] 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.

[27] 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.

[28] 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.

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

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

[31] mvfst: IETF QUIC by Facebook. [Online, Retrieved Jan 26, 2025.] https://github.com/facebook/mvfst, 2025.

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

[33] Proxygen: Facebook's C++ HTTP Libraries. [Online, Retrieved Jan 26, 2025.] https://github.com/facebook/proxygen, 2025.

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

[35] 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.

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

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

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

[39] Tom Barbette, Chen Tang, Haoran Yao, Dejan Kostić, Gerald Q. Maguire Jr., Panagiotis Papadimitratos, and Marco Chiesa. A High-Speed Load-Balancer design with guaranteed Per-Connection-Consistency. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

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

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

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

[43] Christopher Canel, Balasubramanian Madhavan, Srikanth Sundaresan, Neil Spring, Prashanth Kannan, Ying Zhang, Kevin Lin, and Srinivasan Seshan. Understanding incast bursts in modern datacenters. In *ACM Internet Measurement Conference (IMC)*, 2024.

[44] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. Bbr: Congestion-based congestion control. *ACM Queue*, 2016.

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

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

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

[48] Piet De Vaere, Tobias Bühler, Mirja Kühlewind, and Brian Trammell. Three bits suffice: Explicit support

for passive measurement of internet latency in quic and tcp. In *Internet Measurement Conference (IMC)*, 2018.

[49] 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. Inferring persistent interdomain congestion. In *ACM SIGCOMM*, 2018.

[50] Hao Ding and Michael Rabinovich. Tcp stretch acknowledgements and timestamps: findings and implications for passive rtt measurement. *ACM SIGCOMM Computer Communication Review*, 2015.

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

[52] 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. The design and operation of CloudLab. In *USENIX Annual Technical Conference (ATC)*, 2019.

[53] Daniel E Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In *Usenix Symposium on Networked Systems Design and Implementation (NSDI)*, 2016.

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

[55] Tobias Flach, Pavlos Papageorge, Andreas Terzis, Luis Pedrosa, Yuchung Cheng, Tayeb Karim, Ethan Katz-Bassett, and Ramesh Govindan. An internet-wide analysis of traffic policing. In *ACM SIGCOMM*, 2016.

[56] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. 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)*, 2019.

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

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

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

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

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

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

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

[64] Alexis Huet, Antoine Saverimoutou, Zied Ben Houidi, Hao Shi, Shengming Cai, Jinchun Xu, Bertrand Mathieu, and Dario Rossi. Revealing qoe of web users from encrypted network traffic. In *IFIP Networking Conference (Networking)*, 2020.

[65] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The express data path: Fast programmable packet processing in the operating system kernel. In *Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2018.

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

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

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

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

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

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

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

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

[74] Ron Kohavi, Randal M Henne, and Dan Sommerfield. 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*, 2007.

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

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

[77] Ike Kunze, Constantin Sander, and Klaus Wehrle. Does it spin? on the adoption and use of quic's spin bit. In *Proceedings of the 2023 ACM on Internet Measurement Conference*, pages 554–560, 2023.

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

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

[80] Shihan Lin, Yi Zhou, Xiao Zhang, Todd Arnold, Ramesh Govindan, and Xiaowei Yang. Latency-aware inter-domain routing, 2025.

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

[82] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. Characterizing microservice dependency and performance: Alibaba trace analysis. In *ACM Symposium on Cloud Computing (SoCC)*, 2021.

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

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

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

[86] Ravi Netravali, Ameesh Goyal, James Mickens, and Hari Balakrishnan. Polaris: Faster page loads using fine-grained dependency tracking. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2016.

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

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

[89] Palo Alto. 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.

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

[91] Vern Paxson. *Measurements and Analysis of End-to-End Internet Dynamics*. PhD thesis, University of California, Berkeley, 1997.

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

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

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

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

[96] Shawn Ostermann. tcptrace. [Online, Retrieved Jan 26, 2025.] https://www.tcptrace.org/index.shtml.

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

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

[99] Sanjay Sinha, Srikanth Kandula, and Dina Katabi. Harnessing tcp's burstiness with flowlet switching. In *ACM Workshop on Hot Topics in Networks (HotNets)*, 2004.

[100] Joel Sommers, Paul Barford, Nick Duffield, and Amos Ron. Accurate and efficient sla compliance monitoring. In *ACM SIGCOMM*, 2007.

[101] Srikanth Sundaresan, Mark Allman, Amogh Dhamdhere, and Kc Claffy. Tcp congestion signatures. In *ACM Internet Measurement Conference (IMC)*, 2017.

[102] Lalith Suresh, Marco Canini, Stefan Schmid, and Anja Feldmann. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[103] Michal Szymaniak, David Presotto, Guillaume Pierre, and Maarten van Steen. Practical large-scale latency estimation. *Computer Networks*, 2008.

[104] Steve Uhlig and Olivier Bonaventure. Designing bgp-based outbound traffic engineering techniques for stub ases. *ACM SIGCOMM Computer Communication Review*, 34(5):89–106, 2004.

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

[106] Vijay Vasudevan, Amar Phanishayee, Hiral Shah, Elie Krevat, David G Andersen, Gregory R Ganger, Garth A Gibson, and Brian Mueller. Safe and effective fine-grained tcp retransmissions for datacenter communication. *ACM SIGCOMM computer communication review*, 2009.

[107] Bryan Veal, Kang Li, and David Lowenthal. New methods for passive estimation of tcp round-trip times. In *Passive and Active Network Measurement (PAM)*, 2005.

[108] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *European Conference on Computer Systems (EuroSys)*, 2015.

[109] Nikolas Wehner, Michael Seufert, Joshua Schuler, Sarah Wassermann, Pedro Casas, and Tobias Hossfeld. Improving web qoe monitoring for encrypted network traffic through time series modeling. *SIGMETRICS Perform. Eval. Rev.*, 2021.

[110] Damon Wischik and Nick McKeown. Part i: Buffer sizes for core routers. *ACM SIGCOMM Computer Communication Review*, 2005.

[111] Bartek Wydrowski, Robert Kleinberg, Stephen M. Rumble, and Aaron Archer. Load is not what you should balance: Introducing prequal. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2024.

[112] Curtis Yu, Cristian Lumezanu, Abhishek Sharma, Qiang Xu, Guofei Jiang, and Harsha V Madhyastha. Software-defined latency monitoring in data center networks. In *Passive and Active Measurement*, 2015.

[113] Minlan Yu, Albert Greenberg, Dave Maltz, Jennifer Rexford, Lihua Yuan, Srikanth Kandula, and Changhoon Kim. Profiling network performance for

multi-tier data center applications. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[114] Jiao Zhang, F Richard Yu, Shuo Wang, Tao Huang, Zengyi Liu, and Yunjie Liu. Load balancing in data center networks: A survey. *IEEE Communications Surveys & Tutorials*, 2018.

[115] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y Zhao, et al. Packet-level telemetry in large datacenter networks. In *ACM SIGCOMM*, 2015.