Fine-Grained RTT Monitoring Inside the Network

Satadal Sengupta, Hyojoon Kim, Jennifer Rexford
Princeton University
{satadals,hyojoonk,jrex}@cs.princeton.edu

ABSTRACT

High-speed programmable switches (e.g., Intel Tofino) make it possible for the network to monitor Quality of Experience (QoE) and react quickly to improve performance. Round-trip time (RTT) is a central metric that influences end-user's QoE. In this paper, we argue that an inline, real-time, and fine-grained RTT measurement system—called *P4RTT*—can enable automated QoE monitoring and network adaptation (e.g., via changes in routing, scheduling, etc.) inside the network. However, *P4RTT* is fraught with challenges; the vagaries of the TCP protocol and the resource constraints in high-speed switches make accurate RTT measurement difficult. We discuss solution strategies to address these challenges and present early results on an anonymized campus trace collected using our *P4Campus* testbed. *P4RTT*, with very limited memory, is able to match the performance (collects 98% of the RTT samples) of a baseline (a variant of *tcptrace*) that has access to unlimited memory.

1 INTRODUCTION

The advent of commodity programmable switches (e.g., the Intel Tofino [1]) and the P4 programming language [3] opens up promising opportunities for in-network QoE monitoring and network adaptation [14]. Round-trip time (RTT) is one of the central components of end-user QoE. RTT relates directly to TCP throughput and also heavily influences higher-level metrics such as video QoE, page load time, and so on [2, 4]. Monitoring and minimizing RTTs is especially critical for latency-sensitive applications like interactive video, multi-player online gaming, and algorithmic trading in stock markets [16]. RTT monitoring in software is computationally expensive and is therefore unsuitable for networks with high traffic volume. Instead, an RTT monitoring system deployed in the data plane of an on-path programmable switch can trigger routing and scheduling changes automatically in reaction to RTT anomalies (e.g., steady increase during video streaming). Consider the following examples:

- Network congestion can cause an increase in video startup delay and a decrease in video resolution; increasing RTT is an indicator of such a change [4]. The switch could dynamically reroute traffic to an alternate, less congested path when video QoE starts to decline;
- Anycast-based CDN replica selection could be done dynamically by the switch based on evolving user QoE CDN replicas are often selected using IP anycast to allow for dynamic adaptation to the network conditions. The switch could detect congestion enroute to a selected CDN replica based on high RTTs, and reroute to a replica with lower latency;
- In a WiFi network, the switch could trigger hand-offs among in-range access points based on an end-user's QoE; the QoE here is estimated using RTTs between access points and end-users.

Furthermore, in-network QoE adaptation obviates the need to modify client and server end-points or applications.

Our vision is to design a system that can monitor on-path RTTs in real time, detect a decline in a user's QoE (e.g., by identifying steadily increasing RTTs), and adapt the network rapidly to improve QoE (e.g., by changing routing and scheduling policies). We are implementing this system—called *P4RTT*—in the P4 language and deploying it on our campus network *P4Campus* [9] to demonstrate feasibility. To be effective in practice, *P4RTT* needs certain properties:

(1) Passive measurement: Many popular measurement tools use *active* probes such as ICMP pings to estimate the RTT to remote hosts (e.g., iperf3 [7] and RIPE Atlas [10]). Since QoE depends on application-specific RTTs, probe-based RTT estimates do not suffice. Instead, *P4RTT* should measure RTTs *passively* by observing the actual user traffic [8].

(2) Continuous measurement: Many existing passive RTT monitoring techniques estimate a flow's RTT based only on the TCP three-way handshake [6]. This approach is inaccurate for long flows spanning minutes to hours (e.g., video streaming) since RTTs can vary significantly during the lifetime of the connection. Also, handshake RTTs tend to be smaller than the average RTT of the connection [6]. Therefore, *P4RTT* must monitor RTTs continuously, beyond the initial handshake.

(3) Accurate measurement: Continuous RTT measurement involves matching data packets with their corresponding acknowledgments (ACKs) [5]. However, the vagaries of the TCP protocol—including retransmissions and reordering—can make some RTT samples inaccurate. *P4RTT* should operate correctly even under such conditions.

(4) Efficient operation: High-speed data planes impose significant constraints on arithmetic operations, memory size, the number of pipeline stages, and recirculation bandwidth.

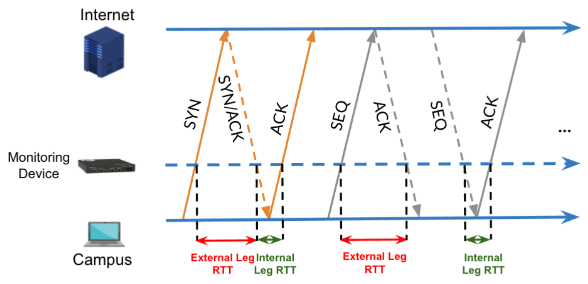

Figure 1: Continuous RTT measurement at a monitoring device by matching TCP data (SEQ) packets with corresponding acknowledgement (ACK) packets.

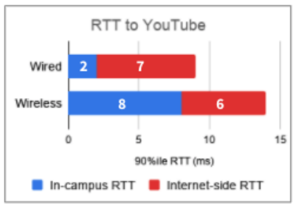

Figure 2: Comparison between the share of internal leg (in-campus) RTT vs. external leg (Internet-side) RTT for YouTube in wired vs. wireless networks.

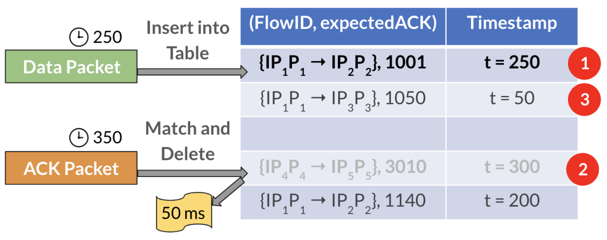

Figure 3: Strawman design: A hash table with flow ID + expected ACK as key and timestamp as value. Arrival of a data (SEQ) packet causes insertion into the hash table, whereas arrival of an ACK triggers deletion of the matching SEQ entry and collection of an RTT sample.

Despite these constraints, *P4RTT* should scale to high traffic rates, by collecting a representative RTT distribution even under heavy load or adversarial traffic (e.g., SYN floods).

In the rest of this paper, we discuss our solution strategies for *P4RTT* and highlight some encouraging initial results. We conclude by discussing promising future directions.

2 CONTINUOUS RTT MONITORING

TCP carries a bi-directional data stream between two end-hosts; bytes in one direction are acknowledged in the other by appropriately setting sequence and acknowledgment numbers in the TCP header. A monitoring device placed strategically (§2.1) can leverage its location to continuously monitor RTTs by matching data and ACK packets (§2.2).

2.1 Seeing Both Directions of the Traffic

Since *P4RTT* relies on matching data packets with corresponding ACKs, it needs to be deployed to a device that can "see" both sides of the traffic, i.e., sender (e.g., client inside the campus) to receiver (e.g., web server on the Internet) and vice-versa. As indicated in Fig. 1, we denote the direction of the TCP data segment as the *SEQ* (sequence) direction, and the direction of the acknowledgment segment as the *ACK* direction. The RTT computed for a SEQ/ACK pair between the monitoring device and the Internet constitutes the *external leg* of the RTT, whereas that between the monitoring device and the client within the campus constitutes the *internal leg*. Fig. 2 illustrates the utility of such an arrangement; we found that for the 90^{th} percentile RTT to YouTube from the Princeton campus, the internal leg of the campus wireless network contributes 57% (8/14 ms) of the total RTT whereas the internal leg of the wired network contributes only 22% (2/9 ms). This reveals that the campus wireless infrastructure adds significant latency to YouTube traffic.

This setup can in principle be extended to include multiple monitoring devices along the path of the traffic, thus providing the operator with the unique ability to divide the total client-server RTT into multiple fine-grained components (e.g., WiFi user to access point, access point to wireless controller, wireless controller to firewall, and so on). Such a design allows the operator to identify the precise component of the end-to-end path that is causing a particular drop in user QoE, and address it accordingly.

2.2 Matching Data Packets with ACKs

Continuous RTT measurement requires storing some SEQ packet state until a matching ACK packet arrives. We need a data structure that stores this SEQ packet state as the *key* and the packet timestamp as the corresponding *value*. When a matching ACK packet arrives, we lookup the SEQ entry using the key, and subtract the entry timestamp from the ACK timestamp to compute the RTT sample. The key should identify a packet uniquely, i.e., it should be composed of a unique flow identifier (the 4-tuple of client and server IP addresses and TCP port numbers) and a unique packet identifier within the flow (the expected ACK number or eACK). Thus an entry in the data structure looks like:

$$S_{packet}[\{sIP, dIP, sPort, dPort, eACK\}] = t_{SEQ}$$

Fig. 3 illustrates this *strawman design*. The entry tagged 1 (white literal within a red circle) shows the insertion of packet information into a hash table on the arrival of a SEQ

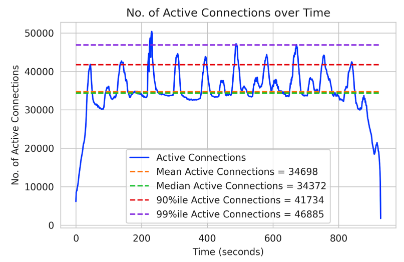

Figure 4: ≈49K active connections seen in a second in the campus trace during peak traffic.

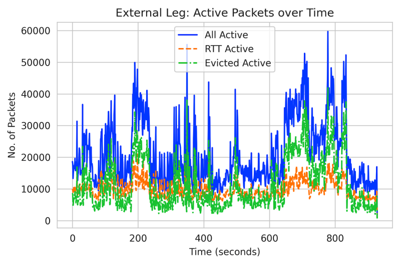

Figure 5: ≈60K active (unmatched) packets seen in the campus trace during peak traffic.

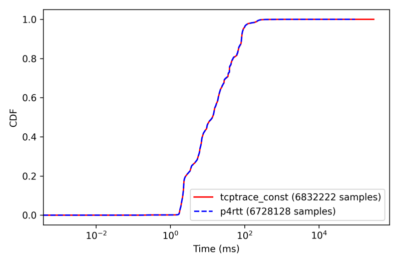

Figure 6: *P4RTT* collects 98% of samples collected by *tcptrace_const*; CDFs are almost identical.

packet. Such entries stay on in the hash table until a matching ACK packet is seen; a *matching* ACK packet has the same flow ID as an existing entry (upon reversal of source and destination fields) and the exact ACK number indicated by the eACK of that entry. The entry tagged 2 shows this phenomenon; a matching ACK packet causes the entry to be deleted, and the corresponding RTT sample is reported. Note that the RTT (50 ms here) is computed as the difference between the timestamp of the ACK packet (350) and the timestamp of the SEQ entry (300). This arrangement is straightforward and works well when TCP behavior is favorable [5]. However, as we discuss in the next section, certain vagaries of the TCP protocol pose serious challenges to *P4RTT*.

Some methods measure RTTs by using the *TCP timestamp* option available in the TCP header, instead of matching data and ACK packets (e.g., *pping* [11]); however, since timestamps can only be set by end-points, an in-network device would suffer from time synchronization issues. Furthermore, TCP timestamps are often too coarse-grained (e.g., 10 or 100 ms granularity), and many services do not use them at all since they are optional [6].

It is worth mentioning here that the QUIC protocol which is replacing a lot of TCP traffic on the Internet encrypts the header information that allows us to match data packets with ACKs in the aforementioned way [15]. However, QUIC implements *packet numbers* and a *spin bit*, which make it possible to implement a similar technique. The exact specifics of such a technique is part of our future work.

3 *P4RTT* DESIGN AND EVALUATION

The strawman (§2) is vulnerable to inaccuracies (§3.1) and can also be memory inefficient (§3.2). The following discussion illustrates this, and proposes mitigating strategies.

3.1 Handling Vagaries of TCP Protocol

When TCP retransmits lost packets, the monitoring device sees two copies of the same data packet and fails to conclusively match one of these with an ACK packet (a condition known as the *TCP retransmission ambiguity*). Sometimes packets get reordered enroute to the destination, causing the ACK for a late-arriving reordered packet to mistakenly get matched with an in-order data packet (a condition known as the *TCP reordering ambiguity*). Although it is impossible to correct for such inaccuracies to produce valid RTT samples, we can at least detect the occurrence of these ambiguities if we maintain some flow-state. Our idea is to maintain a window of *unambiguous* bytes (specifically, the starting byte number and the ending byte number) for which it is safe to collect RTT samples—we call this byte-range the *measurement range*. *P4RTT* is therefore designed as a cascade of two data structures, instead of just one in the strawman:

- The *flow table* stores the flow state, with the 4-tuple as the *key* and the measurement range as the *value*;
- The subsequent *packet table* stores packet state just as in the strawman solution.

RTT samples are collected only for packets determined as *unambiguous* by the flow table. Previous work that measures RTT passively by matching data and ACK packets (e.g., *tcptrace* [12]), has indeed observed and corrected for the aforementioned ambiguities, by keeping a large amount of per-flow state. In contrast, *P4RTT* is able to deal with the same ambiguities using only *constant* per-flow state, which makes it well-suited for the resource-constrained high-speed data planes (discussed next).

3.2 Working within Resource Constraints

High-speed data planes have severe resource constraints. Packets are processed in a streaming fashion, and there are only a limited number of computational stages (e.g., 10-20); although recirculations are allowed, doing so for more than 5-10% of packets starts to incur a heavy performance hit. Register memory that could be allocated for storage of flow and packet tables is also limited (≈50K fixed-size records). Under these constraints, malicious behavior such as SYN flood attacks or port scans can easily overwhelm our data structures. Even legitimate user traffic during peak usage hours could subject *P4RTT* to enormous memory pressure. Furthermore, TCP often ACKs multiple data packets cumulatively, i.e., only one out of two or more packets get ACKed; some packet records that never receive ACKs keep occupying scarce memory as a result. Idle flows and really long RTTs do not help either. *P4RTT* is able to overcome these constraints through several design choices:

- Our data structures span multiple stages of the hardware pipeline to allow for multiple insertion opportunities to increase occupancy;
- We adopt an insertion policy called *cuckoo hashing* [13] that forces old packet records to recirculate upon memory contention—every old packet that is now *stale* (i.e., its flow measurement range is ahead of its eACK) is evicted, thus making space for new entries;
- *P4RTT* does not store information about SYN and SYN-ACK packets since it does not rely on handshake RTTs. This helps thwart SYN floods and port scans. This prevents SYN floods and port scans from exhausting switch resources, thus allowing for the collection of more measurement data.

Furthermore, during peak traffic when *P4RTT* cannot possibly collect all potential RTT samples, we ensure that *P4RTT* collects unbiased samples, i.e., the distribution of sampled RTTs matches the distribution of actual RTTs closely, so that QoE estimates are not distorted.

3.3 Early Results on Campus Trace

We collected a 15-minute long trace from the Princeton campus using our P4Campus testbed in early April, 2020[1]. Fig. 4 shows the number of active connections per-second over the duration of the trace. The number of active (unmatched) packets per-second is shown in blue in Fig. 5; the number of packets that finally contribute to RTT samples is shown in orange, while the number of packets that do not is shown in green. The 99^{th} percentile number of active connections is ≈49K, while the same for active packets is ≈60K; these statistics provide an indication of the peak load on *P4RTT*'s flow table and packet table, respectively.

We simulated *P4RTT* in Python to resemble a Tofino-based implementation faithfully and parameterized the total sizes of data structures and the number of stages (for multi-stage data structures). We compared its performance, for various parameter sets, against *tcptrace_const*, a constant-space ($O(1)$ per record) Python version of *tcptrace* [12] that we implemented as a baseline. Fig. 6 shows that even when parameterized with a flow table of size 1024 and a packet table of size 4096 (with 4 stages allocated to each table), which are mere fractions of the peak flow and packet loads respectively, *P4RTT* is able to collect 98% of the samples that *tcptrace_const* can collect with unlimited memory. The RTT distributions are roughly identical as well, as can be observed from Fig. 6.

[1]The trace was anonymized and the study is approved by the Institutional Review Board (IRB) and the Institutional Review Panel for the use of Administrative Data in Research (PADR) at Princeton.

4 CONCLUSION

The potential benefits of an in-network QoE monitoring and network adaptation system are undeniable. However, building such a system is rife with difficult challenges, both due to the vagaries of TCP traffic and the severe resource constraints of high-speed programmable switches. In this paper, we posit that it is indeed possible to design and implement such a system—which we call *P4RTT*. We present early results that show promise in its role as a continuous, fine-grained RTT monitoring system. Our future work involves expanding the scope of *P4RTT* to network adaptation in reaction to RTT anomalies, and deploying it to our P4Campus testbed to fully realize our vision. Our future work also includes expanding the scope of *P4RTT* to the increasingly popular QUIC protocol.

REFERENCES

[1] Anurag Agrawal and Changhoon Kim. 2020. Intel Tofino2: A 12.9 Tbps P4-Programmable Ethernet Switch. In *IEEE Hot Chips Symposium (HCS)*. IEEE Computer Society, 1–32.

[2] Debopam Bhattacherjee, Muhammad Tirmazi, and Ankit Singla. 2017. A cloud-based content gathering network. In *USENIX Workshop on Hot Topics in Cloud Computing*.

[3] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014), 87–95.

[4] Francesco Bronzino, Paul Schmitt, Sara Ayoubi, Guilherme Martins, Renata Teixeira, and Nick Feamster. 2019. Inferring streaming video quality from encrypted traffic: Practical models and deployment experience. In *ACM SIGMETRICS*. 1–25.

[5] Xiaoqi Chen, Hyojoon Kim, Javed M Aman, Willie Chang, Mack Lee, and Jennifer Rexford. 2020. Measuring TCP round-trip time in the data plane. In *ACM SIGCOMM Workshop on Secure Programmable Network Infrastructure*. 35–41.

[6] Hao Ding and Michael Rabinovich. 2015. TCP stretch acknowledgements and timestamps: Findings and implications for passive RTT measurement. *ACM SIGCOMM Computer Communication Review* 45, 3 (2015), 20–27.

[7] Jon Dugan, Seth Elliott, Bruce A Mah, Jeff Poskanzer, and Kaustubh Prabhu. 2014. iperf3, tool for active measurements of the maximum achievable bandwidth on IP networks. *URL: https://github.com/esnet/iperf* (2014).

[8] Hao Jiang and Constantinos Dovrolis. 2002. Passive estimation of TCP round-trip times. *ACM SIGCOMM Computer Communication Review* 32, 3 (2002), 75–88.

[9] Hyojoon Kim, Xiaoqi Chen, Jack Brassil, and Jennifer Rexford. 2021. Experience-driven research on programmable networks. *ACM SIGCOMM Computer Communication Review* 51, 1 (2021), 10–17.

[10] RIPE NCC. 2010. RIPE Atlas. https://atlas.ripe.net/. (2010).

[11] Kathleen Nichols. 2017. pping (pollere passive ping). https://github.com/pollere/pping. (2017).

[12] Shawn Ostermann. 2007. tcptrace Homepage. *http://www.tcptrace.org/* (2007).

[13] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo hashing. *Journal of Algorithms* 51, 2 (2004), 122–144.

[14] Larry Peterson, Tom Anderson, Sachin Katti, Nick McKeown, Guru Parulkar, Jennifer Rexford, Mahadev Satyanarayanan, Oguz Sunay, and Amin Vahdat. 2019. Democratizing the network edge. *ACM SIGCOMM Computer Communication Review* 49, 2 (2019), 31–36.

[15] B. Trammell and M. Kuehlewind. 2018. *The QUIC Latency Spin Bit.* Technical Report. https://datatracker.ietf.org/doc/html/draft-ietf-quic-spin-exp.

[16] Philip Treleaven, Michal Galas, and Vidhi Lalchand. 2013. Algorithmic trading review. *Commun. ACM* 56, 11 (2013), 76–85.