I Sent It: Where Does Slow Data Go to Wait?

Youngbin Im
University of Colorado Boulder
Youngbin.Im@colorado.edu

Parisa Rahimzadeh
University of Colorado Boulder
Parisa.Rahimzadeh@colorado.edu

Brett Shouse
University of Colorado Boulder
Brett.Shouse@colorado.edu

Shinik Park
Ulsan National Institute of Science and Technology
sipark@unist.ac.kr

Carlee Joe-Wong
Carnegie Mellon University
cjoewong@andrew.cmu.edu

Kyunghan Lee
Ulsan National Institute of Science and Technology
khlee@unist.ac.kr

Sangtae Ha
University of Colorado Boulder
Sangtae.Ha@colorado.edu

Abstract

Emerging applications like virtual reality (VR), augmented reality (AR), and 360-degree video aim to exploit the unprecedentedly low latencies promised by technologies like the tactile Internet and mobile 5G networks. Yet these promises are still unrealized. In order to fulfill them, it is crucial to understand where packet delays happen, which impacts protocol performance such as throughput and latency. In this work, we empirically find that sender-side protocol stack delays can cause high end-to-end latencies, though existing solutions primarily address network delays. Unfortunately, however, current latency diagnosis tools cannot even distinguish between delays on network links and delays in the end hosts. To close this gap, we present ELEMENT, a latency diagnosis framework that decomposes end-to-end TCP latency into endhost and network delays, without requiring admin privileges at the sender or receiver.

We validate that ELEMENT achieves more than 90% accuracy in delay estimation compared to the ground truth in different production networks. To demonstrate ELEMENT's potential impact on real-world applications, we implement a relatively simple user-level library that uses ELEMENT to minimize delays. For evaluation, we integrate ELEMENT with legacy TCP applications and show that it can reduce latency by up to 10 times while maintaining throughput and fairness. We finally demonstrate that ELEMENT can significantly reduce the latency of a virtual reality application that needs extremely low latencies and high throughput.

Keywords TCP latency, Measurement tool, Latency control

ACM Reference Format:
Youngbin Im, Parisa Rahimzadeh, Brett Shouse, Shinik Park, Carlee Joe-Wong, Kyunghan Lee, and Sangtae Ha. 2019. I Sent It: Where Does Slow Data Go to Wait?. In *Fourteenth EuroSys Conference 2019 (EuroSys '19), March 25–28, 2019, Dresden, Germany.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3302424.3303961

1 Introduction

Emerging real-time Internet applications such as teleconferencing, live video streaming, telemedicine, and 3D virtual reality (VR) and augmented reality (AR) require unprecedentedly low latency and jitter when communicating over data networks. In response to these needs, the tactile Internet has promised extremely low latency in combination with high availability, reliability and security [1], and fifth generation (5G) mobile networks target 1 millisecond air latency [16]. New application architectures that concentrate computational power on the network edge, e.g., cloudlets [51], edge computing [26], and fog networking [20] similarly attempt to provide the infrastructure for frequent, real-time interaction between users and edge devices.

Despite these advances, many end devices cannot provide low-latency services. For example, they may suffer from a phenomenon called "bufferbloat" [27, 28], a significant increase in queuing delay and jitter due to large buffers in routers, hosts, access points, basestations, etc. These buffers were introduced to maximize throughput by minimizing the number of packet drops, thus reducing overhead on communication links, but can significantly increase end-to-end latency due to extra latency within the buffers.

One common approach to solving the bufferbloat problem is to drop packets in the routers to keep the average packet delay below a target threshold [45, 46]. Another, orthogonal, approach will be pinpointing the causes of this excessive delay and using them to control the latency. In Section 2, for instance, we show that delays in the endhost can significantly exceed those in the network. Existing latency measurement solutions, however, cannot even distinguish between endhost and network delays on the flow path. This lack of visibility has driven existing latency reduction efforts to either solely target network delays or to abandon TCP's transport layer altogether by combining UDP with re-designed application layer protocols [23]. While the latter solution can be effective, it is a drastic step that may compromise other benefits of TCP, such as reliable transfer and fair throughput sharing among

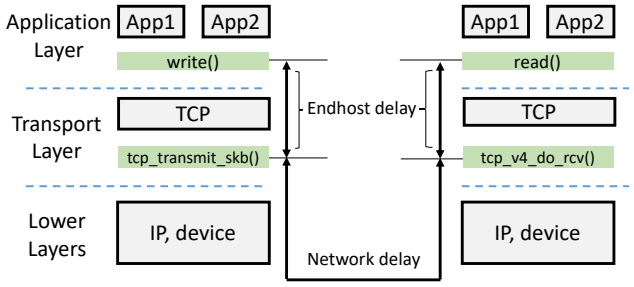


Figure 1. The Linux protocol stack is illustrated. ELEMENT targets the system delay on endhosts incurred between the socket API *write* and TCP stack's *tcp_transmit_skb* and *tcp_v4_do_rcv* and *read*. This TCP socket buffer latency constitutes the majority of the system delay we observed.

different flows. Many applications may therefore prefer to use TCP as long as they can mitigate its latency challenges.

1.1 Challenges and existing approaches

Many existing tools can measure various aspects of data transport performance over the Internet, as classified in [11]. However, *none of these tools provide information on the endhost TCP delays.* Existing latency diagnosis tools such as `ping` and `traceroute`, for instance, measure the end-to-end or hop-by-hop delay, but do not provide endhost delays.

Differentiating endhost delays from end-to-end delays requires visibility into multiple layers of the protocol stack on end hosts. Figure 1, for instance, shows that a packet passes through multiple layers at both the sender and receiver. Since both `ping` and `traceroute` are based on the Internet Control Message Protocol (ICMP), a network layer protocol, they cannot measure the latency incurred in the transport layer and above. Doing so would require determining the delay incurred by each protocol stack component, e.g., with visibility into different buffers at the host. Such visibility often requires root access and deep understanding of the networking stack, which many users do not have. Note that the IP and physical layers that sit in network interface cards (NICs) are well optimized, and the delays incurred in these layers are far less than those incurred above the transport layer.[1] They often leverage techniques like TSO (TCP segment offload) or GSO (general segment offload) that move multiple packets instantly into the NIC.

In this paper, we introduce ELEMENT, a latency diagnosis framework in the application layer that decomposes end-to-end TCP latency into delays on network links and in the protocol stacks at the end hosts. ELEMENT is the *first* solution to provide this information to TCP applications so that they can accurately measure and control their latencies.

[1]Contention in wireless PHY and MAC layers can increase this delay, but it is still far less than that incurred in the TCP stack's socket buffer.

1.2 ELEMENT: Decomposing endhost and network delays at the user level

ELEMENT can *measure the endhost delays at the sender and receiver*, differentiating them from network delays. To do so, ELEMENT uses `tcp_info`[2] socket information in Linux, which is available at the user level, to obtain basic TCP statistics such as the number of acknowledged and unacknowledged bytes and timestamps. It then combines this with application-layer information to infer the delay incurred at the TCP stack. We measure the TCP socket buffer delays because they dominate the endhost delay in Section 2's experiments.

In demonstrating ELEMENT's effectiveness and deployability, we make the following contributions:

A practical endhost TCP latency measurement framework. We design and implement ELEMENT, a framework that overcomes the long-standing limitations of legacy latency measurement tools such as `ping` and `traceroute` to provide rich information on TCP latency. ELEMENT can decompose the TCP application latency into endhost and network latencies. Our testbed and Internet experiments show that ELEMENT achieves more than 90% accuracy compared to the ground-truth latency values at the sender and receiver.

New latency-aware TCP socket primitives at the user level. We provide ELEMENT's latency diagnosis in a user-level library. This library provides new socket primitives that make the endhost and network latencies visible to TCP applications, thus providing them with the first means to effectively monitor and control both latencies, e.g., applications can adjust their data rates according to the information provided by ELEMENT. We also show that ELEMENT can help legacy TCP applications optimize their latencies through its latency minimization algorithm. The algorithm uses application-level packet pacing based on latency measurement. With this, TCP maintains a proper amount of packets in the buffer to keep the latency low. ELEMENT library is transparent to user applications through dynamic binding, which is particularly attractive as it does not require code modifications.

Two representative TCP applications leveraging ELEMENT. To show that ELEMENT can be easily integrated with legacy applications as well as with new TCP applications requiring low latency, we implemented two representative applications: an open source network performance measurement tool `Iperf` and virtual reality streaming application that requires extremely low latencies with high throughput. Our experiments in production networks, including WiFi and LTE, show that the `Iperf` tool integrated with ELEMENT reduces TCP latency by up to 10 times, while maintaining the same or slightly higher throughput. Furthermore, the VR application is shown to meet its playback

[2]For iOS/MacOSX, `tcp_connection_info` can be used instead.

deadline by optimizing the data rate and latency (number of frames and their resolutions) based on the information provided by ELEMENT.

In Section 2, we show that existing latency mitigation techniques fail to identify the true sources of end-to-end latency, leading to suboptimal solutions. We discuss ELEMENT's architecture in Section 3, and then present details of how ELEMENT finds the endhost latencies at the sender and receiver (Section 4). In Section 5, we prototype two TCP applications that use ELEMENT to reduce their latencies. We discuss related work in Section 6 and possible solutions to our system limitations in Section 7. Section 8 concludes.

2 Latency in Real Networks

Though many Internet user devices suffer from high latencies, no studies have systematically quantified which components of the end-to-end delays contribute to this latency. To demonstrate the need for an ELEMENT-like latency diagnosis framework, we first anatomize the source of delays between two end hosts in our controlled testbed. We then show that current latency mitigation solutions fail to successfully reduce delays due to their inability to differentiate between network and endhost system delays.

2.1 Anatomy of end-to-end delays

To understand how each component on the path affects the end-to-end delay, we set up a testbed and observe the delays between each path component. The testbed consists of a sender, receiver, and WAN emulator between them that emulates network characteristics like bandwidth, delay, and packet loss. We collect network and system statistics by using `write()` and `read()` calls in the application layer, `tcp_v4_do_rcv()` and `tcp_transmit_skb()` in the transport layer, `dev_queue_xmit()` in the Linux queueing discipline, and `dev_hard_start_xmit()` in the physical device. While [32] reports non-negligible device driver delays, these were small compared to the transport system delay in our experiment and thus are not shown. We modify the Linux kernel profiler `perf` to measure the latency at each observation point.

Setup: The servers used for the experiment run Ubuntu 16.04 and use TCP Cubic [34], their default TCP congestion control algorithm. We develop and evaluate the latency of a simple TCP application that generates traffic to fully utilize the available bandwidth. We set the bandwidth at 10Mbps and one way delay at 25 msec by using Linux `tc` command on the WAN emulator, and we run three Cubic flows. We also use Linux's default FIFO queueing discipline *pfifo_fast* at the WAN emulator. Linux's default buffer tuning algorithm automatically sets the socket buffer size.

Results: Figure 2 shows the composition of the overall delay of a representative TCP Cubic flow in our simple application. The total delay is over 2.5 seconds, which is extremely large considering that the bandwidth delay product to fully utilize the network is only around 44 packets. The delays in the TCP sender and receiver dominate the total delay. This phenomenon of a large delay at the sender can be attributed to bufferbloat, or the rapid growth of the send buffer size caused by the interaction of loss-based TCP algorithms and Linux systems' default send buffer tuning algorithms. Loss-based TCP algorithms, such as TCP Cubic, increase the congestion window until a packet loss occurs, while the default send buffer tuning algorithms continue to increase the buffer size to a value of approximately two times that of the current congestion window size. In Section 5.1, we also show that even latency-optimized TCP protocols such as Vegas [18] and BBR [19] exhibit large delays at the TCP sender.

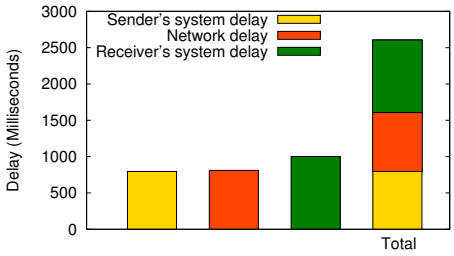


Figure 2. Delay composition of a TCP flow with the default queueing discipline (pfifo_fast) in Linux. We run three TCP Cubic flows in a 10 Mbps bandwidth and 25 ms one way delay network. We average the network delay and system delay at both end hosts for one flow.

2.2 Existing latency mitigation solutions

Existing solutions to the bufferbloat problem focus on better queue management to avoid excessive buffer sizes. However, they still lead to large overall latencies, suggesting that a more detailed latency diagnosis can help identify the necessary solutions for reducing latency.

We compare *pfifo_fast* with CoDel [45], FQ CoDel [35], and PIE [46], state-of-the-art AQM (active queue management) solutions to bufferbloat, which intelligently drop packets in the router. We evaluate their impact on four production networks: a wired network with 10 Mbps bandwidth and 25 ms one way delay (Low bandwidth), a wired network with 1000 Mbps bandwidth in local network (High bandwidth), a WiFi network (using Intel(R) Dual Band Wireless-AC 7260 for the client and ARRIS Touchstone TG1682G Telephony Gateway for the access point), and an LTE network (using Netgear AC340U modem in AT&T LTE network). As shown in Figure 3, the three bufferbloat mitigation solutions still exhibit relatively large endhost system delays in each network, indicating that they cannot completely meet end-to-end latency challenges. We also test ECN (explicit congestion notification) in the low bandwidth network, but

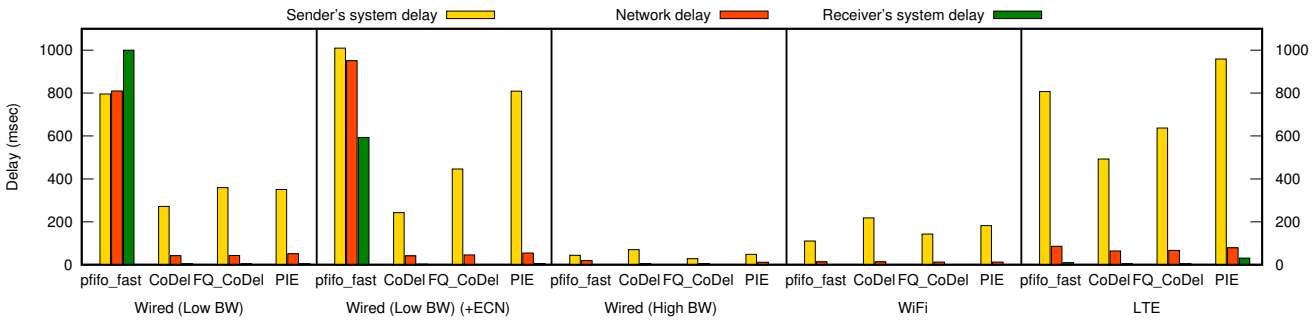


Figure 3. Delay composition for different queuing disciplines (unit: milliseconds) in a wired network with 10 Mbps bandwidth and 25 ms one way delay (Low BW), a wired network with 1000 Mbps bandwidth in the local network (High BW), a WiFi network, and an LTE network. ECN is tested on the Low BW network. The delay reductions from the latest bufferbloat mitigation solutions (CoDel, FQ CoDel, and PIE) are not enough as they still suffer from non-negligible endhost system delays.

it does not significantly affect the latency compared to the AQM schemes without ECN.

3 ELEMENT's Design

One feasible approach to reduce endhost delays would be to revisit TCP's buffer tuning and/or congestion control algorithms. This approach, however, blindly controls the TCP stack latency without feedback from or to the application, which could harm throughput-sensitive and/or latency-tolerant TCP applications. Furthermore, as computing becomes more virtualized in the cloud, users may not have the privileges needed to upgrade or modify the TCP stack in their systems. Thus, we instead implement ELEMENT as a lightweight *user-level* library that can be integrated into any TCP application without significant modifications or admin privileges. In this section, we first discuss our guiding design challenges (Section 3.1) and present ELEMENT's architecture in Section 3.2. We finally present use cases that can benefit from ELEMENT (Section 3.3).

3.1 Design challenges

In designing our user-level latency diagnosis framework, we solve several technological challenges:

Extrapolating from user-level information. Measuring the endhost delay requires knowing the latency from when the application sends data to when the data leaves the transport layer, and vice versa. However, the exact packet transmission times from each layer cannot be measured at the user level, requiring us to infer the true latencies.

Trading off between accuracy and overhead. As capturing latency at a very fine granularity in the application layer may incur too much system overhead, we have to consider the trade-off between the accuracy and overhead involved with this measurement.

Non-intrusive integration with TCP applications. For practical deployment, ELEMENT's user-space library should be easy to use and require minimal or no modification to legacy TCP applications. Thus, ELEMENT includes an algorithm that uses its delay measurements to reduce end-to-end latency, with APIs that can be easily invoked.

3.2 ELEMENT architecture

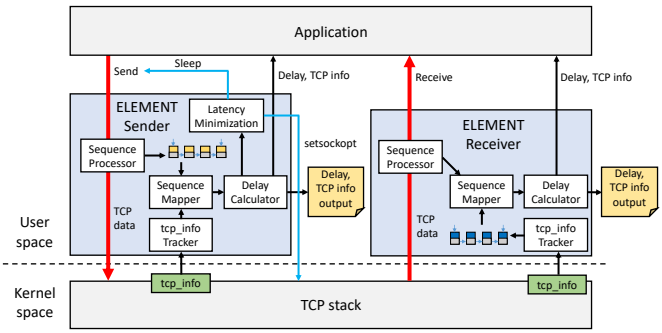


Figure 4. The architecture of ELEMENT is illustrated.

Figure 4 shows ELEMENT's architecture. ELEMENT leverages a standard BSD socket API, `getsockopt()`, with the `TCP_INFO` option to obtain basic TCP statistics. In particular, ELEMENT runs a `tcp_info` Tracker that collects TCP statistics in the transport layer while recording the time and the amount of data sent and received in the application at a Sequence Processor. The information in both layers is analyzed by a Sequence Mapper that matches the sequence numbers in the two layers. The Delay Calculator then uses this matching to estimate the time difference between when a packet is sent/received in the application and TCP layers. The calculated delay and other TCP information are sent to the application and recorded into output files. The latency minimization algorithm at the sender controls the transmission of packets with `epoll()` and `sleep()` and adjusts the TCP buffer size by calling the socket control function, if needed. We detail ELEMENT's latency measurement and minimization algorithms in Section 4.

3.3 Use cases

ELEMENT can be integrated into existing TCP applications as a lightweight library, decomposing the endhost and network latencies. Latency-sensitive applications can take advantage of this framework to improve their latencies and users' quality of service (QoS). Examples include:

TCP latency tuning. Users without admin privileges on either the sender or receiver can use ELEMENT to mitigate bufferbloat in TCP applications, without changing the buffer tuning and congestion control algorithms in the TCP stack. ELEMENT allows these applications to detect large delays due to buffer size by providing them with the system and network delays, throughput, and TCP congestion window size. We show that a legacy TCP application can significantly reduce its latency while maintaining throughput in Section 5.1.

TCP-based video conferencing. Video conferencing requires multidirectional, synchronized real-time streaming between different participants in a video call, which today is usually provided by UDP or application-layer protocols. With ELEMENT, these applications can monitor the send and receive latency of each stream while using TCP, identifying gaps in the synchronization and proactively moderating any latency increases before the streams fall too far out of sync.

Virtual reality (VR) and telemedicine. Both of these applications require real-time streaming of visual, aural, and/or tactile signals between the sender and receiver. Any delays can disrupt the user experience, with possibly serious consequences, e.g., in surgeons receiving tactile signals from the patient's body over the Internet. ELEMENT's latency monitoring can be used to alert users to interruptions and react by reducing their latencies. We demonstrate this effect with TCP-based virtual reality streaming in Section 5.2.

4 ELEMENT: End-to-End Latency Measurement

In this section, we present ELEMENT's delay decomposition and minimization algorithms. In order to estimate the system delay within the sender and receiver protocol stacks, we obtain TCP statistics for each flow with the `tcp_info` structure on both devices, by calling `getsockopt()` with TCP_INFO as the request code. The `tcp_info` structure includes statistics such as the number of bytes sent and acknowledged, congestion window size (cwnd), slow start threshold (ssthresh), RTT, and many more. Calling `getsockopt()` does not require admin or root privileges, making ELEMENT accessible to all device users.

4.1 Estimation of sender-side delay

Algorithm 1 presents the pseudo code for finding the delay at the TCP sender's protocol stack. In order to measure this delay, we subtract the time at which a packet is sent at the application layer from the time at which the packet is transmitted by the TCP layer to the lower layer. Finding these times is non-trivial: the application layer and TCP layer use different transmission sizes depending on the application buffer, socket buffer and TCP status, and the actual transmitted bytes in the TCP layer is not available in the current `tcp_info` implementation[3]. Thus, we must *infer* the times of transmission at the TCP layer corresponding to individual socket writes in the application layer.

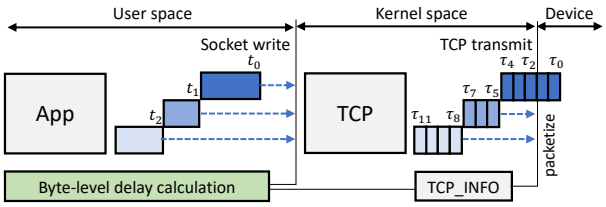


Figure 5. Operation of the algorithms calculating the delay at TCP sender and receiver. The application layer and TCP layer use different transmission/reception sizes. We estimate the transmitted/received bytes at TCP layer and match these bytes with the sequence numbers at socket write/read calls. We obtain the delay by calculating the time difference of these events.

We first define the variable *seq*, which records the cumulative number of actual bytes sent at the application layer when each call of the socket write function returns. We record the elapsed time (T) and the accumulated bytes (*seq*) to keep track of application write calls. Given that we know the sequence of when and how much the application writes, we now need to calculate when each application write call leaves the TCP stack to the lower layer. We estimate the accumulated bytes left in the TCP layer from `tcp_info` by adding the acknowledged bytes (*tcpi_bytes_acked*) and the number of unacknowledged segments (*tcpi_unacked*) times the sender's maximum segment size (*tcpi_snd_mss*). Finally, we can measure the delay between the application layer write calls and the TCP layer as the difference in the timestamps at the two layers for the matched bytes. We find the closest pair if the bytes do not match; we do not collect TCP statistics on every application write call due to `getsockopt()`'s CPU overhead. This method overestimates the transmitted bytes at TCP layer, but is still fairly accurate (cf. Section 4.3).

To implement this procedure, we introduce two threads: a data sending thread and a `tcp_info` tracking thread. The data sending thread records the total sent bytes and sending times in a linked list data structure when the application writes data. The `tcp_info` tracking thread, which runs periodically, estimates the number of sent bytes at the TCP layer. It then searches the entries starting from the back of the linked list of the total sent bytes. If it finds an entry whose total sent bytes do not exceed the estimated sent bytes at

[3]At the time of this writing, the most recent kernel version was 4.15.

Algorithm 1: Algorithm calculating the delay at the TCP sender with the `tcp_info` structure at the user level.

```
T_start ← // TCP connection start time
T_cur ← // current time
P ← 10 msec // sleep period
T ← T_cur − T_start // elapsed time
seq ← 0 // total sent bytes
sendInfo ← // a struct that consists of
    bytes, sendTime, next
tcp_info tracking thread: // Obtain, save the TCP information in
    ti
while true do
    B_est ←
      ti.tcpi_bytes_acked + ti.tcpi_unacked * ti.tcpi_snd_mss
    // estimated sent bytes at TCP layer
    while true do
        if back ≠ NULL and back.bytes ≤ B_est then
            D ← T − back.sendTime // buffer delay
            Print T, D, ti.tcpi_snd_cwnd,
              ti.tcpi_snd_ssthresh, ti.tcpi_rtt
            old_back ← back
            back ← back.next
            free old_back
            if back = NULL then
                front ← NULL
        else
            break
    Sleep for P
data sending thread:
while true do
    if front = NULL then
        Allocate front with sendInfo struct
    else
        old_front ← front
        Allocate front with sendInfo struct
        old_front.next ← front
    front.bytes ← seq
    front.sendTime ← T
    front.next ← NULL
    if back = NULL then
        back ← front
    Send a packet and add the packet size to seq
```

Algorithm 2: Algorithm for calculating the delay at the TCP receiver using the TCP_INFO socket option.

```
T_start ← // TCP connection start time
T_cur ← // current time
P ← 10 msec // sleep period
T ← T_cur − T_start // elapsed time
seq ← 0 // total received bytes
recvInfo ← // a struct that consists of
    bytes, recvTime, next
TCP INFO tracking thread: // Obtain, save the TCP information in
    ti
while true do
    B_est ← ti.tcpi_segs_in * ti.tcpi_rcv_mss // estimated
      received bytes at TCP layer
    if B_est > B_prev then
        B_prev ← B_est
        if front = NULL then
            Allocate front with recvInfo struct
        else
            old_front ← front
            Allocate front with recvInfo struct
            old_front.next ← front
        front.bytes ← B_est
        front.recvTime ← T
        front.next ← NULL
        if back = NULL then
            back ← front
    Sleep for P
data receiving thread:
while true do
    Receive a packet and add the packet size to seq
    while true do
        if back ≠ NULL then
            if back.bytes ≤ seq then
                old_back ← back
                back ← back.next
                free old_back
                if back = NULL then
                    front ← NULL
            else
                D ← T − back.recvTime // buffer delay
                Print T, D, ti.tcpi_snd_cwnd,
                  ti.tcpi_snd_ssthresh, ti.tcpi_rtt
                break;
        else
            break
```

the TCP layer, it calculates the buffer delay and sends this information back to the ELEMENT framework. Entries less than the total bytes will be removed from the list. Figure 5 illustrates this algorithm.

4.2 Estimation of receiver-side delay

The algorithm that calculates the delay in the TCP receive buffer (shown in Algorithm 2) uses a similar procedure as that of Algorithm 1 for the send buffer. On the receiver side, the packet is received first at the TCP layer and then conveyed to the application layer; thus, the `tcp_info` tracking thread records the timestamp and sequence number (cumulative received bytes) of `sk_buff` in the TCP layer storing the received packets in the linked list. The data receiving thread then matches these sequence numbers and its own records of data received at the application in order to calculate the receiver-side delay.

Specifically, we define the variable *seq* to record the cumulative number of bytes received at the application layer. We estimate the accumulated bytes received to `sk_buff` in the TCP layer from `tcp_info` by multiplying the total number of segments received (*tcpi_segs_in*) and the receiver's maximum segment size (*tcpi_rcv_mss*). Finally, we measure the delay between the application layer read calls and the TCP layer as the difference in the timestamps at the two layers for the matched bytes. We find the closest pair if the bytes do not match as in the sender-side algorithm.

4.3 Accuracy

To measure the ground-truth delays, we add trace points to `perf` [10], a kernel profiler in Linux, for packet transmission and reception. We also add trace points to socket

Table 1. Comparison of ELEMENT with existing TCP-based delay measurement tools (values in seconds).

	Sender's system delay (stdev)	Average network delay (stdev)	Receiver's system delay (stdev)
Ground truth	1.044 (0.374)	0.05	0.027 (0.113)
ELEMENT	1.049 (0.373)	0.056	0.025 (0.101)
tcpping	x	0.056 (0.004)	x
paping	x	0.058 (0.004)	x
hping3	x	0.059 (0.003)	x
echoping	0.657 (0.139)		

write/read in the application. At the TCP layer, we trace the `tcp_transmit_skb, tcp_v4_do_rcv` functions. We calculate the delay between the application and TCP layers by subtracting the timestamps at these two layers.

TCP-based delay measurement tools. We compare ELEMENT to four TCP-based delay measurement tools: `tcpping`[12], `paping`[2], `hping3`[8], and `echoping`[4]. `echoping` periodically measures the total download time of a file from an HTTP server, while the other three tools use TCP control packets to measure the round trip time (RTT), but do not saturate the network to send data packets. Table 1 compares ELEMENT to these tools. We report the average and standard deviation over 15 experimental runs. Three tools only report the RTT of the path, while `echoping` only reports the total end-to-end latency. By contrast, ELEMENT estimates both sender- and receiver- side delays and has an RTT very close to the ground truth latencies.

Ground truth vs. ELEMENT. We now plot ELEMENT's latency estimates side by side with the ground truth over time. Figure 6 shows that ELEMENT accurately estimates the delays at both the TCP sender and receiver. While the kernel-level `perf` profiler captures the exact time when each traced function is called, ELEMENT cannot as it only retrieves the TCP statistics periodically. Thus, we find the error in ELEMENT's delay estimation by interpolating between the ground truth values closest to the time of ELEMENT's measurement. We can then observe how the delay varies. Since the packets at the TCP send buffer are processed in a batch (e.g., TCP segmentation offload), the observed sender delays rapidly oscillate. Packet loss events also affect the observed delays: when a loss occurs, the sender delay increases significantly since other packets accumulate in the send queue while the lost packet is retransmitted. Similarly, at the receiver, packets received after a lost packet must wait in the out-of-order queue until the lost packet is received. Despite these frequent delay variations due to batch processing and packet loss, ELEMENT achieves high measurement accuracy as shown in Figure 6c.

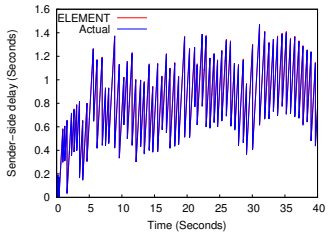
Different network environments. We evaluate the accuracy of ELEMENT's delay estimation in different network environments (Figure 7). We obtained the measurement errors by comparing with the kernel profiler explained above. We first fix the RTT in our testbed to 50 ms and vary the network bandwidth from 30 Mbps to 200 Mbps in Figures 7a to 7d, and fix the bandwidth at 10 Mbps and vary the RTT from 10 ms to 200 ms in Figures 7e to 7h. We also evaluate ELEMENT in LAN, cable (using Motorola DCT700/US modem), WiFi (same setting as Section 2.2), and LTE (same setting as Section 2.2) production networks in Figures 7i to 7l. ELEMENT's receiver side delay estimation achieves 95% accuracy across all environments. Its sender-side accuracy is 90% over all environments, with a higher accuracy for higher bandwidths: a higher bandwidth indicates smaller inter-packet delays, which improves accuracy. There is no clear correlation of RTT with estimation accuracy. The sender-side estimation accuracy also varies for different types of production networks, mostly due to their differences in bandwidth. We confirmed that the large errors in some cases are because the delay varies significantly in a short period of time. Note that ELEMENT records the transmitted or received bytes every 10 msec by default. If we decrease this measurement interval we can obtain higher accuracy. However, even with 10 msec interval, ELEMENT estimates the overall variations of the delay quite accurately for all network types.

Dynamic network environments. To see the impact of network dynamics on ELEMENT's estimation accuracy, we test two separate scenarios: first, we vary the network bandwidth every 20 seconds between 10 Mbps and 50 Mbps (dynamic bandwidth); and second, we introduce three background flows every 20 seconds. As shown in Figure 8, ELEMENT is accurate in both scenarios, but more accurate with background traffic. We presume this is because ELEMENT's accuracy depends on the available network bandwidth, which is lower in the dynamic scenario when the bandwidth is limited to 10Mbps.

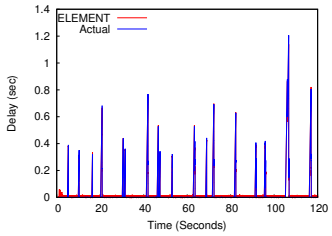
4.4 Minimizing end-to-end latency

As shown above, the TCP send/receive buffers can have high latencies, leading to an end-to-end application layer latency of up to several seconds. To minimize this latency, we take two approaches. In the first approach, we let applications control their data rates according to the information that ELEMENT provides. New emerging applications (e.g., real-time video, VR, and AR) can benefit from this approach; whenever a data packet is sent or received, recent buffer delay information is retrieved, and applications can adjust their data rate, e.g., through their video resolutions and encoding schemes. Applications such as 360-degree video streaming and SVC (Scalable Video Coding) video streaming can benefit from efficient bandwidth utilization by dropping unnecessary data in the application buffer right before they are sent to the TCP layer depending on the ROI (region of interest) change or network bandwidth variation, respectively.

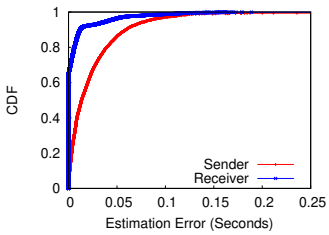
The second approach is for legacy TCP applications that we do not want to change, or do not have an access to their source code. For this, we design an algorithm to minimize the latency at end-to-end TCP connections, as shown in



(a) The ground-truth and measured TCP sender's delay.

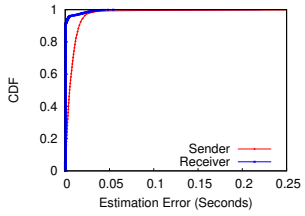


(b) The ground-truth and measured TCP receiver's delay.

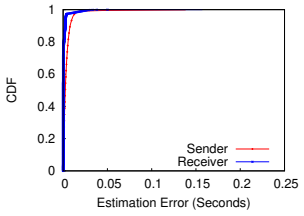


(c) The CDF of ELEMENT's estimation error.

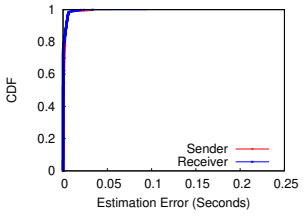
Figure 6. The comparison between the delay ground-truths and ELEMENT's delay estimations on a TCP Cubic flow in the 10 Mbps and 50ms RTT network. ELEMENT precisely tracks latency variations.



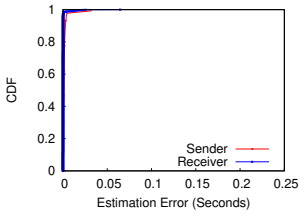
(a) 30 Mbps.



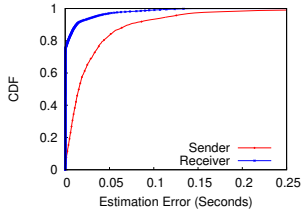
(b) 50 Mbps.



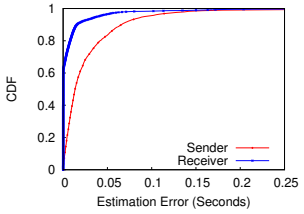
(c) 100 Mbps.



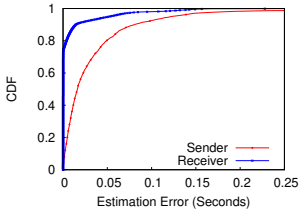
(d) 200 Mbps.



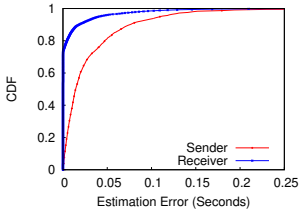
(e) 10 msec RTT.



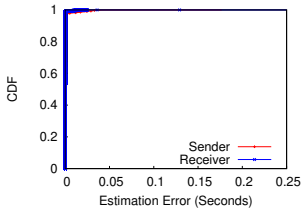
(f) 100 msec RTT.



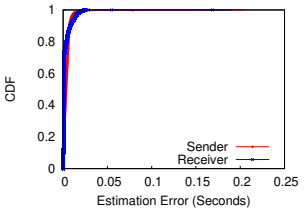
(g) 150 msec RTT.



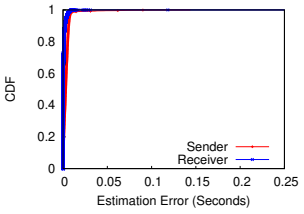
(h) 200 msec RTT.



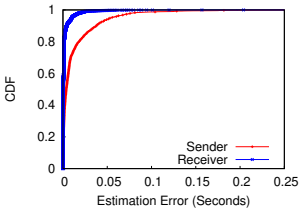
(i) LAN.



(j) Cable network.



(k) WiFi network.



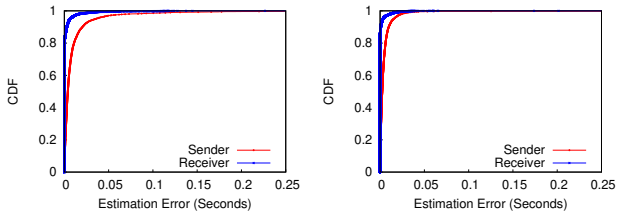
(l) LTE network.

Figure 7. ELEMENT returns accurate latency estimates in all the tested scenarios.

Algorithm 3[4]. ELEMENT uses this as its default latency minimization algorithm, which we evaluate in Section 5.1, though users can override it with their own algorithm. The idea is to adaptively control the number of packets in the TCP send buffer according to the buffer delay variation. If the average delay at the send buffer increases (respectively decreases), we decrease (increase) the number of packets that are permitted to stay in the TCP send buffer. To enforce this limit, we control the rate at which the packets are sent to the TCP layer by putting a sleep operation after the packet send operation. This approach is not ideal but suitable for most legacy TCP applications. All of these operations can be implemented in the application layer without root privileges, in accordance with our design principles (Section 3.1).

To put these ideas into practice, we control the maximum number of allowed packets in the buffer, defined as S_{target}. We create a thread that periodically checks the buffer delay and sets S_{target} according to the ratio of the delay to the pre-determined delay threshold D_{thr}. We limit S_{target} so that it does not exceed the amount that corresponds to the congestion window size in the TCP layer. The sleep operation

[4]For Algorithm 3, we assume a legacy application that uses a shallow application buffer. With a deep buffer, the delay can be simply moved to the application buffer from TCP send buffer.



(a) Dynamic bandwidth. **(b)** Background traffic.

Figure 8. ELEMENT's estimation errors in CDF for dynamic network scenarios. ELEMENT maintains high accuracy in dynamically changing network conditions.

Algorithm 3: Algorithm for minimizing the latency at end-to-end TCP connections.

$cwnd \leftarrow$ // congestion window
$snd\_mss \leftarrow$ // sender maximum segment size
$D_{thr} \leftarrow 0.025$ // delay threshold for the rate control
$D_{avg} \leftarrow$ // average delay at TCP
$S_{target} \leftarrow 0$ // target data amount in send buffer
$T_{cur} \leftarrow$ // current time
$T_{last} \leftarrow$ // last time that the algorithm calulated S_{target}
$SRTT \leftarrow$ // smoothed RTT

When the delay at send buffer is calculated by Algorithm 1:
$D_{measure} \leftarrow$ newly measured buffer delay
$D_{avg} \leftarrow D_{avg} * 7/8 + D_{measure}/8$
At a separate checking thread:
while *true* **do**
  if $T_{cur} - T_{last} > SRTT$ **then**
    if $S_{target} = 0$ **then**
      $S_{target} \leftarrow$ send buffer size obtained by $getsockopt$
    $ratio \leftarrow (D_{avg}/D_{thr})^{\Delta}$
    $S_{target} \leftarrow S_{target}/ratio$
    if $S_{target} > cwnd \cdot snd\_mss \cdot \beta$ **then**
      $S_{target} \leftarrow cwnd \cdot snd\_mss \cdot \beta$
    $T_{last} \leftarrow T_{cur}$
    if *sender's network is LTE or WiFi* **then**
      Set the buffer size to $S_{target} \cdot \gamma$ by using $setsockopt$

After sending a packet using functions such as $send$:
$cnt \leftarrow 0$
$seq \leftarrow$ total sent bytes by application
$B_{est} \leftarrow$ current estimated sent bytes at TCP layer by Algorithm 1
while *true* **do**
  if $cnt > \delta$ *or* $seq - B_{est} \leq S_{target}$ **then**
    break
  $cnt \leftarrow cnt + 1$
  sleep for cnt^{λ} msec

is added only when the estimated amount of buffered data is larger than the target amount S_{target}. To prevent excessively delayed return of write calls due to sleep, we limit the total number of sleep operations for one write call and increase the sleep time as the number of sleep operation increases.

Algorithm 3 can be seen as an application-layer rate control analogue to FAST TCP [57], a delay-based TCP congestion control algorithm. Based on algorithm 3,

$$S_{target} = \min\left(\beta \cdot cwnd, \left(\frac{D_{thr}}{D_{avg}}\right)^{\Delta} S_{target}\right), \qquad (1)$$

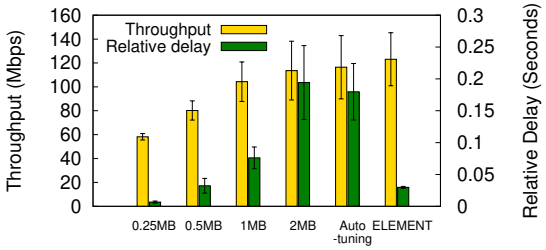


Figure 9. Comparison of the average throughput and delay of three different approaches – set fixed buffer sizes, Linux's default buffer auto-tuning, and ELEMENT's algorithms.

where $cwnd$ denotes the congestion window size, and D_{avg} denotes the measured average delay, respectively. FAST TCP updates the window size w based on $w \leftarrow \min\{2w, (1-\gamma)w + \gamma(\frac{\text{baseRTT}}{\text{RTT}}w + \alpha)\}$. With $\gamma = 1$, FAST's main difference from ELEMENT is that since $\Delta < 1$, the window size increase and decrease is smoother in our algorithm. ELEMENT uses the parameter values $\Delta = 0.25$, $\beta = 2.1$, $\gamma = 1.1$, $\delta = 8$, and $\lambda = 1.5$.

One might think that properly setting the buffer size can achieve both the high throughput and low latency without any latency control algorithm as ours. Figure 9 shows that it is not true. We set two servers in Amazon EC2, and measure the average throughput and delay by changing the send buffer size to 0.25, 0.5, 1, 2MB. We also compare with the Linux default auto-tuning and ELEMENT algorithms. We can observe that with any static buffer size, we cannot achieve both the high throughput and low latency at the same time: if we increase the buffer size, the throughput is increased, but the delay is also increased, while when we decrease the buffer size, the delay is decreased, but the throughput is also decreased. On the other hand, with ELEMENT we can achieve the high throughput and low latency at the same time.

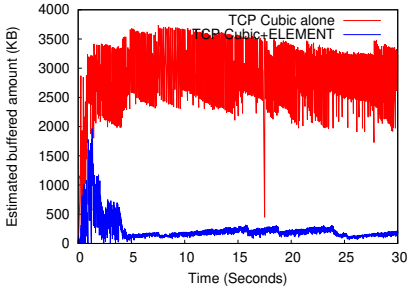


Figure 10. Estimated amount of buffered packets for CUBIC alone and CUBIC + ELEMENT flows. ELEMENT minimizes the amount of buffered data without exhausting the buffer.

Figure 10 shows an example variation of estimated amount of buffered packets for a CUBIC flow and CUBIC + ELEMENT flow between two severs in Chameleon Cloud [39].

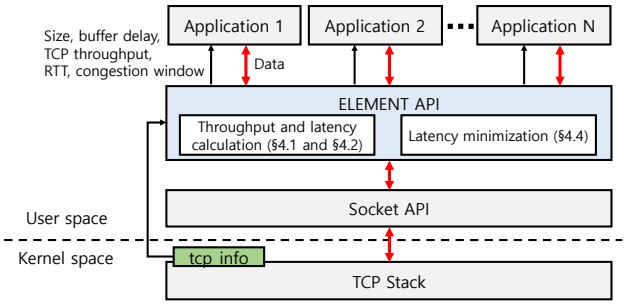


Figure 11. Architecture of the ELEMENT API, which sits between the application and socket API. ELEMENT API returns the size of written/read bytes, buffer delay, throughput, RTT, and congestion window and runs sender- and receiver- side delay estimation algorithms and a latency minimization algorithm.

```
typedef struct retinfo {
        ssize_t size;
        float buf_delay;
        float throughput;
        float rtt;
        int cwnd;
} retinfo;

// inititlization
int init_em(bool is_wireless, int algorithm);

// wrapper functions for various send/recv functions
retinfo em_send(int fd, const void *buf, size_t len,
        int flags);
retinfo em_write(int fd, void *buf, size_t count);
retinfo em_read(int fd, void *buf, size_t count);
...

// close
void fin_em();
```

Figure 12. Socket wrapper type ELEMENT APIs. We provide a set of wrapper functions for various data sending and receiving functions at the socket layer.

We can observe that the CUBIC flow keeps an excessively large amount of buffered data. On the other hand, ELEMENT keeps the amount of buffered data as small as possible as long as it does not exhaust the buffer. This explains why ELEMENT does not reduce the throughput while minimizing the latency as we can see in Section 5.

4.5 ELEMENT APIs

Figure 11 shows the architecture of the ELEMENT APIs, which lie between the application and socket API. These APIs estimate the buffer delay, calculate throughput, and run the latency minimization algorithm introduced above.

The socket wrapper ELEMENT APIs (shown in Figure 12) contain a set of wrapper functions for various data sending and receiving functions at the TCP socket layer. To ensure BSD socket compatibility, we use the same arguments for each wrapper function as for the original function. The wrapper functions return information that includes the size of the written bytes, measured buffer delay, throughput at the TCP layer, RTT, and congestion window. Note that the original send and receive functions return the size of the written bytes. This information can be used to control the application layer data rate, e.g., its video bitrate or encoding scheme, in order to adjust the application latency. With these APIs, new application developers can use ELEMENT for their applications to optimize its latency.

On the other hand, ELEMENT also supports legacy TCP applications by providing a dynamic shared library which overrides the BSD socket library and runs the default latency minimization algorithm (Algorithm 3) from the user level. These legacy TCP applications do not require any changes on the source or their binary executables; they need only set the LD_PRELOAD variable to load the ELEMENT library before running a TCP application.

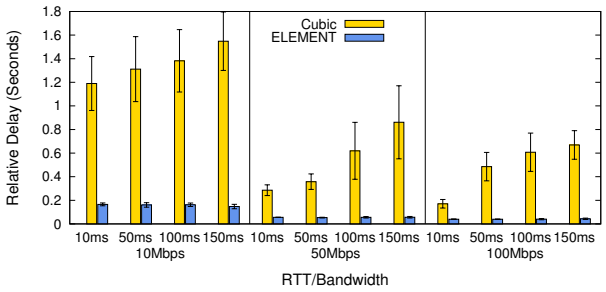
5 Evaluation

We evaluate the efficacy of ELEMENT in two different application scenarios. We first demonstrate that ELEMENT can be easily integrated with legacy TCP applications, without any modification on their source code or executables. Next, we integrate ELEMENT with a new, real-time VR application that exploits the information offered by ELEMENT to control its latency.

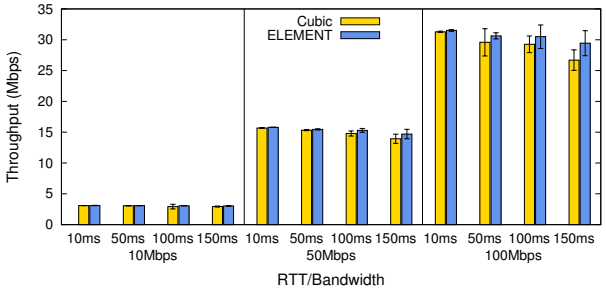
5.1 Impact on legacy TCP applications

To see the impact of ELEMENT on legacy TCP applications, we use `Iperf`, a popular open source tool for measuring network performance. It continuously sends data to measure TCP performance, which is common in legacy TCP applications. We run `Iperf` with ELEMENT by using dynamic preloading and the default latency control algorithm introduced in Section 4.4. In each case, we display the average and standard deviation of the performance over 15 experimental runs.

Controlled environments. Figure 13 compares the flow performance before and after ELEMENT's integration with `Iperf` running a TCP Cubic flow. We vary the network bandwidth and RTT. We first run three Cubic flows and then replace one Cubic flow with an ELEMENT flow. The other two flows serve as competing background TCP flows with which we can evaluate ELEMENT's fairness. ELEMENT significantly reduces the latency of its TCP Cubic flow by up to 10 times in almost all conditions, while barely decreasing or even slightly increasing the flow throughput. The throughput of the other two TCP Cubic flows remains almost the same, indicating that ELEMENT does not interfere with TCP fairness.

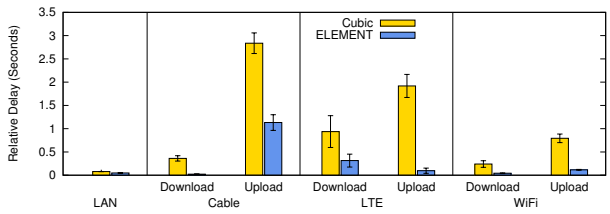


(a) Latency is significantly reduced by up to 10x.

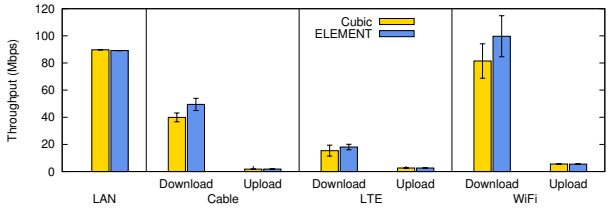


(b) Throughput and fairness across flows is maintained.

Figure 13. Impact of ELEMENT over networks with different bandwidths and RTTs. We run three TCP Cubic flows and replace one with an ELEMENT flow.



(a) Latency is significantly reduced by 4x to 10x.



(b) Throughput is maintained or slightly improved.

Figure 14. Impact of ELEMENT on a TCP Cubic flow on four different production networks.

Production networks. We evaluate ELEMENT on four different production networks, including LAN, cable networks (same setting as Section 4.3), WiFi (same setting as Section 2.2), and LTE (same setting as Section 2.2). Similar to our previous experiments, we first run two TCP Cubic flows and then replace one Cubic flow with an ELEMENT flow. Figure 14 shows our results. While ELEMENT barely reduces latency on the LAN as the RTT is already very small (less than 2ms), ELEMENT's impact on the other networks is significant. The delay is significantly reduced compared to Cubic (e.g., a 10x latency decrease in LTE upload) even though the overall delay reduction is smaller (4 to 10 times) than in the controlled experiments (Figure 13). The throughput is not decreased and slightly increased in some cases as on the wired network (with up to a 22.3% increase for WiFi uploads).

Latency-optimized TCP. The recently introduced BBR protocol from Google aims to minimize latency while achieving high throughput. It is unclear, however, if BBR can effectively detect and minimize the latency incurred at the end host as well as in the network. It is also unclear whether ELEMENT will have a significant impact if it runs on top of delay-based TCP protocols in the TCP stack. To answer these questions, we measure the latency of flows using TCP Cubic, Vegas [18], and BBR [19], as well as flows that use each protocol plus ELEMENT. We run a single flow for each experiment in our wired testbed with 50 Mbps bandwidth and 50ms RTT. Figure 15 shows the average and standard deviations of the sender-side host delay, network delay, and receiver-side host delay. BBR's latency control incurs the largest delay at both the sender and the receiver[5]. TCP Vegas achieves a smaller sender-side delay and RTT than Cubic and BBR. ELEMENT maintains Vegas's low sender- and receiver-side delays, reducing its sender-side delay even further, and reduces BBR's sender- and receiver-side delays.

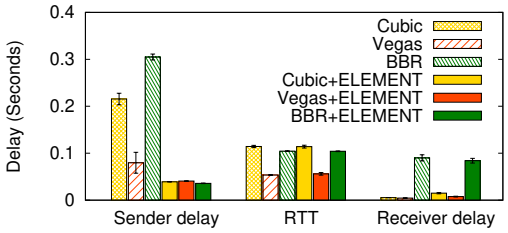
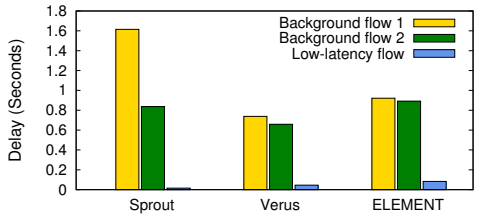


Figure 15. When used in combination with TCP flows, ELEMENT effectively removes the endhost latency, while TCP Cubic and TCP BBR suffer from large latency at the sender.

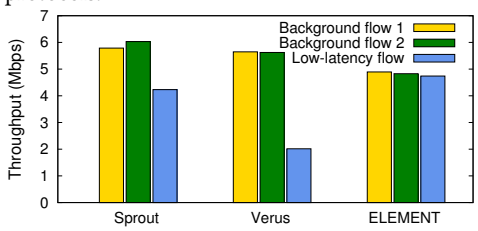
Comparison with UDP-based low-latency protocols. Recently, several low-latency protocols utilizing UDP (e.g., Sprout [58] and Verus [59]) have been proposed. In this experiment, we compare Sprout and Verus with ELEMENT. We run a single flow for each tested algorithm (the "low-latency" flow in Figure 16a), with two background TCP Cubic flows. We dynamically vary the bandwidth, RTT, and loss rate. Figure 16a shows that Sprout and Verus obtain very low latency, and ELEMENT achieves a higher but comparable latency. However, Sprout and Verus exhibit poor throughput fairness, as shown in Figure 16b, while ELEMENT benefits from TCP's

[5]Our evaluation is based on the BBR implementation in Linux 4.12.10, which may perform worse than the latest version (BBR v2.0). BBR is not designed to remove latency at the end host, and thus this increased delay may stem from Linux's default buffer tuning algorithms in the TCP stack.

fairness. We presume their latency reduction mechanisms make their bandwidth estimation too conservative, leading to low throughput.



(a) ELEMENT maintains comparable latency to UDP-based low-latency protocols.



(b) ELEMENT maintains the throughput fairness, while UDP-based low-latency protocols do not.

Figure 16. UDP-based low-latency protocols maintain very low latency but poor fairness. ELEMENT provides a slightly higher latency, but keeps the throughput fairness. The "low-latency" flow in the figure uses Sprout, Verus, or ELEMENT.

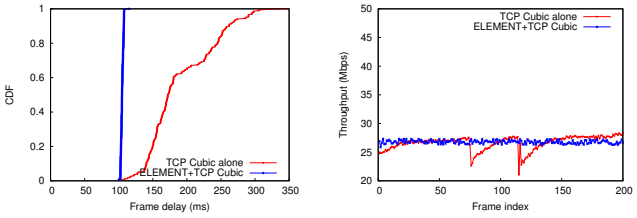
5.2 Impact on virtual reality applications

We finally design and implement a real-time virtual reality (VR) application using TCP to show the effectiveness of ELEMENT. Figure 17 shows the application scenario and its system architecture. A user wears a VR headset and watches a 360 degree streaming video. When the user moves his or her head to see a certain part of the video, this control information (e.g., the viewpoint of a user expressed in x- and y-coordinates, and the speed of head movement) is sent back to the server. The server then sends more frames corresponding to this viewpoint after rendering and encoding. The user's VR headset (or the VR app on a headset-connected computer) receives and decodes the frames to show the user updated scenes. We use `GPUJPEG` [7] encoding on a `feh` [5] image display. `GPUJPEG` is set to use `CUDA` [3], which runs on a GeForce GTX 970 graphic card [6].

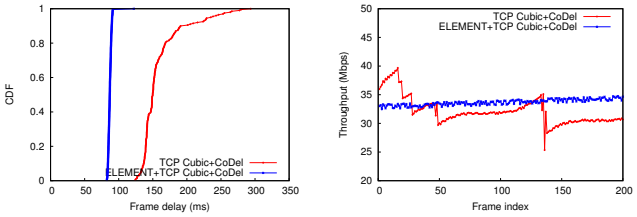
In VR applications, delivering video frames within a constant threshold (e.g., an additional 100ms) upon headset movement is important to prevent VR sickness or cognitive dissonance [25], reinforcing the need for strict latency control. Our VR application server thus needs more visibility on when and where the latency is added, which is provided by the ELEMENT framework as a set of API calls. In particular, the server uses the congestion window, system delay, and RTT to determine how many frames and at what resolutions they should enter the encoder buffer. It also checks the

Figure 17. 360 degree VR application scenario.

change of the sender-side system delay upon sending each frame and discards frames to reduce this latency if needed; the server increases the number of frames if it does not see any increase in the sender-side delay.



(a) Cubic vs. ELEMENT + Cubic.



(b) Cubic + CoDel vs. ELEMENT + Cubic + CoDel.

Figure 18. Frame delay and throughput in 360 degree video streaming with and without ELEMENT.

Figure 18 shows the frame delays and throughput with and without ELEMENT on TCP Cubic and TCP Cubic with CoDel, respectively. We use two algorithms - one for loss-based TCP Cubic and the other for TCP Cubic with the CoDel AQM scheme, which helps prevent bufferbloat, and evaluate their performance before and after their integration with ELEMENT. As shown in Figures 18a and 18b, ELEMENT helps both protocols tightly control their system latencies so that almost no packets deviate from the base latency. Without ELEMENT, more than 40% of frames miss this VR sickness deadline (200ms, considering the 100ms VR sickness threshold added to the base latency) with TCP Cubic, and around 10% of frames miss the deadline for TCP Cubic with CoDel, though the latter has less latency variation. Furthermore, the constant throughput of ELEMENT indicates that our VR application effectively leverages ELEMENT's APIs to control

its latency and rate, while the other two algorithms (TCP Cubic and TCP Cubic with CoDel) grab time-varying available bandwidth in the network, which may not be best for real-time video streaming.

6 Related work

Diagnosing latency: A large body of work measures network characteristics like RTT, one-way delay, and bandwidth [31, 33, 42, 48, 52, 53, 55]. Delay measurement tools such as `ping` and `traceroute` measure network layer delays using ICMP, while TCP-specific tools such as `tcpping`[12], `paping`[2], `hping3`[8], `echoping`[4] and `tcptraceroute`[14] measure TCP or application layer delays. Tools for measuring one-way delay are particularly related to ELEMENT, but no existing works can decompose the endhost and network delays. For instance, [33] estimates the one-way delay between nodes in the network without time synchronization, while [42] estimates the one-way queuing delay and [55] proposes more accurate one-way delay estimation algorithms. `Tcpsnitch`[13], like ELEMENT, examines interactions between the application and TCP/IP protocol stacks, but focuses on the socket usage, not on flow-level delays. RFC 4898 [43] and ModNet [47] provide applications with various information on the network condition, but do not provide detailed delay information as in ELEMENT. However, some features of RFC 4898 are included in Linux's TCP_INFO implementation.

Other works focus on large-scale Internet services. YTrace [38] provides real-time diagnosis of end-to-end performance across the network and service layers. System component logs [21] can be used to generate performance hypotheses to compare with empirical data. LatLong [60] allows CDNs to diagnose large latency increases with passive measurements of performance, traffic, and routing, while [54] combines application layer logs, network layer packet traces, active probing and transport-layer statistics to find latency sources in CDN servers' network stacks. Yet these approaches depend on multiple data logs and specific system architectures. ELEMENT instead focuses on the latency in general systems monitored by end-users or service and network providers. A similar approach is taken for datacenters in [22], which collects statistics from TCP measurement agents at each virtual machine, but does not consider end-to-end latency.

Bufferbloat solutions: ELEMENT enables bufferbloat solutions that examine delays in the endhost system stack as well as the network. While our experiments in Section 2 show that both delays contribute significantly to bufferbloat, many existing solutions take network-based AQM approaches [44–46, 56], as evaluated in [36, 40, 41, 49], without considering the endhost delay. In particular, CoDel [45] is a parameterless AQM solution that aims to solve the bufferbloat problem. Adaptive CoDel [44] aims at overcoming CoDel's uniform target delays, which may not suitable for real-time video streaming applications. Furthermore, Hybrid FQ-CoDel [56] combines Adaptive CoDel with fair queuing to reduce the queuing delay and optimize video transmission. Host-based approaches control the send buffer [29] or receiver advertisement window [37] to maintain low delay, while [32] focuses on reducing the firmware buffer delay for uploads in cellular networks.

Application-layer pacing: Trickle [24] and Netbrake [9] are technically very similar to ELEMENT's shared library for legacy TCP applications as they also take advantage of the UNIX dynamic preloading capability to interposition between the application and the BSD socket library in the user level. However, they are mainly used to rate-limit the TCP connections by pacing the I/Os on sockets, not to control the system delay on end hosts.

7 Discussion

ELEMENT's latency control: ELEMENT's latency minimization algorithm may appear to reduce the latency in the TCP layer but not the application layer, providing no benefit to the end-to-end user experience. However, ELEMENT provides knobs to observe and control the latency in the application layer itself. Without ELEMENT, the delay in TCP layer is completely hidden from the application.

ELEMENT applications: ELEMENT is especially beneficial for applications that can adapt their data rates. Some applications may need another type of interface with the ELEMENT API; for example, jitter-sensitive applications will benefit from an event-driven interface like `select()`. The application can then react as soon as the jitter exceeds a given threshold. ELEMENT currently provides a default latency minimization algorithm in Section 4.4, but other applications may provide their own rate control algorithms.

Benefits of TCP based real-time applications: Many real-time applications use UDP as their transport protocol, but low-latency TCP applications can provide several benefits. They do not need complex mechanisms for NAT/firewall traversal, and as we show in Section 5, existing TCP congestion control algorithms also preserve fairness among TCP flows, with our efforts concentrated on the latency control.

Coexistence with congestion control algorithms in the TCP stack: We confirm that ELEMENT's latency minimization algorithm does not sacrifice the throughput of other flows, even increasing it in some settings. However, more work is needed on the coexistence of ELEMENT with other TCP congestion control algorithms, especially delay-based ones. We expect ELEMENT will fairly share resources with other TCP flows, since it only controls the amount of data in the TCP stack's send queue, not the sending rate itself.

CPU overhead of ELEMENT: We measure the overhead of our latency minimization algorithm in terms of CPU usage. We run 40 traffic generator processes for 60 seconds on a network with 1 Gbps bandwidth and 50 msec RTT. This

is a heavy workload scenario as the bandwidth delay product of the network is large, meaning that ELEMENT needs to track many packets being sent and received. We measure the sum of all processes' CPU usage with and without ELEMENT. We observe that the average CPU overhead slightly increased with ELEMENT (around 4%), which is quite reasonable. To further optimize the overhead, we also modified the kernel to record the required TCP information on a shared page between kernel and user-space, which could reduce the overhead of `getsockopt` by 1%.

Application-limited TCP applications: Little work has explored the potential of application-limited TCP transfer. ELEMENT could be a good example application as it dynamically limits the amount of data being pushed from the application to the TCP layer to control the latency on the end host. With ELEMENT, an application-limited TCP application can obtain accurate feedback (e.g., throughput, endhost latency, and network latency) upon each transmission, allowing TCP to be used for a much wider range of applications with strict performance requirements.

Extending ELEMENT to different layers and virtualized environment: eBPF is an extension of original Berkeley Packet Filter (BPF), in which packet filters are implemented as programs that run on a register-based virtual machine [17]. eBPF is useful for writing network programs, debugging the kernel, analyzing the performance, etc. ELEMENT can be extended using the framework of eBPF [15]. Using eBPF, we can write a code that inspects the transmitted bytes, timestamps in different kernel functions that traverse different network layers, extending ELEMENT to IP, MAC and physical layers. By inspecting functions in hypervisors, we can also measure the latency incurred in the virtualization layer.

Supporting application-specific delay/jitter requirements: There have been some works that allow applications to explicitly specify their QoS requirements [30, 50]. They can be used for applications to inform ELEMENT of their delay and delay jitter requirement.

8 Conclusion

We introduce ELEMENT, the first user-level framework that decomposes end-to-end application latency into delays in end hosts and in networks. ELEMENT does not require any admin or root privileges, and utilizes only user-level information; we show that this information can be used to accurately infer the end host delays above the transport layer, and that these delays can be a significant component of end-to-end application delays. We implement ELEMENT as a set of APIs that require minimal changes to legacy TCP applications, and we use ELEMENT to reduce the realized latencies in a legacy TCP application, `Iperf`, and a virtual reality application. In our experiments, using ELEMENT to reduce application latency does not reduce, and sometimes increases, the achieved throughput, while preserving fairness with other TCP flows. Our work thus enables TCP applications to proactively control their latencies, which will be needed for emerging interactive applications like video conferencing, tele-medicine, and virtual reality.

Acknowledgements

We thank our shepherd Costin Raiciu and the anonymous reviewers for their insightful comments and feedback. This work was supported by the NSF under Grant CNS-1525435, in part by the Defense Advanced Research Projects Agency (DARPA), under contract No. HR001117C0048, and in part by IITP grants funded by the Korea government (MSIT) (No. 2018-0-00693, Development of An Ultra Low-Latency User-Level Transfer Protocol and No. 2017-0-00562, UDP-based Ultra Low-Latency Transport Protocol with Mobility Support).

References

[1] 2014. The Tactile Internet. *ITU-T Technology Watch Report* (2014).
[2] 2018. Cross-platform TCP port testing, emulating the functionality of ping (port ping). https://code.google.com/archive/p/paping/.
[3] 2018. CUDA. https://www.geforce.com/hardware/technology/cuda.
[4] 2018. echoping - tests a remote host with TCP or UDP. https://linux.die.net/man/1/echoping.
[5] 2018. feh - a fast and light image viewer. https://feh.finalrewind.org/.
[6] 2018. GeForce GTX 970. https://www.geforce.com/hardware/desktop-gpus/geforce-gtx-970.
[7] 2018. GPUJPEG: JPEG compression and decompression accelerated on GPU. https://sourceforge.net/projects/gpujpeg/.
[8] 2018. Hping - Active Network Security Tool. http://www.hping.org/.
[9] 2018. Netbrake. .http://www.hping.org/netbrake/.
[10] 2018. perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page.
[11] 2018. Performance Measurement Tools Taxonomy. https://www.caida.org/tools/taxonomy/performance.xml.
[12] 2018. tcpping: test response times using TCP SYN packets. http://www.vdberg.org/~richard/tcpping.
[13] 2018. Tcpsnitch. https://tcpsnitch.org/.
[14] 2018. tcptraceroute(1) - Linux man page. https://linux.die.net/man/1/tcptraceroute.
[15] 2018. A thorough introduction to eBPF. https://lwn.net/Articles/740157/.
[16] NGMN Alliance. 2015. 5G white paper. *Next Generation Mobile Networks, White paper* (2015).
[17] Daniel Borkmann. 2016. Advanced programmability and recent updates with tc's cls bpf. *tc* 1/13 (2016).
[18] Lawrence S. Brakmo and Larry L. Peterson. 1995. TCP Vegas: End to end congestion avoidance on a global Internet. *IEEE Journal on selected Areas in communications* 13, 8 (1995), 1465–1480.
[19] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2016. BBR: Congestion-based congestion control. *Queue* 14, 5 (2016), 50.
[20] Mung Chiang. 2016. Fog networking: An overview on research opportunities. *arXiv preprint arXiv:1601.00835* (2016).
[21] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F Wenisch. 2014. The mystery machine: End-to-end performance analysis of large-scale internet services. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 217–231.

[22] Selim Ciraci, Boon Thau Loo, Assaf Schuster, and Geoff Outhred. 2016. Taking the Blame Game out of Data Centers Operations with NetPoirot. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*. ACM, 440–453.

[23] L Eggert and Godred Fairhurst. 2008. *Unicast UDP usage guidelines for application designers*. Technical Report.

[24] Marius Eriksen. 2005. Trickle: A Userland Bandwidth Shaper for UNIX-like Systems. In *Usenix Annual Technical Conference*. 61–70.

[25] A. S. Fernandes and S. K. Feiner. 2016. Combating VR sickness through subtle dynamic field-of-view modification. In *2016 IEEE Symposium on 3D User Interfaces (3DUI)*. 201–210. https://doi.org/10.1109/3DUI.2016.7460053

[26] Pedro Garcia Lopez, Alberto Montresor, Dick Epema, Anwitaman Datta, Teruo Higashino, Adriana Iamnitchi, Marinho Barcellos, Pascal Felber, and Etienne Riviere. 2015. Edge-centric computing: Vision and challenges. *ACM SIGCOMM Computer Communication Review* 45, 5 (2015), 37–42.

[27] Jim Gettys and Kathleen Nichols. 2011. Bufferbloat: Dark buffers in the internet. *Queue* 9, 11 (2011), 40.

[28] Jim Gettys and Kathleen Nichols. 2012. Bufferbloat: dark buffers in the internet. *Commun. ACM* 55, 1 (2012), 57–65.

[29] Ashvin Goel, Charles Krasic, and Jonathan Walpole. 2008. Low-latency adaptive streaming over TCP. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)* 4, 3 (2008), 20.

[30] Xiaohui Gu, Klara Nahrstedt, Wanghong Yuan, Duangdao Wichadakul, and Dongyan Xu. 2002. An XML-based quality of service enabling language for the web. *Journal of Visual Languages and Computing* 13, 1 (2002), 61–95.

[31] Cesar D Guerrero and Miguel A Labrador. 2010. On the applicability of available bandwidth estimation techniques and tools. *Computer Communications* 33, 1 (2010), 11–22.

[32] Yihua Guo, Feng Qian, Qi Alfred Chen, Zhuoqing Morley Mao, and Subhabrata Sen. 2016. Understanding On-device Bufferbloat for Cellular Upload. In *Proceedings of the 2016 ACM on Internet Measurement Conference*. ACM, 303–317.

[33] Omer Gurewitz, Israel Cidon, and Moshe Sidi. 2006. One-way delay estimation using network-wide measurements. *IEEE/ACM Transactions on Networking (TON)* 14, SI (2006), 2710–2724.

[34] Sangtae Ha, Injong Rhee, and Lisong Xu. 2008. CUBIC: a new TCP-friendly high-speed TCP variant. *ACM SIGOPS Operating Systems Review* 42, 5 (2008), 64–74.

[35] Toke Hoeiland-Joergensen, P McKenney, J Gettys, and E Dumazet. 2016. The flowqueue-codel packet scheduler and active queue management algorithm. *IETF Draft, March* 18 (2016).

[36] Ilpo Järvinen and Markku Kojo. 2014. Evaluating CoDel, PIE, and HRED AQM techniques with load transients. In *39th Annual IEEE Conference on Local Computer Networks*. IEEE, 159–167.

[37] Haiqing Jiang, Yaogong Wang, Kyunghan Lee, and Injong Rhee. 2016. DRWA: a receiver-centric Solution to Bufferbloat in Cellular Networks. *IEEE Transactions on Mobile Computing* 15, 11 (2016), 2719–2734.

[38] Partha Kanuparthy, Yuchen Dai, Vahid Fatourehchi, Sudhir Pathak, Sambit Samal, and PPS Narayan. 2016. YTrace: End-to-end Performance Diagnosis in Large Content Providers. *arXiv preprint arXiv:1602.03273* (2016).

[39] Kate Keahey, Pierre Riteau, Dan Stanzione, Tim Cockerill, Joe Mambretti, Paul Rad, and Paul Ruth. 2018. Chameleon: a Scalable Production Testbed for Computer Science Research. In *Contemporary High Performance Computing: From Petascale toward Exascale* (1 ed.), Jeffrey Vetter (Ed.). Chapman & Hall/CRC Computational Science, Vol. 3. CRC Press, Boca Raton, FL, Chapter 5.

[40] Naeem Khademi, David Ros, and Michael Welzl. 2014. The new AQM kids on the block: An experimental evaluation of CoDel and PIE. In *Computer Communications Workshops (INFOCOM WKSHPS), 2014 IEEE Conference on*. IEEE, 85–90.

[41] Nicolas Kuhn, Emmanuel Lochin, and Olivier Mehani. 2014. Revisiting old friends: is CoDel really achieving what RED cannot?. In *Proceedings of the 2014 ACM SIGCOMM workshop on Capacity sharing workshop*. ACM, 3–8.

[42] Wei-Zhou Lu, Wei-Xuan Gu, and Shun-Zheng Yu. 2009. One-way queuing delay measurement and its application on detecting DDoS attack. *Journal of Network and Computer Applications* 32, 2 (2009), 367–376.

[43] Matt Mathis, John Heffner, and Rajiv Raghunarayan. 2007. *TCP extended statistics MIB*. Technical Report.

[44] S Matilda, B Palaniappan, and P Thambidurai. 2013. Bufferbloat Mitigation for Real-time Video Streaming using Adaptive Controlled Delay Mechanism. *International Journal of Computer Applications* 63, 20 (2013).

[45] Kathleen Nichols and Van Jacobson. 2012. Controlling queue delay. *Commun. ACM* 55, 7 (2012), 42–50.

[46] Rong Pan, Preethi Natarajan, Chiara Piglione, Mythili Suryanarayana Prabhu, Vijay Subramanian, Fred Baker, and Bill VerSteeg. 2013. PIE: A lightweight control scheme to address the bufferbloat problem. In *High Performance Switching and Routing (HPSR), 2013 IEEE 14th International Conference on*. IEEE, 148–155.

[47] Sharvanath Pathak and Vivek S Pai. 2015. ModNet: A modular approach to network stack extension. In *NSDI*. 425–438.

[48] Ravi Prasad, Constantinos Dovrolis, Margaret Murray, and KC Claffy. 2003. Bandwidth estimation: metrics, measurement techniques, and tools. *IEEE network* 17, 6 (2003), 27–35.

[49] Dipesh M Raghuvanshi, B Annappa, and Mohit P Tahiliani. 2013. On the effectiveness of CoDel for active queue management. In *Advanced Computing and Communication Technologies (ACCT), 2013 Third International Conference on*. IEEE, 107–114.

[50] Timothy Roscoe and Gene Bowen. 1999. Script-driven packet marking for quality-of-service support in legacy applications. In *Multimedia Computing and Networking 2000*, Vol. 3969. International Society for Optics and Photonics, 166–177.

[51] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. 2009. The case for vm-based cloudlets in mobile computing. *IEEE pervasive Computing* 8, 4 (2009), 14–23.

[52] Jacob Strauss, Dina Katabi, and Frans Kaashoek. 2003. A measurement study of available bandwidth estimation tools. In *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*. ACM, 39–44.

[53] Stephen D Strowes. 2013. Passively measuring TCP round-trip times. *Commun. ACM* 56, 10 (2013), 57–64.

[54] Peng Sun, Minlan Yu, Michael J Freedman, and Jennifer Rexford. 2011. Identifying performance bottlenecks in CDNs through TCP-level monitoring. In *Proceedings of the first ACM SIGCOMM workshop on Measurements up the stack*. ACM, 49–54.

[55] Ahmad Vakili and Jean-Charles Gregoire. 2012. Accurate one-way delay estimation: Limitations and improvements. *IEEE Transactions on Instrumentation and Measurement* 61, 9 (2012), 2428–2435.

[56] V Vimaladevi. 2016. Optimization of Video Streaming through Bufferbloat Mitigation using Hybrid FQ-CoDel Algorithm. *International Journal of Computer Applications* 142, 12 (2016).

[57] D. X. Wei, C. Jin, S. H. Low, and S. Hegde. 2006. FAST TCP: Motivation, Architecture, Algorithms, Performance. *IEEE/ACM Transactions on Networking* 14, 6 (Dec 2006), 1246–1259. https://doi.org/10.1109/TNET.2006.886335

[58] Keith Winstein, Anirudh Sivaraman, and Hari Balakrishnan. 2013. Stochastic forecasts achieve high throughput and low delay over cellular networks. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 459–472.

[59] Yasir Zaki, Thomas Pötsch, Jay Chen, Lakshminarayanan Subramanian, and Carmelita Görg. 2015. Adaptive congestion control for unpredictable cellular networks. In *ACM SIGCOMM Computer Communication Review*, Vol. 45. ACM, 509–522.

[60] Yaping Zhu, Benjamin Helsley, Jennifer Rexford, Aspi Siganporia, and Sridhar Srinivasan. 2012. LatLong: Diagnosing wide-area latency changes for CDNs. *IEEE Transactions on Network and Service Management* 9, 3 (2012), 333–345.