Measuring TCP Round-Trip Time in the Data Plane

Xiaoqi Chen, Hyojoon Kim, Javed M Aman, Willie Chang, Mack Lee, Jennifer Rexford
Princeton University
{xiaoqic,hyojoonk,javeda,whchang,mackl,jrex}@cs.princeton.edu

ABSTRACT

We present a data-plane algorithm that *passively* and *continuously* monitors the Round-Trip Time of TCP traffic, by matching data packets with their associated acknowledgments and calculating a time difference. Compared with traditional measurement systems based on active probing or measuring only SYN/ACK packets, our algorithm passively produces many samples for long-running connections. This enables network operators to observe abnormal RTT increases, which signal possible security or performance issues in the network, in real-time. To satisfy the stringent memory size and access constraints of programmable switches, our algorithm uses a multi-stage hash table data structure to maintain records for in-flight packets; the records not receiving their acknowledgments are lazily expired and overwritten. We implement our algorithm on a Barefoot Tofino programmable switch. Evaluation using a real-world traffic trace from a 10 Gbps campus network link demonstrates that our solution can accurately capture 99% of available RTT samples, using only 4 MB of data-plane memory.

CCS CONCEPTS

• **Networks** → **Data path algorithms**; **Network measurement**;

KEYWORDS

Network Monitoring, Data Plane, P4, TCP RTT, Hash Tables

ACM Reference Format:
Xiaoqi Chen, Hyojoon Kim, Javed M Aman, Willie Chang, Mack Lee, Jennifer Rexford. 2020. Measuring TCP Round-Trip Time in the Data Plane . In *Workshop on Secure Programmable Network Infrastructure (SPIN'20), August 14, 2020, Virtual Event, NY, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3405669.3405823

1 INTRODUCTION

Round-Trip Time (RTT) is a key metric for network latency. An increasing RTT not only affects user's Quality of Experience, but also indicates possible performance or security issues in the network, such as congestion or routing changes. Although RTT statistics are often readily available at end hosts [14, 15, 28], an Internet Service Provider (ISP), such as a consumer broadband provider or an enterprise network operator, does not have direct visibility into the latency experienced by its customers. Even in a data center, continuously monitoring RTTs at all hosts is costly. Yet, continuous RTT monitoring would allow an ISP to better understand both the security and performance of its network traffic:

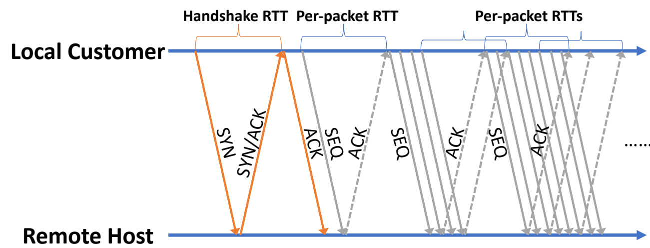

Figure 1: Match data and ACK packets to measure RTT.

- **BGP Routing Security**: When an attacker uses BGP routing attacks [3, 19] to detour and intercept traffic, that traffic would likely experience higher-than-normal RTT. Thus, unexpected changes in the RTT to a remote host may signal a reroute, either due to equipment failure, or a routing attack. Continuous RTT monitoring can help the ISP discover reroutes, even if the re-routing happens further downstream.
- **Detecting IP Spoofing**: A spoofed IP address exhibits discordant RTT values than the legitimate traffic from the same address, therefore RTT can be used to improve the accuracy for IP spoofing detection [13, 16].
- **Service-Level Agreement (SLA)**: An ISP usually has RTT requirements in its SLA with customers. For example, Verizon agrees to set 45 ms and 30 ms as the maximum RTT of intra-North-America and intra-Europe traffic, respectively [26, 27]. Monitoring RTT in real-time allows an ISP or its customer to verify the RTT is within limits, or be notified about an upcoming breach of the SLA.
- **Quality of Experience (QoE)**: An ISP may want to measure the QoE for customers using a variety of applications. Some applications such as video live-streaming are sensitive to high latency and jitter [4], which can be captured in RTT measurements. An increase in RTT may reflect persistent congestion and queuing on peering links [8], which can inform an ISP to upgrade its equipment for those links to better accommodate customers' demand.

To measure RTT, network operators today rely on active measurement tools such as NDT [10], PingMesh [12], and perfSONAR [22], sometimes after a client reports a degradation in service quality. Meanwhile, passive performance measurement tools (e.g., Ruru [7]) mostly report RTT samples based only on the three-way TCP connection handshake. Such tools cannot capture the latency change during long-running TCP connections such as video streaming. Also, they may be biased when SYN/SYN-ACK packets are processed differently than regular TCP packets; for example, SYN/SYN-ACK packets might go through a middlebox or get delayed by the remote server before accepting new connections.

SPIN'20, August 14, 2020, Virtual Event, NY, USA

ACM ISBN 978-1-4503-8041-6/20/08…$15.00
https://doi.org/10.1145/3405669.3405823

In this paper, we present an algorithm to continuously measure RTT for *all* outgoing TCP packets, on a programmable switch at an ISP vantage point, *passively* and *continuously*. Our algorithm does this by matching an outgoing TCP packet using its sequence number with an incoming packet that has the corresponding acknowledgment number. As illustrated in Figure 1, our algorithm captures RTT samples beyond the three-way handshake, allowing continuous monitoring throughout a TCP session. Running in the data plane of commodity programmable switches gives us the opportunity to measure per-packet RTT in real-time, at a higher line rate. This enables potential future work on real-time mitigation directly in the data plane (e.g., reroute for further inspection) when RTT anomalies are detected (e.g., IP spoofing).

Continuously monitoring RTT in the data plane has several unique challenges. To achieve high throughput and constant-time processing, the programmable switch imposes strict constraints, including limited memory size and memory access pattern. We need to store records for outgoing packets in the memory, then efficiently look up these records to calculate RTT upon seeing incoming acknowledgments (ACKs). Due to the TCP delayed ACK mechanism, some packets never receive their corresponding ACKs, so we need to clean up their records; some ACKs are delayed, which inflates RTT, so we also need to filter them. Finally, as memory is limited in the data plane, sometimes we cannot record every packet and inevitably lose some RTT samples; we want to ensure the subset of RTT samples we indeed measure are unbiased: a high-RTT packet with a late-arriving ACK shall not be discriminated against, and it should have an equal chance to be reported as the samples with lower RTT.

Our solution is to use a multi-stage hash table data structure that performs "lazy garbage collection", by assigning an expiration time for each record and overwriting expired records only upon hash collisions. For each outgoing packet, we record a fingerprint (a hash of 5-tuple flow ID and expected ACK number) and a timestamp in the hash table. The records matching with incoming packets produces RTT samples and are deleted, while those never matched with incoming packets are considered expired based on their timestamps, and are overwritten when hash collisions occur. When the data structure runs out of memory, it randomly rejects new records, thus automatically achieves unbiased sub-sampling.

We have implemented our algorithm on a commodity programmable switch using the P4 language [21]. We are in the process of deploying it in our local campus network. Our deployment will provide researchers with valuable measurement data about RTTs "in the wild," while also giving the local network operators a useful tool for diagnosing end-user performance problems in real-time.

The remainder of this paper is structured as follows. Section 2 introduces our RTT measurement algorithm based on multi-stage hash tables, as well as some considerations in measuring real-world TCP flows. In Section 3, we evaluate our algorithm for its accuracy and resource requirements. Section 4 discusses some related work on RTT monitoring, and we conclude the paper in Section 5.

2 MEASURING RTT IN THE DATA PLANE

In this section, we present our data-plane RTT monitoring technique using a multi-stage hash table data structure.

2.1 Overview of Measuring TCP RTT

A TCP connection carries bi-directional data streams between two end hosts. In our application scenario, one end host resides in our local network and the other is a remote host, similar to [1]. At the vantage point, we can see both incoming and outgoing TCP packets, and observe TCP sequence (SEQ) and acknowledgment (ACK) numbers. In particular, each outgoing TCP packet with non-zero payload may be acknowledged by a future ACK number sent from the remote host. We can then infer the round-trip time from the vantage point to the remote host using the time difference between the two packets. Note that we only consider the Internet leg of the RTT and ignore the local leg from our vantage point to the local host, which we consider negligible for a local ISP.

Thus, at our vantage point, we do the following:

(1) For each outgoing TCP packet with a unique expected ACK number (*eACK*), we record its flow ID (IP address pair and port pair), *eACK* (calculated using the SEQ number plus the payload size), and a timestamp. Non-handshake packets that have no payload are not recorded.
(2) For each incoming TCP packet, we look up our records using its flow ID and ACK number. If we find a match, we subtract the current time with the recorded outgoing timestamp to recover an RTT sample from this packet.

2.2 Lazily Expiring Records

In reality, many TCP packets do not receive a corresponding ACK for various reasons; for example, a remote host using the TCP "delayed ACK" mechanism may only send one ACK for every two consecutive data packets. A strawman solution that removes records only when they are matched will soon find its memory filled up by stale records. To efficiently use the limited memory space, we need to clean up the records for the unmatched packets. Yet, if we clean the records too aggressively, ACKs arriving long after the data packets may fail to match their records, so we cannot produce RTT samples for high RTTs.

We set an expiration threshold for all records: a record that was not matched after a predetermined interval T_Expire will be considered stale and get evicted. Fortunately, as a record includes a timestamp already, we do not need any extra memory to implement this expiration mechanism. This threshold is set to be reasonably larger than the RTT samples observed in a network to avoid prematurely removing records. For example, in Section 3 we set this to 500 ms as it corresponds to the 99^{th}-percentile of the RTT samples observed in our network, so we rarely under-sample high RTTs.

When we set T_Expire too small, an outgoing packet's record may get overwritten before the incoming packet can match it. When T_Expire is too large, the algorithm's memory fills up with useless records, preventing the tracking of new packets that would produce RTT samples later. Both cases result in missing a lot of RTT samples unnecessarily. When T_Expire is set appropriately, the algorithm uses its memory efficiently to store records and is not under memory pressure.

It is, however, expensive to track all the records and actively remove a record from the data structure once it expires. Therefore, we *lazily expire* such records: if a record's timestamp becomes too old, it is *overwritten* by a future attempted insertion into the same

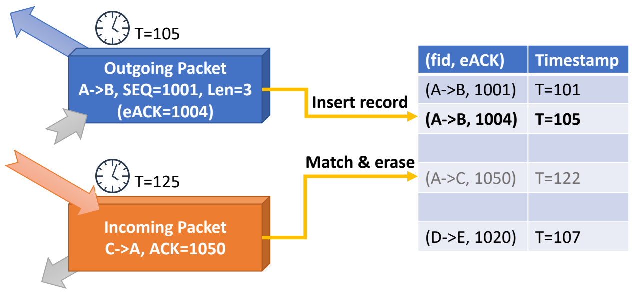

Figure 2: We store a record for outgoing packets in a hash table. An incoming packet can find its matching record, calculate its RTT, and remove the record from the table.

location, making our data structure self-cleaning. As a bonus, this mechanism allows producing RTT samples for packets with true RTT higher than T_Expire that were not removed immediately upon expiration; observing many such samples implies a need to increase T_Expire.

We assume the traffic at our vantage point exhibits a stable RTT distribution, therefore we only need to set the constant T_Expire once for any given vantage point. However, if the RTT distribution changes frequently and significantly, we may need to adjust T_Expire on the fly, based on the observed RTT distribution. We leave the adaptive adjustment of T_Expire as future work.

2.3 Multi-stage Hash Table

Programmable switches are constrained in how they access their data-plane memory, as surveyed in prior works [2, 18]. In particular, the amount of memory available in a hardware pipeline stage is limited, and an algorithm can only perform a limited number of memory accesses per stage.

In this paper, we present a novel implementation of hash tables in the data plane to store the records of outgoing packets. Unlike prior works that utilize hash table-based data structure to aggregate data, our implementation uses the tables to perform a *join* of outgoing and incoming packet streams in the data plane. Our algorithm also lazily cleans expired entries to further reduce workload.

In our use case, a strawman solution can use a simple one-stage hash table to store packets, as illustrated in Figure 2:

(1) For outgoing packets, we compute a memory address using the hash function. If the location is empty or the existing record has expired, we write the record tuple $(fid, eACK, timestamp)$; otherwise, we record nothing.

(2) For incoming ACKs, we calculate the same hash-based address to retrieve the recorded tuple, check if the flow ID and *eACK* matched, and finally compute the RTT based on the recorded timestamp. If the ACK does not match the recorded tuple, we do not compute an RTT sample.

In the example shown in Figure 2, an outgoing packet with flow ID `A->B`, sequence number 1001 and length 3 arrives at time $T = 105$. We first compute its expected acknowledgment number $eACK = 1001 + 3 = 1004$, then use a hash function to find its location in the table $h(flowID, eACK) = 2$, and insert a record into the 2^{nd} row of

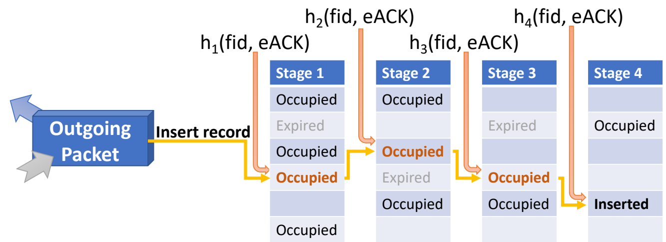

Figure 3: In the multi-stage hash table, each stage uses a different hash function to calculate the location.

the table. Later, an incoming packet with flow ID `B->A` and ACK number 1004 may arrive, to match with (and erase) this record.

For an incoming packet with flow ID `C->A` and ACK number 1050, arriving at $T = 125$, we first reverse its flow ID into `A->C`, and find a location using the hash function $h(flowID, ACK) = 4$. Then, we verify that the record stored in the 4^{th} row indeed matched the incoming packet, and read the stored timestamp 122. We now report an RTT sample 3 for flow `A->C`, and erase this record from the table.

However, the strawman solution suffers from the maximum memory size limit of a single pipeline stage. Furthermore, due to memory access constraints in the data plane, each packet has only one chance to be inserted into the table, and cannot be saved upon a hash collision with another entry.

Therefore, we use multiple memory arrays spread across different pipeline stages to implement a multi-stage hash table; as before, each table stores record tuples of $(fid, eACK, timestamp)$. Note that we use different independent hash functions for addressing in each table, which further reduces the impact of hash collisions. We optimize the algorithm's memory space demand by using a fingerprint hash function H to produce and store a 32-bit fingerprint $H(fid, eACK)$ in the hash tables, instead of the 128-bit original form. For incoming packets, we reverse the flow ID to produce the same fingerprint $H(fid, ACK)$ as their matching outgoing packets.

In Figure 3, we illustrate the process of inserting a record for an outgoing packet into a S=4-stage hash table. We note that evaluation in Section 3.3 showed that $S = 3$ or $S = 4$ yield the most performance improvements given the same total memory space, while having more stages provides diminishing returns. Given the packet's flow ID and expected ACK number, different hash functions h_1, h_2, h_3 and h_4 selects four locations in each stage independently. The algorithm first attempts to insert a record into the first stage, at address $h_1(fid, eACK) = 4$; since it is currently occupied and the current entry has not expired yet, the insertion fails. It subsequently tries inserting into the 2^{nd} and 3^{th} stage, before successfully inserting the record into an empty location at 4^{th} stage. If all four locations are occupied and unexpired, the outgoing packet will not be recorded.

Likewise, for incoming packets, the algorithm checks all four locations to see if they hold a matching record. Any matched record will be cleared, and the RTT is computed. If no matching record was found across all 4 stages, no RTT sample is produced for this incoming packet.

In Section 3, we evaluate the effect of changing the number of tables and their sizes on the algorithm's success rate.

2.4 Analyzing and Reporting

In our prototype implementation, once we obtain an RTT sample, we report it alongside the packet's flow ID to the switch control plane. It is also possible to aggregate the RTT samples based on destinations (defined by IP blocks or domain names), and subsequently calculate the RTT distribution for each remote destination in the data plane in real-time. The switch can report only the aggregated statistics for each IP block, saving the throughput required for reporting each and every RTT sample.

Network operators may specify an expected RTT distribution for a particular destination, and get notified when the distribution deviates significantly. This allows timely detection of SLA violations or BGP hijacks. Detecting these violations in the data plane opens up the possibility of the switch taking immediate action, e.g., to re-route the traffic to a faster or more secure alternative path. We leave this as future work.

2.5 Challenges with Measuring RTT

Bidirectional traffic. Our vantage point must see both directions of the data stream. This assumption is true for most local ISPs, however at Internet scale the outgoing and incoming traffic may traverse different paths due to asymmetric routing.

Outgoing traffic. Each RTT sample requires a unique outgoing SEQ number, thus the outgoing packets cannot have zero payload length. Therefore, we need some amount of data sent in the outgoing direction; a pure incoming TCP flow, such as downloading a large file from a remote server, does not produce RTT samples beyond the initial handshake.

However, we should note that many modern user applications like web apps, video streaming, etc., include two-way traffic for tracking or control purpose. In particular, web-based video playback (such as Netflix) are often chunk-based, with the browser requesting 5-second or 15-second chunks periodically, thus we can expect outgoing data (and hence RTT samples) every 5 or 15 seconds.

Delayed ACK. Delayed ACK is an optimization used by some TCP implementations to combine an acknowledgment packet with response traffic. By not immediately sending back an ACK packet for incoming data, the host has an opportunity to piggyback future response data with this acknowledgment. When there is no response to send, a delayed ACK timer will timeout, usually after 50 ms, and an ACK packet with no piggybacked data will be sent. The hosts also immediately send out the ACK after receiving two consecutive full-sized packets. A packet receiving a delayed ACK produces an artificially higher RTT sample since it includes the delay timeout. To avoid producing biased RTT samples, we need to filter packets that experience a delayed ACK. Rather than track the TCP state machine for each flow, we use a very simple heuristic: the full-sized packets typically do not suffer from delayed ACKs, as end hosts are not allowed to delay ACKs when receiving two consecutive full-sized packets. To further ease implementation, we avoid tracking Maximum Transmission Unit (MTU) or TCP's negotiated Maximum Segment Size (MSS) for each flow, but rather assume a packet is full-sized if its length is one of several commonly used MTUs (e.g. 1440, 1500, etc.); the user can choose to only report RTT samples produced by outgoing packets with these sizes.

Stretch ACK. Due to delayed ACK, we expect observing one incoming ACK for every two data packets. Yet, the server may send even fewer ACK packets, and this practice is referred to as Stretch ACK [17]. As we discussed earlier, packets not receiving corresponding ACKs create stale hash-table entries, which are automatically removed when they expire. Meanwhile, the effect of Stretch ACKs on RTT measurement accuracy has been studied previously by [9].

Selective ACK and retransmissions. When a packet is lost, TCP will re-transmit the packet after seeing duplicated ACKs; we may observe two identical outgoing packets in this case. If there are packets with larger SEQ numbers already delivered, the acknowledgment for the re-transmitted packet will directly jump to a much later ACK number than its *eACK*. The re-transmitted packet will not produce an incorrect RTT sample, however the resulting ACK packet may produce an inflated RTT sample, as it could be matched with an earlier data packet. TCP implementations may also send Selective ACK (SACK) upon packet drops to acknowledge subsequent packets; the SACK packet will share the same ACK number as an earlier normal ACK packet, which would have erased the matching record. Thus, our algorithm will not produce an incorrect RTT sample for these SACK packets.

Sampling under memory pressure. In Section 3, we show that our prototype tracks >99% of RTT samples using a moderate amount of data-plane resources. However, if the monitored link rate grows faster and average RTT grows higher, our algorithm needs more memory to save in-flight records and achieve adequate accuracy. Also, data-plane memory may be shared among other measurement applications running in the data plane, further limiting the memory available for RTT measurement. When memory is insufficient, records for new outgoing packets cannot be inserted into the data structure, which is filled up by unmatched and unexpired records. However, since records are naturally expiring and the location for insertion is pseudo-random (determined by hash functions), some records will be inserted successfully when their randomly chosen location aligns with a just-expired record. In effect, a random fraction of outgoing packets are automatically sampled, and the algorithm produces an unbiased sample subset of RTT measurements.

As an alternative to the packet-level sampling, we can also randomly sample a small fractions of flows or IP addresses, and only insert their records into the hash table data structure. This way, we can accurately measure the RTT distribution for the fraction of flows or IP addresses sampled.

Security. As a proof of concept, our current data plane implementation uses simple hash functions and fixed expiration threshold. An attacker who can inject traffic into the measured network may manipulate the outgoing and incoming packets, to either deliberately cause hash collisions and evade RTT measurement, or maximize the algorithm's memory consumption by controlling the RTT to be just below the fixed threshold. To defend against such adversarial traffic, we should use a more secure hash function implementation in the data plane, and introduce random expiry for hash table entries; we leave these as future work.

3 EVALUATION

We implement our RTT measurement algorithm and multi-stage hash table data structure on a Barefoot Tofino programmable switch using $P4_{16}$ [21]. Our implementation has approximately 600 lines of code, and is open-sourced on GitHub[1]. We also implement the identical algorithm in a Python-based simulator, which allows us to arbitrarily adjust table sizes and the number of tables, beyond hardware limitations.

We use different variants of the CRC16 function (with different polynomials) to calculate indices in hash tables, and use the CRC32 function to calculate packet fingerprints. Since each record consists of a 32-bit nanosecond-precision timestamp and a 32-bit fingerprint hash, a S=8 tables, 64k records-per-table configuration uses S+1=9 hash function computations and $8 \times 2 \times 32bit \times 64k$=4096KB of data-plane memory, both less than 50% of total capacity. To verify our algorithm can report RTT samples under a realistic workload, we collected a bi-directional traffic trace from a vantage point in a university campus network, which is also a future deployment site of the algorithm. We subsequently use the trace to evaluate the effectiveness of our RTT monitoring algorithm using the simulator, under various table sizes and number of stages.

3.1 Dataset and Method

We captured a bi-directional traffic trace from a 10 Gbps peering link between a border router of a university campus network and a local ISP. The traffic trace has been anonymized and sanitized to obfuscate personal data before being used by researchers, and our research has been approved by the university's institutional review board.

The trace contains 1 million TCP packets and lasts 1.10 second. It contains 11,085 unique TCP flows, with a mean and median IP packet size of 1100 and 1500 bytes, respectively; about 58% of packets are likely MTU-sized (longer than 1450 bytes).

After tagging packets as incoming or outgoing based on IP prefix, we calculated the ground truth RTT samples by matching TCP sequence and acknowledgment numbers. The trace contains 0.6 million outgoing packets, 0.4 million incoming packets, and 71K pairs of RTT samples. The median RTT for all samples is 44 ms. We plot the RTT distribution we observed in the trace in Figure 4. As a back-of-envelope calculation, reporting all 71k RTT samples to the control plane in real time (using 32-bit timestamps) requires 2.07 Mbps of additional throughput; reporting the RTT samples in lieu with the 12-byte flow ID requires 8.28 Mbps, a moderate fraction compared with the 10 Gbps line rate. We expect the throughput required for processing samples to grow proportionally when we scale up the measurement effort at a vantage point with multiple 100 Gbps links.

In the following experiments, we use 500ms as the stale threshold (corresponding to 99^{th} percentile of all RTT samples), and investigate our algorithm's success rate under various table size configurations. The success rate is determined by how many incoming packets are matched with a record (out of those having ground truth RTTs, i.e., theoretically could have matched with a record of an outgoing packet).

[1] https://github.com/Princeton-Cabernet/p4-projects/tree/master/RTT-tofino

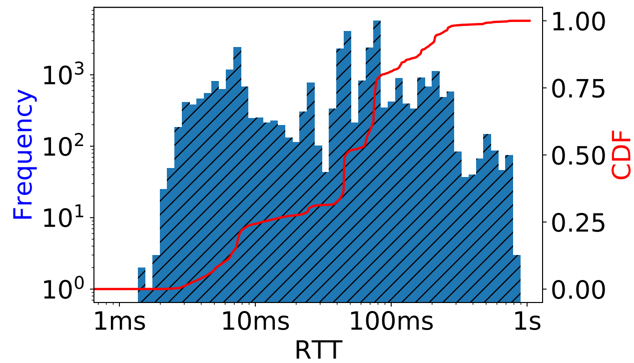

Figure 4: The histogram and Cumulative Distribution Function (CDF) of the RTTs observed in our experiment trace, collected from a university network border router.

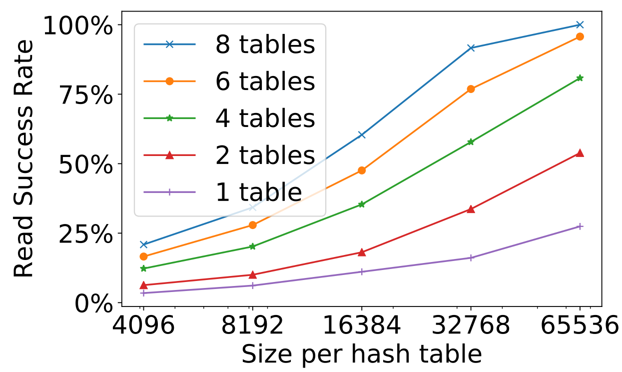

Figure 5: As we allocate more memory to each hash table, the algorithm achieves higher read success rate, defined as the number of RTT samples correctly produced divided by total possible RTT samples.

3.2 Table Size

We first investigate the relationship between the size of multi-stage hash table, which directly relates to our algorithm's memory footprint, to the percentage of successful matches. We now vary the size of each hash table, and check how it affects the algorithm's success rate for reporting all RTT samples. We define **Read Success Rate** as the number of incoming packets successfully matched with a recorded timestamp stored in the data structure, divided by the total number of RTT samples available in the ground truth.

As can be seen from Figure 5, when our hash table grows larger, the likelihood of a hash collision between non-expired records decreases, therefore more outgoing packets can be recorded and more incoming packets can successfully match with a record. We can reach over 99% of successful matches when using 8 tables with 65,536 entries, which corresponds to 4,096 KB of data plane memory, less than half of the total available in our switches.

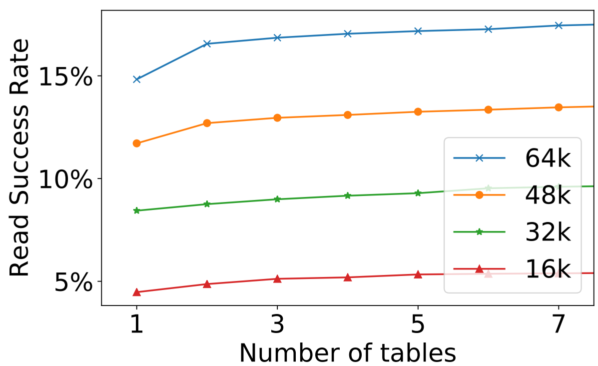

Figure 6: When splitting the same total memory across multiple stages, using S=3 or 4 tables yield the most significant improvement over using a single table, with diminishing return afterwards.

3.3 Optimal Number of Table Stages

We now investigate the optimal number of tables to use in the multistage hash table, given a fixed total memory size, to quantify the benefit of using multiple stages. In this experiment, we fix total memory size and divide it by varying the number of stages; for example, splitting 64k records into S=2 stages means having 32k records per table, while using S=7 stages will have 9.1k records in each table.

Figure 6 shows that the read success rate saturates at S=3 to 4 stages, under a fixed total memory size constraint. Having more than four stages yields diminishing returns, as the memory in later stages is underutilized. We note that the configurations in Figure 6 use a smaller total memory size than most points appearing in Figure 5, therefore they exhibit lower read success rate. Turkovic et.al. [24] explored the idea of using different table sizes per stage to achieve higher memory utilization; we leave this as future work.

4 RELATED WORK

Active/host-based measurements. Several works have explored measuring RTT from end-hosts. PingMesh [12] and NetBouncer [20] monitors the health of data center networks, including RTT, by using end hosts (or VM hypervisors) as vantage points to send and receive probe packets. However, unlike data center networks, ISPs do not have control over end-hosts, thus cannot run agents on them. Furthermore, active measurement tools add extra probing traffic into the network.

RTT measurement at ISP vantage points. Aikat et.al [1] measured the RTT experienced by campus network users by capturing traffic at a border link and later analyzing the traffic to match outgoing TCP packets with corresponding acknowledgments. This previous work shows the presence of significant variability in RTTs in a TCP connection, motivating our work to monitor this variability *in real-time*. Ruru [7] is a system that passively measures the RTT of TCP handshake packets at ISP vantage points. Yet Ruru does not measure RTTs for subsequent packets in long-running TCP connections.Veal et.al. [25] proposed a method to measure RTT beyond handshakes at an intermediate vantage point. However, it requires a modification to a TCP packet to add a timestamp option. It also depends on the recipient host to echo this timestamp.

Measuring RTT on a programmable switch. Dapper [11] is a TCP monitoring tool that tracks various metrics, including RTT, in the data plane. Dapper produces accurate measurements for the tracked flows, but it can only track a single outgoing packet per congestion window for RTT measurement, and must wait until that packet's acknowledgment arrives before recording another outgoing packet for the flow. Our algorithm does not limit the number of outgoing packet records stored for each flow; a flow can produce as many RTT samples as possible as long as the memory space permits.

Hash table data structure. Our multi-stage hash table data structure is motivated by prior works on data-plane algorithms for programmable switches. Count-Min Sketch [6] is a data structure made of several hash-indexed counter arrays, and is often used for estimating flow sizes and detecting heavy hitters. HashPipe [18] and PRECISION [2] designed more sophisticated multi-stage hash tables for heavy-hitter detection, with each record storing a hashed flow ID and a counter. Such data structures, however, are not suitable for continuous RTT monitoring as they are. An RTT sample is produced only when a corresponding acknowledgment arrives, thus a naive implementation of such data structure quickly gets populated by useless entries. Our multi-stage hash table implements an efficient garbage collection scheme by deleting expired entries upon hash collision, which addresses the issue caused by the delayed ACK mechanism. To the best of our knowledge, we are the first to implement a multi-stage hash table data structure for computing RTT in the data plane. Also, our approach is unique in the sense that entries in the hash table are automatically expired and lazily cleaned.

Non-TCP traffic. Google proposed QUIC [5], a UDP-based transport alternative to TCP. QUIC encrypts its packet header fields, which prevents an ISP from performing RTT measurement based on SEQ/ACK matching. The QUIC standardization body is planning to add a "spin bit" [23] specifically for RTT measurement at ISP vantage points.

5 CONCLUSION

We present an algorithm to track the per-packet Round-Trip Time of TCP traffic in a commodity programmable switch using a multi-stage hash table data structure. Our algorithm successfully reports over 99% of all RTT samples, in a traffic trace collected from a 10 Gbps peering link of a campus network. Our evaluation also shows that using three to four stages in our hash table structure achieves the best performance for RTT monitoring, given the same amount of total memory. We are currently deploying continuous RTT monitoring on our university campus network. For future work, we plan to integrate real-time RTT samples with anomaly detection and other routing change detection techniques.

6 ACKNOWLEDGMENTS

This research is supported by NSF Awards CNS-1704077. We sincerely thank the anonymous reviewers, as well as David Walker, Liang Wang, Shir Landau Feibish, Robert MacDavid, Mary Hogan, and Ross Teixeira, for their helpful comments and feedback.

REFERENCES

[1] Jay Aikat, Jasleen Kaur, F Donelson Smith, and Kevin Jeffay. 2003. Variability in TCP round-trip times. In *ACM SIGCOMM Internet Measurement Conference*. 279–284.

[2] Ran Ben-Basat, Xiaoqi Chen, Gil Einziger, and Ori Rottenstreich. 2018. Efficient Measurement on Programmable Switches Using Probabilistic Recirculation. In *IEEE ICNP*. 313–323.

[3] Henry Birge-Lee, Liang Wang, Jennifer Rexford, and Prateek Mittal. 2019. SICO: Surgical Interception Attacks by Manipulating BGP Communities. In *ACM SIGSAC Conference on Computer and Communications Security*.

[4] Francesco Bronzino, Paul Schmitt, Sara Ayoubi, Guilherme Martins, Renata Teixeira, and Nick Feamster. 2020. Inferring Streaming Video Quality from Encrypted Traffic: Practical Models and Deployment Experience. *ACM SIGMETRICS* (2020).

[5] Gaetano Carlucci, Luca De Cicco, and Saverio Mascolo. 2015. HTTP over UDP: an Experimental Investigation of QUIC. In *ACM Symposium on Applied Computing*. 609–614.

[6] Graham Cormode and S. Muthukrishnan. 2005. An improved data stream summary: The Count-Min Sketch and its applications. *Journal of Algorithms* 55, 1 (2005), 58–75.

[7] Richard Cziva, Christopher Lorier, and Dimitrios P Pezaros. 2017. Ruru: High-speed, Flow-level Latency Measurement and Visualization of Live Internet Traffic. In *ACM SIGCOMM Posters and Demos*. 46–47.

[8] Amogh Dhamdhere, David D Clark, Alexander Gamero-Garrido, Matthew Luckie, Ricky KP Mok, Gautam Akiwate, Kabir Gogia, Vaibhav Bajpai, Alex C Snoeren, and Kc Claffy. 2018. Inferring persistent interdomain congestion. In *ACM SIGCOMM*. 1–15.

[9] 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.

[10] Constantine Dovrolis, Krishna Gummadi, Aleksandar Kuzmanovic, and Sascha D Meinrath. 2010. Measurement Lab: Overview and an invitation to the research community. *ACM SIGCOMM Computer Communication Review* 40, 3 (2010), 53–56.

[11] Mojgan Ghasemi, Theophilus Benson, and Jennifer Rexford. 2017. Dapper: Data plane performance diagnosis of TCP. In *ACM SIGCOMM Symposium on SDN Research (SOSR)*. ACM, 61–74.

[12] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, et al. 2015. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *ACM SIGCOMM*, Vol. 45. ACM, 139–152.

[13] Ritu Maheshwari, C Rama Krishna, and M Sridhar Brahma. 2014. Defending network system against IP spoofing based distributed DoS attacks using DPHCF-RTT packet filtering technique. In *International Conference on Issues and Challenges in Intelligent Computing Techniques (ICICT)*. IEEE, 206–209.

[14] Matt Mathis, John Heffner, and Rajiv Raghunarayan. 2007. RFC4898: TCP extended statistics MIB. *IETF* (2007).

[15] Matt Mathis, John Heffner, and Raghu Reddy. 2003. Web100: Extended TCP instrumentation for research, education and diagnosis. *ACM SIGCOMM Computer Communication Review* 33, 3 (2003), 69–79.

[16] Ayman Mukaddam and Imad H Elhajj. 2012. Round trip time to improve hop count filtering. In *Symposium on Broadband Networks and Fast Internet (RELABIRA)*. IEEE, 66–72.

[17] Vern Paxson. 1997. Measurements and analysis of end-to-end Internet dynamics. *PhD Thesis, UC Berkeley* (1997).

[18] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer Rexford. 2017. Heavy-Hitter Detection Entirely in the Data Plane. In *ACM SIGCOMM Symposium on SDN Research*. 164–176.

[19] Yixin Sun, Anne Edmundson, Laurent Vanbever, Oscar Li, Jennifer Rexford, Mung Chiang, and Prateek Mittal. 2015. RAPTOR: Routing Attacks on Privacy in Tor. In *USENIX Security Symposium*. 271–286.

[20] Cheng Tan, Ze Jin, Chuanxiong Guo, Tianrong Zhang, Haitao Wu, Karl Deng, Dongming Bi, and Dong Xiang. 2019. NetBouncer: Active Device and Link Failure Localization in Data Center Networks. In *NSDI*. 599–614.

[21] The P4 Language Consortium. 2018. $P4_{16}$ Language Specification. https://p4.org/p4-spec/docs/P4-16-v1.1.0-spec.html. (Nov. 2018).

[22] Brian Tierney, Joe Metzger, Jeff Boote, Eric Boyd, Aaron Brown, Rich Carlson, Matt Zekauskas, Jason Zurawski, Martin Swany, and Maxim Grigoriev. 2009. perfSonar: Instantiating a global network measurement framework. *SOSP Workshop on Real Overlays and Distributed Systems* (2009).

[23] Brian Trammell and Mirja Kuehlewind. 2019. The QUIC Latency Spin Bit. *IETF Internet Draft* (2019). https://tools.ietf.org/html/draft-ietf-quic-spin-exp-01

[24] Belma Turkovic, Jorik Oostenbrink, and Fernando Kuipers. 2019. Detecting Heavy Hitters in the Data-plane. *arXiv preprint arXiv:1902.06993* (2019).

[25] Bryan Veal, Kang Li, and David Lowenthal. 2005. New methods for passive estimation of TCP round-trip times. In *International Workshop on Passive and Active Network Measurement*. Springer, 121–134.

[26] Verizon. 2020. IP Latency Statistics. (2020). https://enterprise.verizon.com/terms/latency/ Accessed: 2020-04-29.

[27] Verizon. 2020. Service Level Agreements. (2020). http://www.verizonenterprise.com/solutions/public_sector/state_local/contracts/calnet3/sla/ Accessed: 2020-04-29.

[28] Minlan Yu, Albert G Greenberg, David A Maltz, Jennifer Rexford, Lihua Yuan, Srikanth Kandula, and Changhoon Kim. 2011. Profiling Network Performance for Multi-tier Data Center Applications.. In *NSDI*, Vol. 11. 5–5.