Measuring TCP Round-Trip Time in the Data Plane

Anonymous Author(s)

ABSTRACT

We present a data-plane algorithm that *passively* measures TCP Round-Trip Time by matching data packets with their associated acknowledgments. This enables monitoring of the RTT of all outgoing traffic in real time without installing software on the end hosts. To satisfy the stringent memory size and access constraints of programmable switches, our algorithm uses a multi-stage hash table to efficiently maintain records for in-flight packets. To overcome the fact that many packets never receive any corresponding acknowledgement, we lazily expire their records upon hash collisions. We implement our algorithm on a commodity programmable switch, and are in the process of deploying it into a campus network. Evaluation using a real-world traffic trace from a 10 Gbps campus network link has demonstrated that our algorithm can accurately capture 99% of available RTT samples, using only 4 MB of data plane memory, less than 50% of total available.

1 INTRODUCTION

The Round-Trip Time (RTT) of network traffic directly affects a user's Quality of Experience, as it relates closely to the response time of user requests. Varying or increasing RTTs signal potential congestion or failure in the network. Although RTT statistics are often readily available at end hosts, an Internet Service Provider (ISP), such as a consumer broadband provider or enterprise network operator, does not have direct visibility into the RTT experienced by its customers. Even in a data center, continuously monitoring RTTs at all hosts may incur costly overhead. Yet, the ISP needs to measure the RTT of customer traffic for a variety of purposes:

- **Service-Level Agreement (SLA)**: The ISP may have a SLA with its customers regarding the RTT between customers and remote hosts. For example, Verizon provides a SLA of 45 ms and 30 ms, for maximum RTT of intra-North-America and intra-Europe traffic, respectively [21, 22]. Monitoring RTT in real time allows the ISP to verify it is honoring the RTT, or be notified about upcoming breach of SLA.
- **Quality of Experience (QoE)**: The ISP may want to measure the QoE for customers using a variety of applications. Some applications like video live-streaming are sensitive to high latency and jitter [4], which can be captured in RTT measurements. A periodic increase in RTT may reflect persistent congestion and queuing on peering links [8], and ISPs may upgrade their equipment specifically for those links to better accommodate the customer demand.

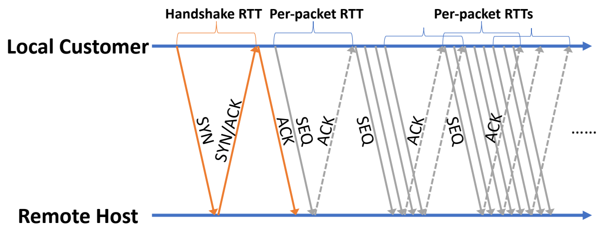

Figure 1: Match data and ACK packets to measure RTT.

- **Interdomain Routing Security**: Unexpected changes in the RTT to a remote host may signal a reroute, due to equipment failure or a BGP routing attack [3, 14]. Continuous RTT monitoring can help the ISP discover reroutes, even if the re-routing happens further downstream.

Currently, to measure RTTs, ISPs often rely on active measurement tools running on client machines (such as NDT [9] and perfSonar [17]), sometimes after the client reports a degradation in service quality. Meanwhile, passive performance measurement tools (such as Ruru [7]) often report RTT samples based only on the three-way TCP connection handshake. Such tools cannot capture the change of RTT during a long-running TCP connection, and may be biased when SYN/SYN-ACK packets are processed differently than regular TCP packets, e.g., going through a middlebox that increases RTT.

In this paper, we present an algorithm to continuously measure RTT for *all* outgoing TCP packets, running on a programmable switch at an ISP vantage point, *passively*, 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 many RTT samples beyond the three-way handshake. The algorithm enables a network operator to better understand the RTT distribution experienced by all hosts connecting to different destinations, as well as discover and respond in a timely way to RTT anomalies.

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, and enables potential future work on real-time re-routing after detecting RTT anomalies. However, to achieve high throughput and constant-time processing, the programmable switch also imposes strict constraints, including limited memory size and memory access pattern. We need to succinctly record information for outgoing TCP packets, and later match them with incoming ACKs efficiently. Furthermore, not all outgoing packets will receive incoming ACKs, and we need to find ways to "garbage collect" their stale records within the data plane; we cannot do housekeeping from the control plane, as it cannot keep up with new records being created for every outgoing packet.

Our solution is to use a multi-stage hash table data structure, which fits the pipeline architecture and memory access constraints of programmable switches. 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 are deleted, while those never matched with incoming packets are lazily expired based on their timestamps when hash collisions occur.

We have implemented our algorithm on a commodity programmable switch using the P4 language [16], and 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. In Section 2, we briefly review some related work on RTT monitoring. Section 3 introduces our RTT measurement algorithm based on multi-stage hash tables in more detail, as well as some considerations in measuring real-world TCP flows. In Section 4, we evaluate our algorithm for its accuracy and resource requirements, and we conclude the paper in Section 5.

2 RELATED WORK

Active/host-based measurements. Several works have explored measuring RTT on end-hosts for network performance monitoring. PingMesh [11] and NetBouncer [15] are end-host based system measuring 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 operators, other ISPs do not control end hosts directly and cannot measure RTTs easily using agents running at end hosts. Furthermore, these active measurement tools add extra probing traffic into the network, while our passive measurement algorithm does not.

RTT measurement at ISP vantage point. An ISP can measure the RTT of traffic when it sees both the outgoing and incoming direction of the traffic. Aikat et al. [1] measured the RTT experienced by campus network users by capturing traffic at a border link, and later analyzed the traffic to match outgoing TCP packet with incoming acknowledgements. We also focus on studying RTTs at a border link of campus network, but our data-plane algorithm produces RTT samples in real time and enables immediate mitigation of RTT anomalies. Ruru [7] is a system to passively measure RTT of TCP handshakes at ISP vantage points. Ruru does not measure RTTs for subsequent packets in long-running TCP connections, while applications such as video streaming may suffer from changing RTT in the middle of a connection. Veal et al. [20] proposed a method to measure RTT beyond handshakes at an intermediate vantage point. It calculates one leg of RTT (from the vantage point to one host) using SYN/ACK matching, and obtains the other leg of RTT using the TCP timestamp option. It requires modifying the TCP packet to add a timestamp option and depends on the recipient host to echo the timestamp.

Measuring RTT on a programmable switch. Dapper [10] 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 track at most one outgoing packet per flow for RTT measurement, and must wait until that packet's acknowledgment arrives before recording another outgoing packet for the flow. In our case, we do not limit the number of outgoing packet records stored for each flow, and a single flow can produce as many RTT samples as possible, as long as 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 commonly-used data structure in the data plane for estimating flow sizes and detecting heavy hitters, and is made of several hash-indexed counter arrays. Gated Sketch [19] explored the idea of using multi-stage hash tables for detecting heavy hitters, including using different table sizes for each stage. HashPipe [13] 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. Our data structure stores a timestamp instead of a counter, for calculating RTT and expiring records. 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.

3 RTT IN A MULTI-STAGE HASH TABLE

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

3.1 Overview of Measuring TCP RTT

A TCP connection carries bi-directional data streams between two end hosts, and in our application scenario, one end host resides in our local network and the other is a remote host, similar to [1]. Since our vantage point can see both directions of the data stream, it is possible to measure the RTT of TCP traffic by observing the packet's sequence (SEQ) and acknowledgement (ACK) numbers. In particular, each outgoing TCP packet with non-zero payload will 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 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.

The main challenges for this approach are managing the records unmatched with incoming ACKs and efficiently storing and looking up records.

3.2 Lazily Expiring Records

As much as half of outgoing data packets will never receive a corresponding ACK, as sometimes the remote host will only send one ACK for every two consecutive data packets (due to the TCP "delayed ACK" mechanism). A strawman solution that removes records only when they are matched will soon find its memory filled up by stale records.

To handle these unmatched records, 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 eventually removed. Fortunately, as the record includes a timestamp already, we do not need any extra memory to implement the expiration mechanism. We set this threshold to be larger than reasonable RTT samples observed to avoid prematurely removing records. For example, in Section 4 we set it to 500 ms as it corresponds to the 99^{th}-percentile of the RTT samples.

When we set *T_Expire* too small, an outgoing packet's record may be quickly overwritten before the incoming packet can match it, thus the algorithm cannot produce any samples. When *T_Expire* is too large, the algorithm's memory fills up with useless records, and only a small fraction of outgoing packets can be recorded (upon the expiration of some very old records); in this case, the algorithm can still produce RTT samples, for a small random fraction of outgoing packets. When *T_Expire* is set appropriately, the algorithm uses its memory efficiently to store records and is not under memory pressure.

We also note that it is expensive to track all the records and actively remove a record from the data structure once it expires. However, we can *lazily expire* it: the record is considered expired when its timestamp becomes too old, and will be overwritten by a future attempted insertion into the same memory location, making our data structure self-cleaning. An effect of this mechanism is that an occasional packet with a true RTT higher than *T_Expire* may still produce an RTT sample.

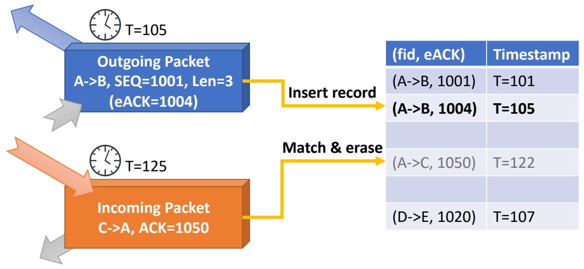

Figure 2: We store a record for outgoing packets in a hash table. An incoming packet can find its matching record, calculate its RTT, and remove the record from the table.

3.3 Multi-stage Hash Table

Programmable switches are constrained in how they access their data-plane memory, as surveyed in prior works [2, 13]. 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.

We use hash tables to store the records of outgoing packets. Prior works like Sonata [12] had implemented similar hash table-based data structure in the data plane, and our implementation is primarily different in that we use the tables to perform a "join" of outgoing and incoming packet streams, and expired entries are lazily cleaned.

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 acknowledgement 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 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

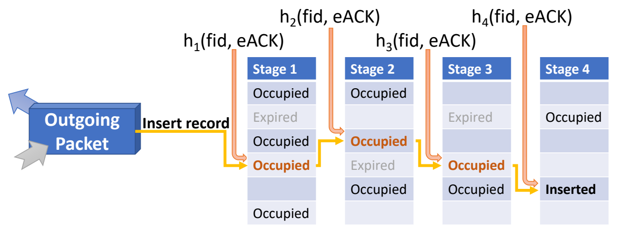

Figure 3: In the multi-stage hash table, each stage uses a different hash function to calculate the location.

$h(flowID, ACK) = 4$. 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 4.3 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 4, we evaluate the effect of changing the number of tables and their sizes on the algorithm's success rate.

3.4 Discussion

Analyzing and reporting RTT samples. Once we obtain an RTT sample, we can report it alongside the packet's flow ID to an outside collector or to the switch control plane. However, since there are many samples, we could filter out those samples with small RTTs and only report the samples that exceed a certain threshold, to reduce the number of generated reports. We can also group the RTT samples into larger groups (e.g., per geographic region), and analyze their statistics directly in the data plane, to check if a SLA for RTT is being violated. In this case, the control plane can poll periodically to learn the latest RTT statistics for every traffic group.

Outgoing traffic. Each RTT sample requires a unique outgoing SEQ number and the corresponding incoming ACK number, thus the outgoing packets cannot have zero payload length—continuous zero-payload TCP packets share the same SEQ number (except during the initial handshake). Therefore, we need some amount of data sent in the outgoing direction; a purely incoming TCP flow from remote host to local user, e.g., downloading a large file via FTP/HTTP, does not produce RTT samples beyond the initial handshake.

However, we should note that many modern user applications like web apps, video streaming, etc., includes 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. Also, services hosted on the local network will produce many outgoing traffic.

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. In our use case, a packet receiving 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 the samples produced by outgoing packets with these sizes.

Selective ACK and retransmissions. When a packet is dropped, 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*, so our algorithm will not produce an incorrect RTT sample for this re-transmitted 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 4, 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 a sampled set of RTT measurements.

QUIC. Recently, Google proposed QUIC [5], a UDP-based transport alternative to TCP. QUIC encrypts its packet header fields, which prevents the ISP from performing RTT measurement based on SEQ/ACK matching. Therefore, the QUIC standardization body is planning to add a "spin bit" [18] specifically for RTT measurement at ISP vantage points.

4 EVALUATION

We implement our RTT measurement algorithm and multi-stage hash table data structure using a Python-based simulator, which supports different table sizes and number of tables as input parameters. The simulator uses different variants of the CRC16 function (with different polynomials) to calculate indices in hash tables, and use the CRC32 function to calculate

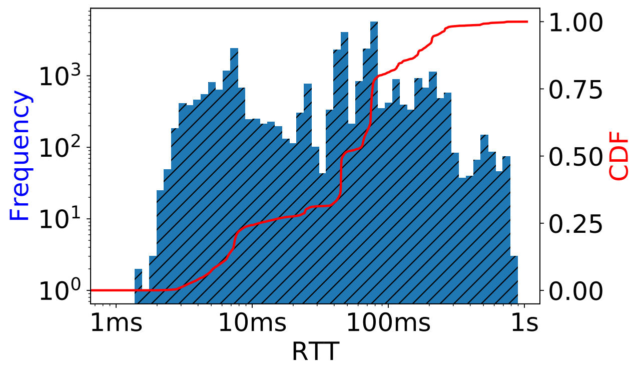

Figure 4: The histogram and Cumulative Distribution Function (CDF) of the RTTs observed in our experiment trace, collected from a university network's border.

packet fingerprints. We also implement our algorithm on a commodity programmable switch using about 600 lines of $P4_{16}$ [16] code, achieving identical functionality. 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.

4.1 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 across 11 thousand 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.

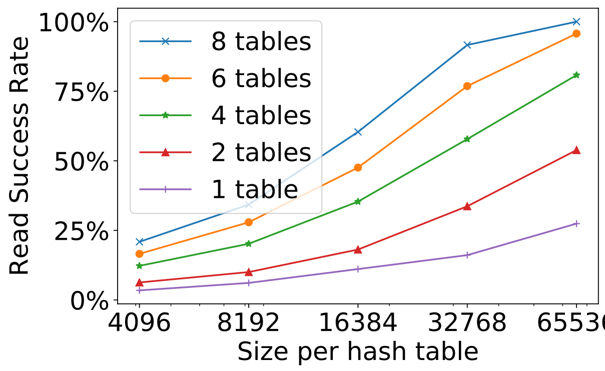

Figure 5: As we allocate more memory to each hash table, the algorithm achieves higher read success rate, defined as the number of RTT samples correctly produced divided by total possible RTT samples.

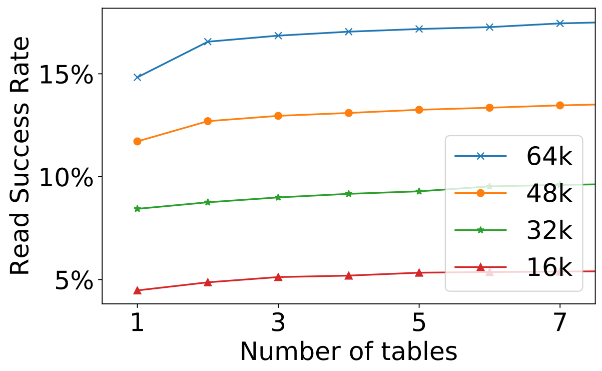

Figure 6: When splitting the same total memory across multiple tables, using S =3 or 4 tables yield the most significant improvement over using a single table, with diminishing return afterwards.

In the following experiments, we use 500 ms as the stale threshold (corresponding to 99^{th} percentile of all RTT samples), and investigated the success rate of our algorithm 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).

4.2 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% percent 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.

4.3 Optimal number of hash tables

We now investigate the optimal number of hash tables to use given a fixed total memory size, to quantify the benefit of using additional hash tables. In this experiment, we fix the total memory size and divide them by varying the number of stages; for example, splitting 64k records into $S = 2$ stages means having 32k records per table, while a configuration 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 for a multi-stage hash table, under a fixed total memory size constraint. Having more than four tables yields diminishing returns, as the memory in later stages is underutilized. Therefore, for practical implementation, we should choose $S = 3$ or $S = 4$ stages. 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. [19] explored the idea of using different table sizes per stage to achieve higher memory utilization; we leave this as future work.

5 CONCLUSION

We present an algorithm to track the per-packet Round-Trip Time of TCP traffic in the data plane of commodity programmable switches using a multi-stage hash table data structure. Our algorithm can successfully report 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 3 to 4 stages in the multi-stage hash table structure achieves the best performance for RTT monitoring, given the same amount of total memory.

We are in the process of deploying our algorithm on our campus network to provide continuous RTT monitoring for network operators. In the future, we will explore implementing more sophisticated measurement primitives based on the real-time RTT samples, including alerts for sudden change in RTT distributions and notifications for SLA violations.

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*. ACM, 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*. ACM.

[4] Francesco Bronzino, Paul Schmitt, Sara Ayoubi, Nick Feamster, Renata Teixeira, Sarah Wasserman, and Srikanth Sundaresan. 2019. Lightweight, General Inference of Streaming Video Quality from Encrypted Traffic. *arXiv preprint arXiv:1901.05800* (2019).

[5] Gaetano Carlucci, Luca De Cicco, and Saverio Mascolo. 2015. HTTP over UDP: an Experimental Investigation of QUIC. In *ACM Symposium on Applied Computing*. ACM, 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*. ACM, 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*. ACM, 1–15.

[9] 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.

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

[11] 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.

[12] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. 2018. Sonata: Query-driven streaming network telemetry. In *ACM SIGCOMM*. ACM, 357–371.

[13] 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.

[14] 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.

[15] 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.

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

[17] 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 Wksp. Real Overlays and Distrib. Sys* (2009).

[18] Brian Trammell, Piet Vaere, Roni Even, Giuseppe Fioccola, Thomas Fossati, Marcus Ihlar, Al Morton, and Stephan Emile. 2018. *Adding Explicit Passive Measurability of Two-Way Latency to the QUIC Transport Protocol*. Technical Report. IETF.

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

[20] 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.

[21] Verizon. 2019. IP Latency Statistics. (2019). https://enterprise.verizon.com/terms/latency/ Accessed: 2019-11-03.

[22] Verizon. 2019. Service Level Agreements. (2019). http://www.verizonenterprise.com/solutions/public_sector/state_local/contracts/calnet3/sla/ Accessed: 2019-11-03.