Princeton University

Department of Computer Science

Senior Thesis

Detecting Violations of Service-Level Agreements in Programmable Switches

Author:
Mack Lee
Princeton University
Undergraduate

Advisor:
Jennifer Rexford
Princeton University
Professor

Date of Submission: May 7, 2018

Table of Contents

Abstract

Service level agreements (SLA) are an integral part in the delivery of services between internet service providers and their customers. They define the quality of service (QoS) that a provider promises to the customer. The implications of monitoring SLA compliance benefits both providers and customers: customers can verify whether providers uphold their QoS guarantees and providers can take corrective action if a SLA violation is detected. However, current active network measurement techniques used to monitor SLA are not accurate and are not representative of the traffic between customers and providers. In this paper, we design an algorithm and data structure to passively measure latency/RTT such that it can be applied to monitoring SLA. Our algorithm uses constant time processing per packet and our data structure uses bounded memory, making our design hardware-friendly. We evaluate the performance of our methods using trace-driven simulations on a Python prototype.

Acknowledgments

This thesis represents the work I have done over the past year. I would like to recognize my friends and family who have helped me succeed in my time as an undergraduate at Princeton. I would also like to thank all the faculty that have mentored me as they have given me the tools and skills to achieve my goals and to grow intellectually.

I would like to thank Professor Jennifer Rexford and Xiaoqi (Danny) Chen for assisting me with my thesis. This thesis would not have been possible without their guidance and advice throughout the research process.

1. Introduction

Internet service providers use service-level agreements (SLA) to describe and quantify the quality of service (QoS) they offer [10]. SLA are legally binding contracts that promise and define a set of metrics regarding the delivery of services (e.g. latency, throughput, packet loss). SLA generally contain the following components: a description of the service, expected performance, and a penalty clause that includes the consequences if the provider does not meet the QoS guarantees. The purpose of SLA is to hold service providers accountable and give customers a legal contract associated to the service they receive. However, in practice, it is difficult to ensure that these standards are being upheld.

Detecting SLA violations is beneficial for both providers and customers. On one hand, it is desirable for providers to detect violations quickly so they can take corrective action. On the other hand, it is desirable for customers to detect violations to verify the quality of service they receive and potentially get their money back if there is a breach of contract. Sometimes providers have the tendency to cheat on SLA (e.g. provide less bandwidth to users than specified in the SLA) to make more money and support more users. Therefore, being able to reliably monitor and verify SLA compliance can benefit both providers and customers.

Verifying SLA compliance requires monitoring all network traffic and determining whether metrics defined by the SLA are true. Network measurement methods is generally classified into two different categories: active and passive. Active measurement uses probes that are sent across the network to gather information (e.g. measure the response time of an application or measure the time it takes to reach one end of the network). Passive measurement records packets without adding traffic to network via monitoring buffers in routers or duplicating link traffic in a hub or more recently, using programmable switches [9] [22].

Currently, verifying SLA is done using active measurement - telemetry packets are sent across the network using the same bandwidth as customers [13] [15] [20]. Active measurement methods have several drawbacks: probes can slow down the network, the data gathered by these probes are not representative of regular traffic, and probe results are not comprehensive. The emergence of new programmable protocol-independent network switches has motivated passive monitoring of network traffic in the data plane because of their flexible, high-speed packet processing capabilities. Passive measurement methods monitor all network traffic in real-time, making them more representative of network behavior. The advantage of conducting performance diagnostics on the data plane is that the data plane is able to detect issues more quickly and make quicker decisions for corrective action (e.g. load balancing for network-limited connections) [9]. That said, developing methods to run on a hardware switch has constraints including limited arithmetic and boolean operations per packet and limited storage for maintaining state. Given the large number of packets being transmitted and received per second, it is infeasible to store information about all of the packets. For example, the amount of data generated by passive measurement on an hour trace for a 1 gigabit link with utilization of 60% (in the IP layer) and average packet size of 300 bytes requires about 270 gibibytes of storage [15]. Even if we were to only IP and transport layer headers, we would still need 36 gb of storage space for an hour trace [15]. In addition to storage, it is important for passive measurement techniques to use a little processing per packet as possible. For a 100 Gbps connection, a passive monitoring scheme that captures all network traffic would require processing a packet about every 6.5 nanoseconds [1]. Considering time and space constraints, it is important to design passive measurement heuristics that have constant time processing per packet and use bounded memory.

In this paper, we present a hardware-friendly algorithm and data structure to passively monitor the latency of TCP traffic and verify SLA compliance in real-time, on the data plane.

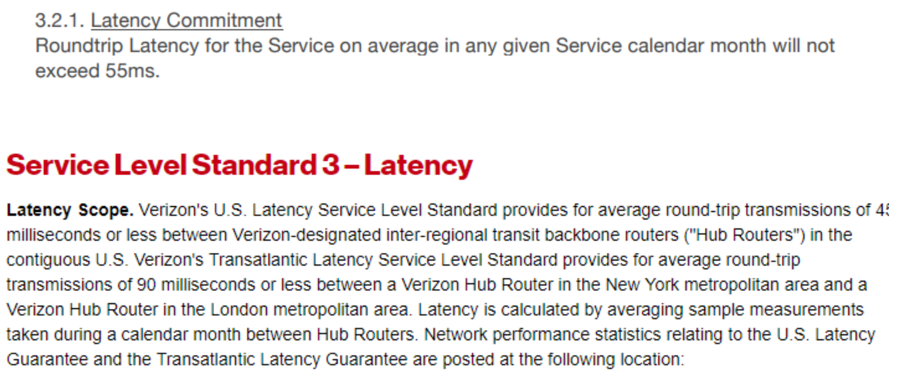
Motivation: The ability to passively detect SLA violations would allow providers to quickly respond to QoS issues and deliver better service to customers. Customers can be more confident about the service they receive from providers. It would ensure that the legal contract between providers and customers is not violated.

2. Problem Statement

2.1 Defining SLA

From investigating state-of-the-art SLA from service providers, we generalize SLA and translate them from plain English to a data structure. Most state-of-the-art SLA define violations in terms of different metrics (e.g. latency, throughput threshold) and guarantee a maximum fraction of violations within a time interval (e.g. days, hours, weeks). In this paper, we focus on monitoring **latency** guarantees as follows:

- For each $< SLA\_interval >$, we guarantee less than $< SLA\_percentage >$ violations.
- For each packet, an RTT greater than $< T >$ is a violation.



3.2.1. Latency Commitment
Roundtrip Latency for the Service on average in any given Service calendar month will not exceed 55ms.

Service Level Standard 3 – Latency

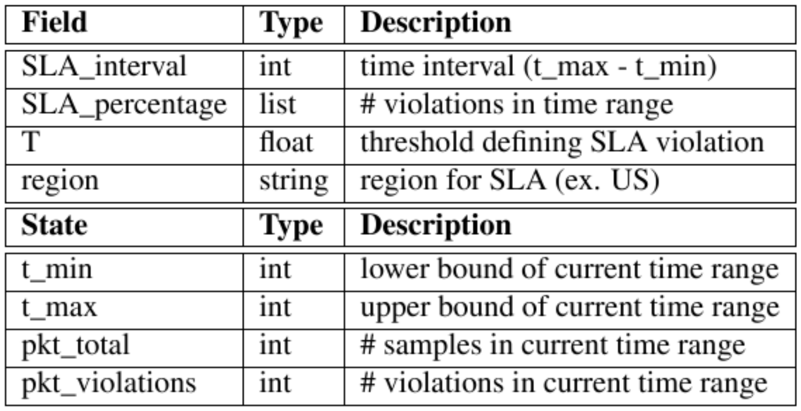
Latency Scope. Verizon's U.S. Latency Service Level Standard provides for average round-trip transmissions of 4 milliseconds or less between Verizon-designated inter-regional transit backbone routers ("Hub Routers") in the contiguous U.S. Verizon's Transatlantic Latency Service Level Standard provides for average round-trip transmissions of 90 milliseconds or less between a Verizon Hub Router in the New York metropolitan area and a Verizon Hub Router in the London metropolitan area. Latency is calculated by averaging sample measurements taken during a calendar month between Hub Routers. Network performance statistics relating to the U.S. Latency Guarantee and the Transatlantic Latency Guarantee are posted at the following location:

Figure 1. Examples of latency guarantees for state-of-the-art SLA from Comcast and Verizon. Each defines a latency threshold and a time interval at which the SLA applies [2] [3].

Usually SLA are defined across different regions (e.g. US, Europe, Asia) as different performance guarantees are possible depending on available resources and infrastructure. For example, an SLA for a specific region (e.g. US) may include:

- For every hour, we guarantee less than 99% SLA violations
- For each packet, an RTT greater than 50 ms, is considered a violation

Our abstraction of SLA allows us to keep track of whether the percentage of violations that occur within a timeframe is acceptable. The abstraction is broken down in to fields and state variables as shown in Table 1. Fields are constant whereas state variables change with each time interval.



Field	**Type**	**Description**
SLA_interval	int	time interval (t_max - t_min)
SLA_percentage	list	# violations in time range
T	float	threshold defining SLA violation
region	string	region for SLA (ex. US)
State	**Type**	**Description**
t_min	int	lower bound of current time range
t_max	int	upper bound of current time range
pkt_total	int	# samples in current time range
pkt_violations	int	# violations in current time range

Table 1: SLA abstraction with field and state variables

We update this SLA as packets are received using the logic in Figure 1. We keep track of the percentage of violations that occur within a certain time interval and update that time interval depending on the current time (extracted from packet timestamps). Henceforth, we use latency and roundtrip time (RTT) interchangeably.

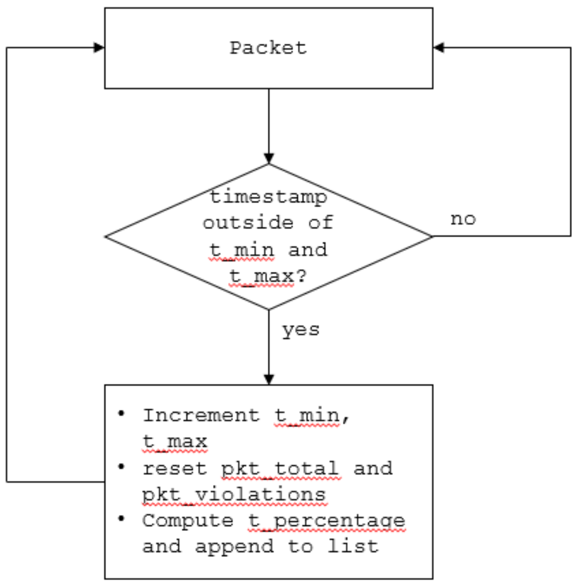

Packet
timestamp outside of t_min and t_max?
no
yes
• Increment t_min, t_max
• reset pkt_total and pkt_violations
• Compute t_percentage and append to list

Figure 2. Flowchart describing the logic used to update SLA abstraction

Monitoring SLA is useful for both customers and service providers. A customer offering a service (e.g. CDN) could be interested in knowing whether an ISP is keeping its promises. An ISP would want to take action if a customer is at risk of having violated a SLA.

2.2 Scalable Passive Monitoring of RTT

We develop algorithms and data structures to passively monitor the latency of TCP packets in real-time on a hardware switch. We focus on TCP traffic because it accounts for the majority of internet traffic. We uniquely identify a TCP/IP connection using a flow ID which contains the bi-directional 4-tuple and sequence/acknowledgment (SEQ/ACK) numbers extracted from packet headers. More specifically, the flow ID consists of the following packet headers:

- Source IP
- Destination IP
- Source Port
- Destination Port
- SEQ/ACK numbers

The 4-tuple in the **flow ID** is bidirectional because a sender and receiver will have the same combinations of IP addresses and port numbers but with the source and destinations swapped. We identify both the sender and receiver with the same flow ID.

We assume that we are running a passive monitoring scheme on a network switch that is close to a host. Monitoring close to one end is representative of how SLA will be monitored by a customer or provider (near the end hosts). It also removes overhead and the need to keep per-flow state for both ends of each flow. The network switch monitors **outgoing packets** and **incoming ACK**s where each packet-acknowledgment pair produces an RTT sample that is either:

1. **High-latency:** sampled RTT exceeds the latency threshold ($\Delta t > T$)
2. **Low-latency:** sampled RTT does not exceed the latency threshold ($\Delta t \leq T$)
3. **Unknown**: sampled RTT may not be valid or representative.

Note that a high latency packet that is above the RTT threshold *T* is different from a **SLA violation** which limits the percentage of violations within a certain time frame (see section 2.1).

We rely on the packet-acknowledgment pairing and match outgoing packets with their incoming ACKs (differentiated using TCP flags and payload length). For each outgoing packet, we use the flow ID and **expected ACK number** (e-ack) which is calculated from the outgoing packet's size and SEQ number to match incoming ACKs with the stored outgoing packets. The difference between the timestamps of the outgoing packet and its corresponding ACK, (Δt) is used to calculate a RTT sample.

It is not feasible to monitor all network traffic at line rate because of the memory and computational restrictions imposed by hardware as well as some of the vagaries of TCP. Therefore, it is important to consider how packets are sampled. We must be unbiased in how packets are sampled in determining SLA compliance. Bias towards high latency packets could lead to an incorrect classification of a SLA violation while bias towards low latency packets could lead to a missed classification of a SLA violation. We want to avoid saying an SLA is violated when it is not and vice versa. Therefore, the **sampling rate** (the percentage of network traffic sampled) is important since memory restrictions make it impossible to be able to monitor all packets within the time-frame defined by the SLA. We also want to ensure that we are detecting all high latency packets accurately. Therefore, we use **true positive rate** which is the percentage of high latency packets that are detected by our algorithms and data structures.

3. Accurate Detection of SLA Violations

In this section, we describe an ideal algorithm to passively measure RTT. We define an algorithm to be ideal if it makes all possible RTT measurements with 100% accuracy using the minimum amount of memory. We assume that each outgoing packet receives a matching incoming ACK.

3.1 Ideal Algorithm

An easy, intuitive solution to measuring RTT is to take each outgoing packet and store the flow ID, expected ACK, and timestamp (t) in a data structure without memory constraints (e.g. dictionary). With this information, we can then match incoming ACKs with outgoing packets and calculate RTT as the difference between their timestamps. This will guarantee 100% accuracy since we assume that there is a 1-to-1 correspondence between packets and ACKs. ACKs that match an outgoing packet stored in the data structure are defined as a **matched incoming ACKs** and those that do not are defined as **unmatched incoming ACKs**. Outgoing packets that have been stored for longer than the RTT threshold are considered **expired**.

Maintaining the assumption that every packet is guaranteed to receive an ACK, we can implement the ideal algorithm using a queue. The queue stores outgoing packets that have not exceeded the *T*. If a packet is stored for longer than the *T*, then it is considered expired and removed. This allows incoming ACKs to determine whether their corresponding outgoing packets exceed or do not exceed the threshold based on whether or not they are in the queue. We describe how outgoing and incoming packets are handled in detail below:

- **Expired packet:** Remove from queue since the latency exceeds *T*
- **Outgoing packet:** Record the flow ID, expected ACK, and timestamp and add <fid, e-ack, t> to back of the queue.
- **Matched incoming ACK:** The corresponding outgoing packet has not exceeded the latency threshold so it is classified as low-latency. Remove matched packet from queue.
- **Unmatched incoming ACK:** The corresponding outgoing packet must have been expired so it is classified as high-latency.

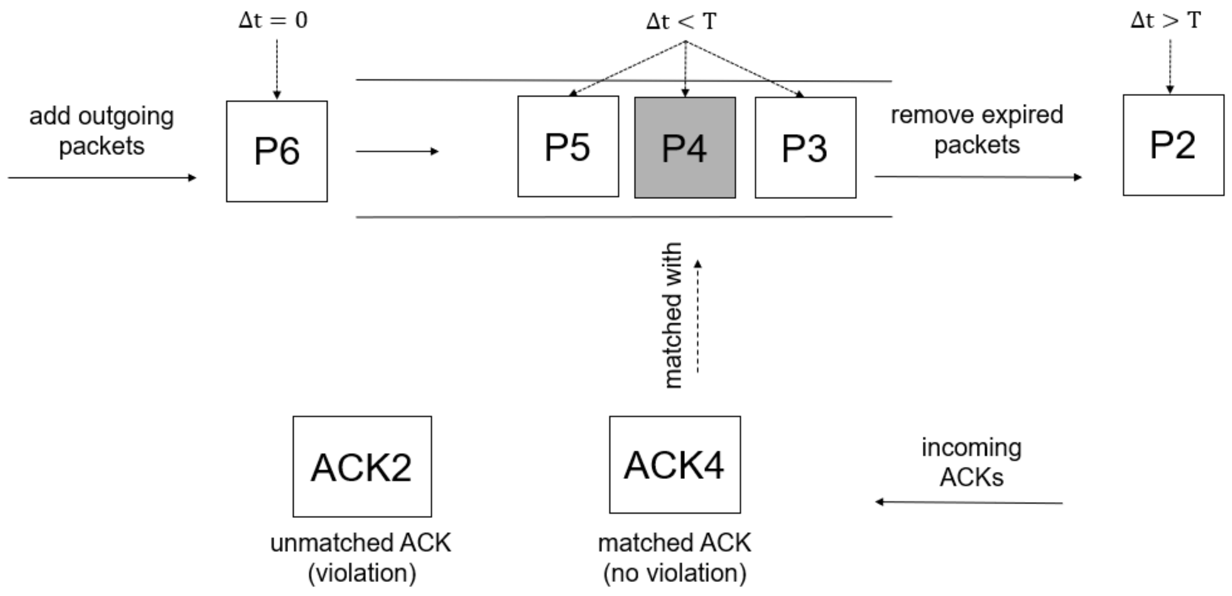


Figure 3: Queue in the ideal algorithm. Outgoing packets (P6) are added to the tail of the queue. Expired packets are removed from the queue (P2). Incoming ACKs that match an outgoing packet in the queue do not exceed the latency threshold and are low-latency (P4-ACK4). Incoming ACKs that do not match an outgoing packet in the queue do exceed the latency threshold and are high-latency (ACK2) since the outgoing packet (P2) must have expired.

The queue achieves 100% accuracy because the arrival of an ACK will produce a high or low latency classification based on whether or not the matching packet is or is not in the queue, which is equivalent to whether the matching packet has or has not exceeded the RTT threshold. It also does not use any more memory than needed because it only keeps track of packets that have not already exceeded the RTT threshold.

3.2 Delayed and Cumulative ACK

The ideal algorithm in Section 3.1 does not account for the fact that, in TCP, acknowledgments can be delayed, aggregated, or lost. Hosts often combine multiple ACKs into one packet to reduce load on the network as shown in Figure 4. This is possible because TCP implements “cumulative ACK” which implies that a receiver has received all previous packets successfully.

TCP acknowledgments can also be delayed because of delayed ACK. Delayed ACK is a mechanism in which the receiver waits to receive a certain number of bytes before sending an ACK within a certain amount of time (delayed ACK timeout). Holding ACKs up to the timeout gives the receiver the opportunity to piggy-back ACKs on outgoing data packets. Because subsequent data packets is common in data transfer (e.g. loading a webpage), delayed ACK reduces overhead and improves network performance by giving hosts a timeframe to aggregate ACKs.

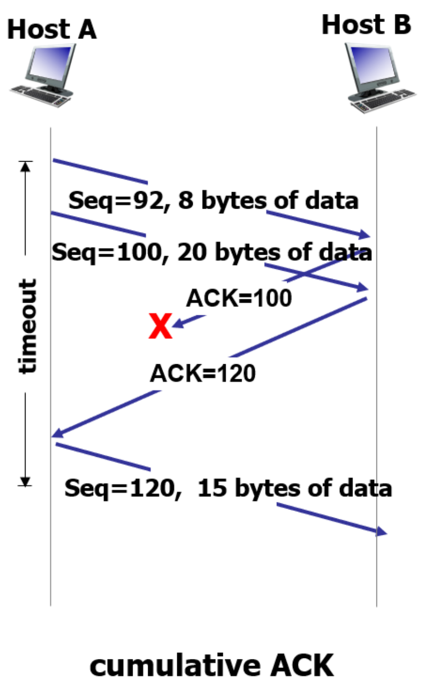


Figure 4: TCP cumulative acknowledgment. Two data packets being sent from host A are acknowledged by one ACK packet from host B [8].

We discussed the issues that cumulative and delayed ACK cause with respect to the algorithm described in Section 3.1. Previously, we assumed that each outgoing packet receives a matching incoming ACK. If a receiver aggregates multiple ACKs into one packet because of cumulative ACK, we can no longer make that assumption. Thus, if a matched incoming ACK is meant to acknowledge multiple packets, we must do extra work to check to check if it corresponds to other previous outgoing packets in the queue if we want to sample them (discussed in section 4.3). However, if an outgoing packet experiences delayed ACK and exceeds *T*, we do not want to count that as a sample. Delayed ACK may cause RTT samples to exceed the latency threshold, leading to high latency packets that are not at the fault of the service provider but rather the TCP protocol. Classifying such packets as high-latency is erroneous. It is difficult, from passive observations of packets alone, to know whether a receiving host used delayed ACK in a RTT sample. However, the likelihood that it experienced delayed ACK can change depending on the size of the packet. Data transmissions often contain subsequent full-size packets followed by a small packet. It is likely that the small packet would experience delayed ACK since the receiver likely did not receive enough data to trigger an acknowledgment. This scenario is shown in Figure 5, where a server receives a full-size packet followed by a small-size packet but does not send an acknowledgment because it continues to wait for more data to reach its threshold

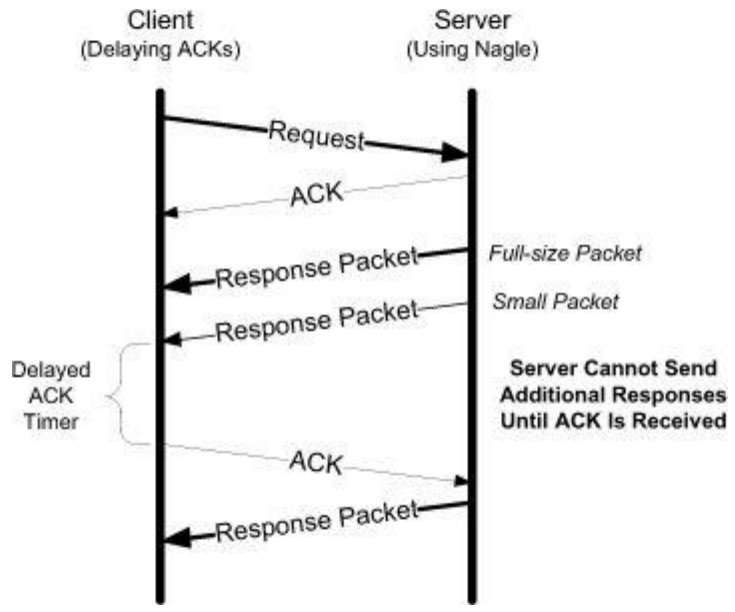


Figure 5: Delayed ACK scenario where a client sending a full-size packet followed by small packet does not trigger acknowledgement from the server [7].

To avoid mistaking packets that experience delayed ACK for high-latency packets, we ignore outgoing packets that likely experienced delayed ACK by performing an extra check on the size of the matched outgoing packets. Each flow in a TCP connection agrees upon a maximum segment size (MSS) that is determined during the TCP handshake which caps the amount of data that can be transmitted in a single packet. We rely on this header and filter out packets that are smaller than the MSS. To avoid missing a potential high latency packet, we revise the threshold for an expired packet (currently equals *T)* and add the delayed ack timeout T_{da}. We can only be certain that an outgoing packet that has not received an ACK is high-latency if it has exceeded $T + T_{da}$. We revise the algorithm from section 3.1 and assume that receiving hosts use delayed ACK as follows:

- **Expired packet:** Remove from queue if $\Delta t > T_{da} + T$ since we can be certain that the packet experiences high latency not just due to delayed ACK.
- **Outgoing packet:** Record the flow ID, expected ACK, and timestamp and add <fid, e-ack, t> to back of the queue.
- **Matched incoming ACK:** If the size of the matched outgoing packet is <MSS, then we remove it from the queue since it might have experienced delayed ACK. If not (=MSS), then calculate Δt and remove matched packet from the queue. If $\Delta t > T,$ then we have detected a high latency sample.
- **Unmatched incoming ACK:** The corresponding outgoing packet must have been expired so we have detected a high latency sample.

4. Hardware-friendly Algorithm

Implementing the algorithm described in Section 3.2 in hardware is not feasible due to memory restrictions and limited computing resources. Passively checking for expired packets is too expensive given the throughput of real traffic. In this section, we describe data structures and algorithms that are hardware-friendly.

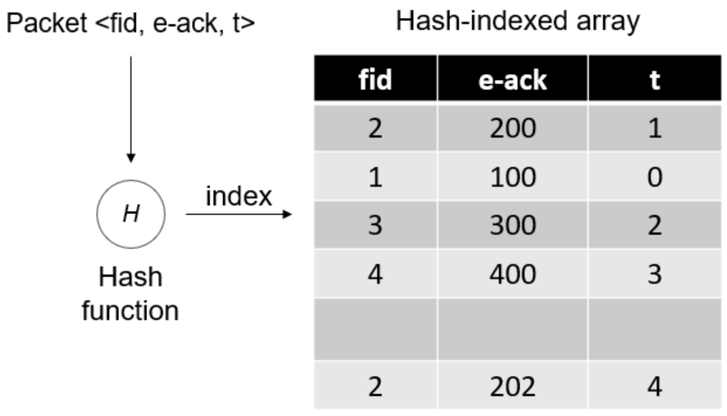
4.1 Hash-Indexed Array of Packets

Since we need constant time processing per packet, we replace the queue of packets from Section 3.2 with a hash-indexed array. Outgoing packets are stored in the array by hashing the packet's <fid, e-ack> to generate an index within the length of the array and storing <fid, e-ack, t>. Henceforth, we assume that hashing a packet <fid, e-ack, t> means only hashing <fid, e-ack>. Incoming ACKs are hashed in the same way to generate an index and are checked against the outgoing packet stored at that index. Henceforth, in the following examples, we assume that the arrays are indexed starting from 1.

As mentioned earlier in section 3.2, we no longer assume that each outgoing packet is guaranteed to receive a corresponding ACK because of cumulative ACK. In section 3.2, we removed expired packets and assumed that an unmatched incoming ACK produce a high-latency sample. However, since we aim to remove the overhead in passively checking for expired packets, we revise this heuristic and store each outgoing packet for long enough for them to receive an ACK even if it takes longer than the *T*. How long should we wait? Since high latency packets are only counted when a matching incoming ACK has been received, we must define a new threshold $S > T$ where *S* represents the time at which we assume an outgoing packet will not receive an ACK. We will call *S* the **stale threshold.**

How do we remove the check for expired packets? An important invariant of this data structure is that hash collisions gives the opportunity to check how long a packet has waited for a matching ACK. If an outgoing packet's hash matches a packet already stored at that index (collision), then the stored packet has not received an ACK yet. Based on this knowledge, we can make decisions on whether a packet will most likely not receive an ACK on hash collisions. We describe how outgoing and incoming packets are handled:

- **Outgoing packet:** If packet size is full size (=MSS), hash the packet <fid, e-ack, t> to produce index. Check to see if a packet is already stored. We only store full-size packets since small packets are likely to experienced delayed ACK.
    - **collision:** If $\Delta t > S,$ evict current entry since it is stale and we assume it will not receive an ACK. Replace with new packet. Otherwise, ignore the new packet (drop the sample).
    - **no collision:** Insert packet at index.
- **Incoming ACK:** Hash the packet <fid, e-ack, t> to produce index. Check if it corresponds to an outgoing packet stored in the array.
    - **matched:** If $\Delta t > T$, then the sample is high latency. Else, it is low-latency. Remove the stored packet.
    - **unmatched:** The corresponding outgoing packet must have been evicted so we have detected a high latency sample.



fid	e-ack	t
2	200	1
1	100	0
3	300	2
4	400	3
2	202	4

Figure 6: Hash-indexed array showing a packet's flow ID, expected ACK, and timestamp being hashed to compute an index into the hash-indexed array. The array stores outgoing packets that have not received an ACK.

We illustrate the described algorithm using the values in the hash-indexed array shown in Figure 6. We see that outgoing packet <2, 202, 4> was hashed into index 6 and stored there. If we receive an incoming ACK, then the hash of <fid, e-ack, t> is computed and the entry in the resulting index is checked for a match. If the incoming ACK was <2, 202, 8>, then it will be hashed to index 5 and checked against the packet currently stored there. Since the fid and e-ack match, the time difference Δt would be calculated, producing a RTT sample of $\Delta t = 4$.

However, if a hash collision occurs (ex. if the incoming ACK hashed to index 6 in Figure 6) and the stored packet has not exceeded S, then the outgoing packet would not be sampled. This could be problematic since it would decrease the sampling rate. Missing a sample could lead to an incorrect conclusion that the SLA is violated (e.g. if the sample could lead to the difference between exceeding or not exceeding the percentage of violations defined by the SLA). This issue motivates the need for the optimization in the following section: using multiple stages of hash functions.

4.2 Multiple Stages of Hashing

The next step in our design is to maximize the sampling rate by reducing the number of hash collisions. One common technique in data plane switching is to handle hash collisions by splitting the hash-indexed array into d disjoint arrays of equal size indexed with d different hash functions. Note that this does not use any more memory than in the hash-indexed array in Section 4.1 (where $d = 1$) since we split the fixed memory allocation. With multiple stages of hashing, instead of dropping outgoing packets as a result of collisions, we hash the outgoing packet into another array using a different hash function. The trade-off is that storing outgoing packets and matching incoming ACKs are more expensive since it requires d lookups instead of 1 lookup because each array needs to be checked. However, since d is a fixed, lookups are still done in constant time and can be pipelined in a network switch. That said, depending on d, this design could introduce excessive hash collisions and dropped samples since each stage would have less memory to work with. Therefore, d must be tuned depending on the nature of the network traffic.

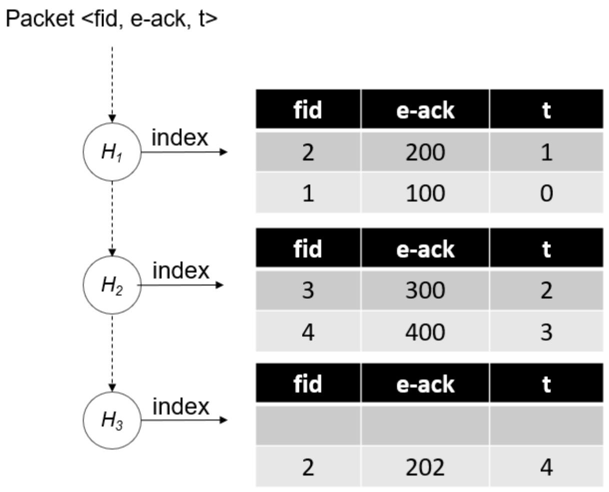


Figure 7. 3-stage hash-indexed array. A packet is hashed into the next stage if the packet cannot be stored in the previous stage as a result of hash collision.

Figure 7 shows a d-stage hash-indexed array with three stages. Outgoing packets <1, 100, 0> and <2, 200, 1> were hashed using H_1 and stored into the first array. When outgoing packet <3, 300, 2> arrived, it collided with one of the entries in the first array so it is hashed using H_2 and stored in the second array. When the ACK arrives for that packet (ex. <3, 300, 6>), it is hashed using H_1 and a match is not found in the first array. Then it is hashed using H_2 and a match is found so Δt is calculated to be $6 - 4 = 2$. The stored packet <3, 300, 2> would then be removed (not shown). The hash-indexed array in section 4.1 is a degenerate case of the d-stage hash-indexed array where $d = 1$. Note that if $d = \infty$, each stage would have an array of size 1 and would achieve perfect correctness as no outgoing packets would be dropped.

4.3 Handling Cumulative ACK

Since incoming ACKs are often meant to acknowledge multiple outgoing packets because of cumulative ACK, we describe a routine that removes all packets acknowledged by the incoming ACK as opposed to just the matched packet. Henceforth, we will call this the backtracking routine. Currently, if an incoming ACK is meant to acknowledge two outgoing packets in the data structure, only the most recent outgoing packet is removed while the other will stay in the array until a hash collision occurs after it has been stored for longer than S.

On each matched incoming ACK, using the size of the stored packet, we can infer the expected ACK of the previous packet by subtracting the size from the acknowledgment number of the incoming ACK. We can then hash and check the array for the previous packet and remove it if a match is found. This requires storing the size of each packet in addition to <fid, e-ack, t>. Figure 8 shows the revised hash-indexed array with an added field to store the size.

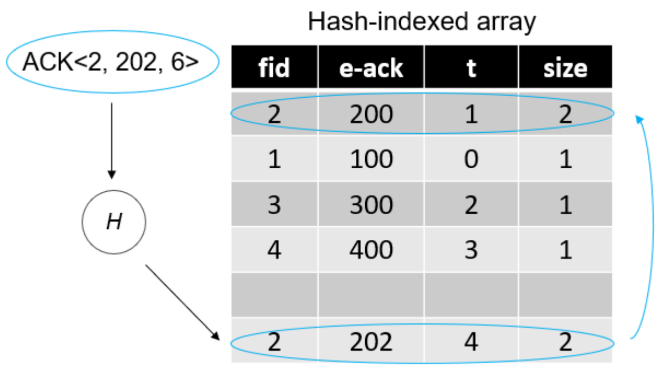


Figure 8: Backtracking routine when incoming ACK matches last entry.

We demonstrate through example how the backtracking routine handles cumulative ACK. In Figure 8, suppose that an incoming ACK <2,202,6> matches the packet stored in the last index of the array. Since the size of the stored packet is 2, we can subtract 2 from the expected ACK of the stored packet to figure out the expected ACK of the previous packet in the flow. We then do a lookup on the previous packet to see if the incoming ACK is meant to acknowledge the previous packet as well. In the example, the incoming ACK also matches the first entry in the array so the first entry would be also be removed by the backtracking routine. This results in a more efficient use of memory as outgoing packets acknowledged by cumulative ACK are properly removed so there will be fewer dropped samples.

5. Python Prototype

We built a prototype to run trace-driven simulations to analyze these algorithms and data structures using Python as the high-level language for implementation. In our prototype, we assume that TCP packets include Ethernet, IPv4, and TCP headers, though it can easily be changed to support other protocols such as IPv6. Some of the libraries that were used included:

- scapy: Provides libraries and functions used to process and abstract packet headers
    - https://scapy.net/
- mmh3: Implementation of the murmur hash
    - https://pypi.org/project/mmh3/2.0/
    - Murmur hash was used to ensure an even distribution of hashes such that the indices they produced would be uniformly distributed.
- pygeoip: Libraries for region lookup based on IP address
    - https://pypi.org/project/pygeoip/

Simulations were run using Python version 2.7 on a 2016 Macbook Pro running MacOS 10.13.3 and a 3.3 GHz Intel Core i7 CPU. The total lines of code that includes the implementations of the revised ideal algorithm in section 3.2 and the *d*-stage hash-indexed array in section 4.2 totals to about 800 lines of code.

6. Evaluation

In this section, we discuss the evaluation of the performance of our algorithms and data structures. The revised ideal algorithm in section 3.2 serves as a ground truth in evaluating the performance of our constant-memory algorithms. In section 6.1, we describe the tuning of different parameters. In section 6.2, we analyze the tradeoff between accuracy and memory overhead and the performance improvement of the backtracking routine described in section 4.3.

Experiment setup: We tested our methods by running simulations using the Python prototype described in section 5 on a university data center trace. This trace used was recorded from a university data center in 2010, consisting of approximately 100 million packets with about 10,000 packets per second [5]. The primary use of this data center is for students and faculty of the university, providing services including Web services, E-mail servers, and distributed file systems. The university trace was used because the data center is located close to the students which is suitable for the application of our algorithms in verifying SLA compliance. We set T=50 ms since it is common among SLA in the US and would provide enough high latency packets to better diversify our metrics with different parameters.

Metrics: We define memory use by the number of entries used in the data structures described in section 3 and 4. We are also interested in minimizing the number of dropped samples in our methods so we use sampling rate to measure the confidence in our decisions on whether the percentage of high latency packets exceeds the threshold defined in a SLA or not. We use true positive rate and false negative rate to measure the number of high-latency packets that are correctly classified as high-latency in our algorithms. Henceforth, we refer to use accuracy and true positive rate interchangeably.

6.1 Tuning

Number of stages: Assuming we allocate a fixed amount of memory for our data structures, we tune the number hash stages *d* and divide the memory equally among each stage. As *d* increases, the amount of memory for each stage decreases but allows for more collisions to occur. However, if *d* is too high, a small memory allocation for each stage could lead to an unnecessarily high number of collisions as well. We analyze the tradeoff between the number of stages and accuracy.

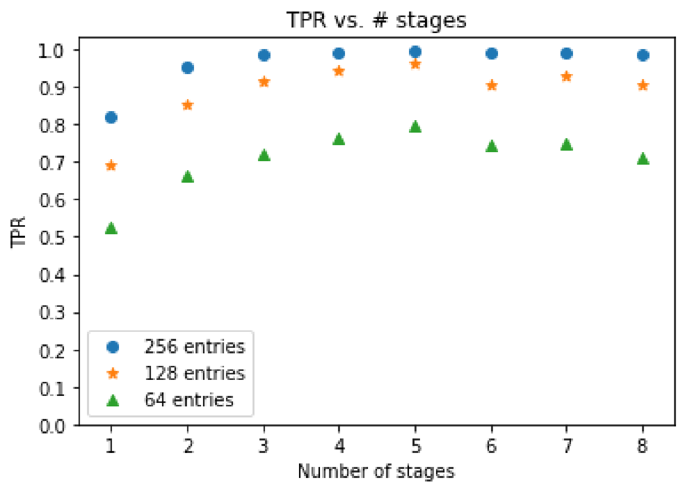


Figure 9: True positive rate vs. number of stages for three fixed memory values.

Figure 9 shows how the number of stages *d* impacts accuracy for different fixed memory allocations. We plot true positive rate vs. memory tradeoff for 1,2,5, and 8-stage hash-indexed arrays in Figure 10. We observe that as *d* increases from 1-5, true positive rate improves. However, as *d* increases from 5-8, the true positive rate worsens. This is consistent with our expectation that while a higher number of stages allows more hash collisions to occur with fewer packet drops, using too many stages could lead to an unnecessarily high number of collisions and an inefficient use of memory. We find that using *d* = 5 stages yields the highest accuracy for all tested memory allocations and gives the optimal balance between allowing for more hash collisions while maintaining appropriate size for each stage. This balance is trace-dependent as a different traffic may require more stages or more memory for each stage (e.g. higher throughput, many flows with high traffic). It is clear that the *d* = 1 case is described in section 4.2 exhibited the worst true positive rate, showing that applying *d*-stage hashing improves performance. Henceforth, we use *d*=5 for all simulations.

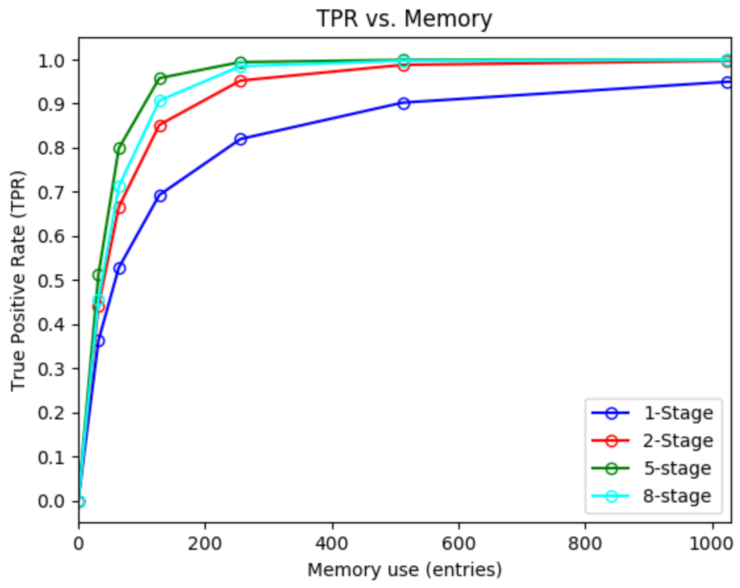


Figure 10: True positive rate vs. memory use for different values of *d*.

Stale threshold: The stale threshold that yields the best performance is dependent on the trace. For example, a trace with overall higher latency would require a higher stale threshold to ensure that more packets are sampled. We tune the stale threshold based on false negative rate (the number of high-latency packets that are not detected) since we aim to minimize the number of missed high-latency packets. In theory, increasing the stale threshold would give outgoing packets more time to receive an ACK whereas decreasing the stale threshold may lead to removing outgoing packets too early before they receive their corresponding ACKs. We plot the false negative rate vs. stale threshold S for 100 entries in Figure 11. Since we want to minimize false negative rate and detect high-latency packets at high accuracy, a stale threshold of $1.6 * T$ is chosen to be optimal. This is consistent with the nature of the packet trace as Figure 12 shows the median RTT per flow. There is a high percentage of packets that have a RTT of around 80 ms. Therefore, setting a threshold that captures all of these samples just long enough for them to receive their corresponding ACK packets would be most memory-efficient. Henceforth, unless otherwise mentioned, we use $S = 1.6 * T$ where *T=50 ms*.

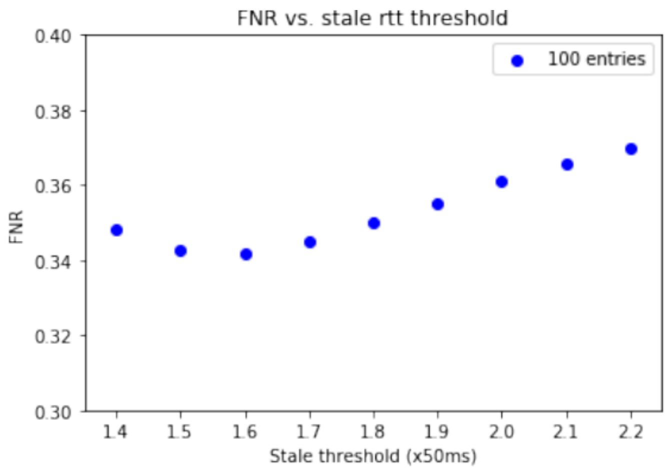


Figure 11: False negative rate vs. *S* for 100 entries.

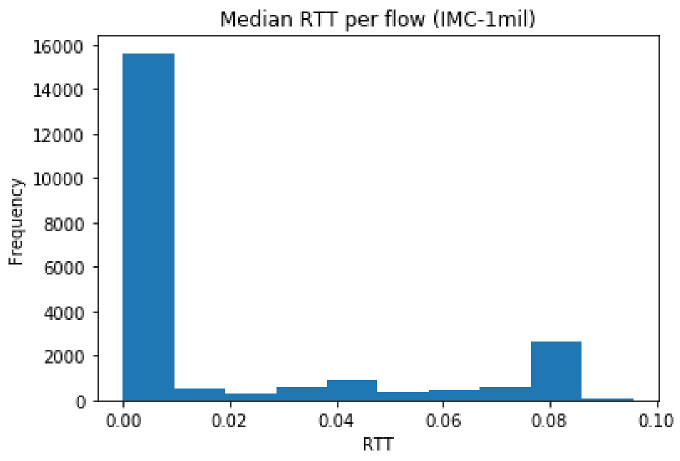


Figure 12: Distribution of median RTT per flow for a sample of the data center trace.

6.2 Memory and Performance Tradeoff

We plot the true positive rate vs. memory tradeoff in Figure 13 using the tuned parameters mentioned in Section 3.1 for the d-stage hash-indexed array. The true positive rate converges to 100% as memory increased. We expect this since increasing memory allows for the storage of more outgoing packets without suffering from the loss of samples. Overall, the performance of the d-stage hash indexed array with respect to true positive rate (correctly detecting high latency packets) is high. We achieve nearly 100% accuracy only using about a quarter of the maximum amount of memory used in the ideal case (~300 entries vs. ~1200 entries). In Figure 14, we plot the sampling rate vs. memory use for different values of S to show how different stale thresholds can impact sampling rate. The value $S = 1.6 * T$ yields the highest sampling rate for memory allocations up to ~125 entries. With the allocated memory exceeds ~125 entries, a higher stale threshold (S=3.0*T in the figure) begins to yield a higher sampling rate. Once the data structure has enough memory to samples to the point that collisions are not likely to occur, a higher stale threshold becomes advantageous. We achieve a sampling rate of around 40-45% which is high considering that most network hosts implement cumulative ACK. Only packets that produce RTT samples are considered in the sampling rate. If there are packets that are ACK'd by a cumulative ACK, then those are not counted in the sampling rate. Most systems combine two packets into one ACK, so a sampling rate of 40-45% means we consider most of the packet trace.

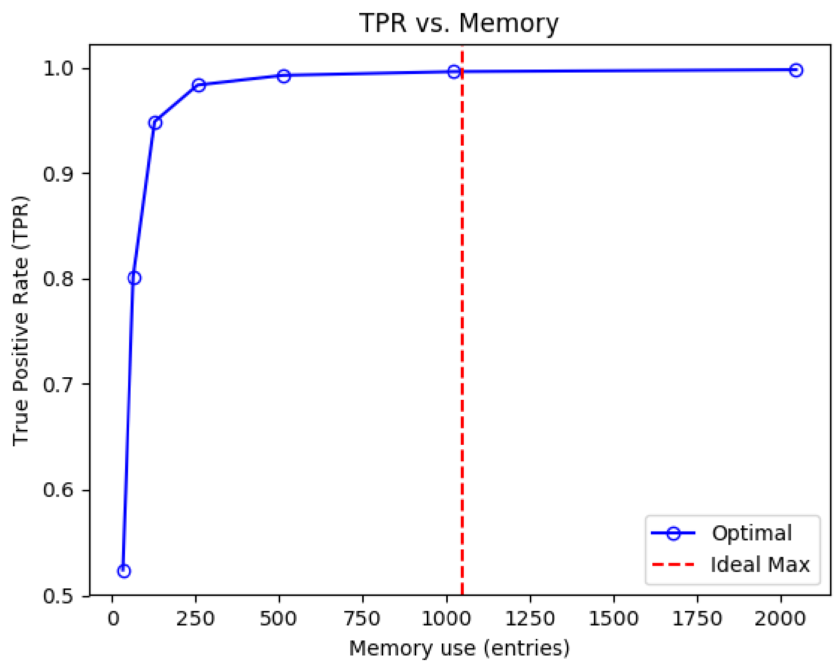


Figure 13: True positive rate vs. memory use using tuned parameters (blue) compared to maximum usage of the ideal algorithm (red).

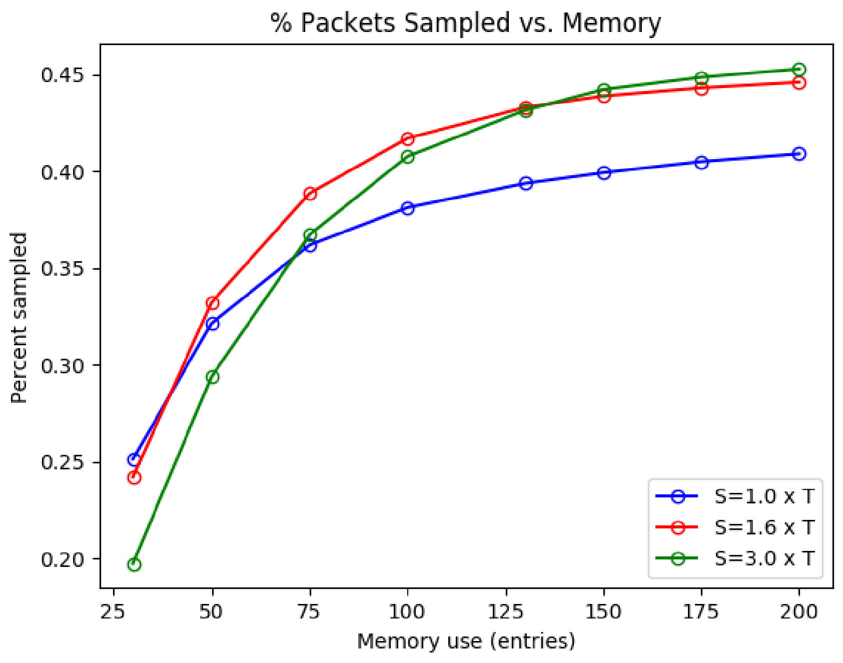


Figure 14: Sampling rate vs. memory for different values of *S*.

6.3 Performance of Backtracking

We now evaluate how the backtracking routine described in section 4.3 boosts performance with respect to sampling rate and true positive rate. We expect backtracking to yield more true positives and a higher sampling rate for a fixed memory allocation since the routine removes outstanding packets that have already been acknowledged by a cumulative ACK. In Figure 15, we plot the true positive rate vs. memory tradeoff with and without backtracking. The backtracking routine strictly improves true positive rate for a given memory allocation by 3-4% and the improvement is greater for lower memory allocations. With less space to store packets, the impact of keeping packets that will never receive an ACK in the array increases since more samples would be dropped.

We plot sampling rate vs. *S* with and without backtracking for a fixed memory allocation in Figure 16. We see that the impact of backtracking which removes outgoing packets that are acknowledged by cumulative ACK increases the sampling rate by 3-4%. Without backtracking, the impact of using a higher *S* decrease the sampling rate more than with backtracking. Backtracking creates more space for outgoing packets to be stored so having a higher stale threshold will more slowly impact the sampling rate when backtracking is not used.

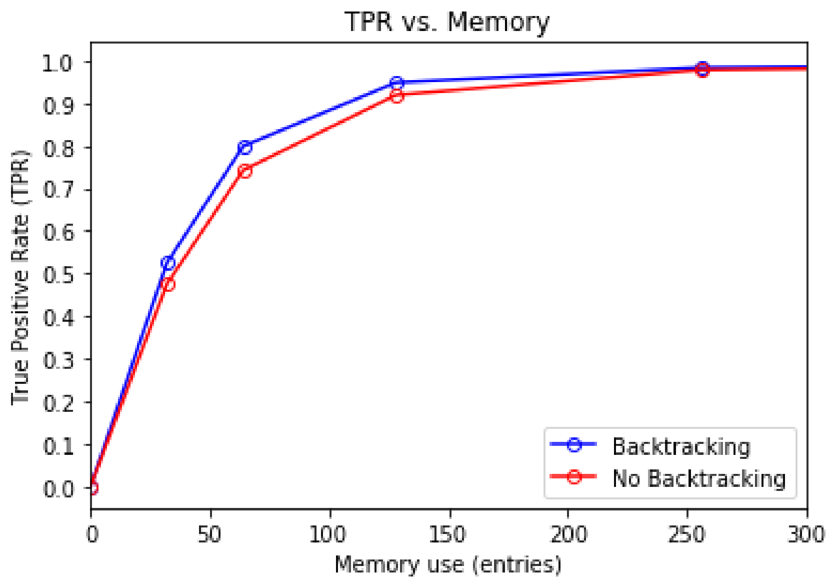


Figure 15: True positive rate vs. memory use with and without backtracking.

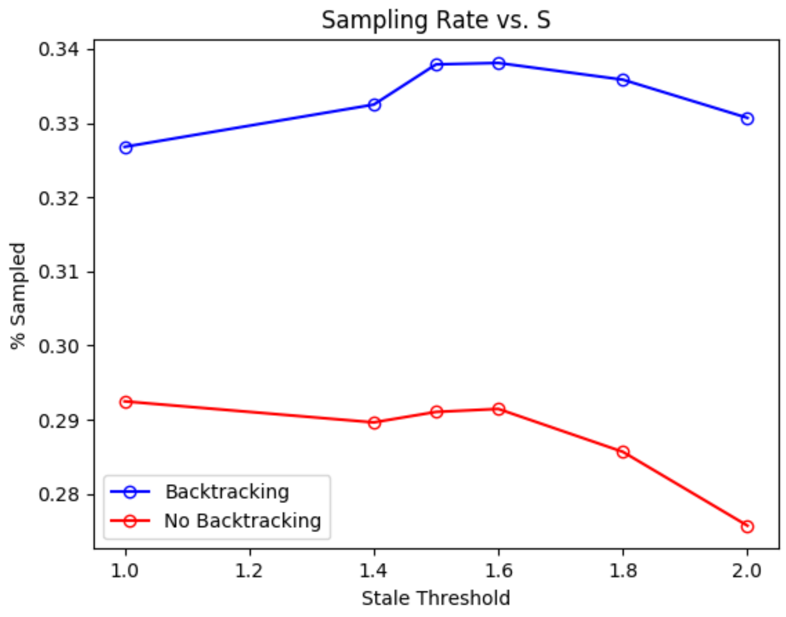


Figure 16: Sampling rate vs. *S* with and without backtracking using a memory allocation of 50 entries in the array.

7. Related Work

Several works have explored passively monitoring TCP performance. The emergence of programmable switches has made network monitoring at line rate a possibility, allowing for network measurement in real-time on the data plane [11]. Researchers have begun to design and develop algorithms to take advantage of this technology. Ghasemi et al. developed a system called Dapper that analyzes TCP performance in real-time near the end-hosts [9]. Dapper detects TCP bottlenecks and then focuses on diagnosing those troubled connections. They built a prototype in P4 (a protocol independent language that allows a user to specify how packets are processed and forwarded) and tested it on real and synthetic network traffic, finding that performance monitoring in the data plane removes the overhead of packet capturing and processing from the end-hosts, allowing the tool to diagnose results in real-time [9].

The main issue with passive measurement is the amount of memory required to store packet-level information. Researchers have sought to reduce the amount of storage and overhead needed for passive measurement. Some researchers have experimented with to compressing packets using compression algorithms (e.g. gzip) and removing unneeded fields such as the packet payload and other headers depending on the metric of interest [16] [17]. Peuhkuri compressed packet storage by only storing differences between consecutive packets in the same flow [17]. Sampling traffic is also a common technique to reduce the amount of overhead required for passive network measurement [15]. However, not all sampling techniques yield good results so it is important to ensure that sampling is unbiased and performs well for the application it is applied to [6]. For instance, Paxson et al. implementing a sampling technique that had difficulty analyzing flow-level statistics [16].

Although there has been a lot of work in passive network measurement, because of the amount overhead required, monitoring network performance in practice is mostly done actively by injecting packets with the purpose of gathering flow or network statistics into the network. For example, one active measurement method measures available bandwidth by sending probes at increasing rates and monitoring the rate that packet delays increase [19]. Some have explored the use of the OpenFlow protocol and SDN controllers to generate measurement probes and direct them around the network [20]. Kim et al. used P4 to prototype an abstraction that uses data packets to gather information related to the state of a network switch and diagnose performance problems [13].

Significant work exists on monitoring SLA has focused on architectural design because of the challenges inherent in monitoring a large system. One successful architecture that is implemented is service oriented architecture (SOA) which is where services are treated as independent entities in a system that rely on each other [4]. Monitoring SLA between services in these systems is difficult because a single service malfunction could lead to issues with the entire system [4]. Some researchers have designed an architecture that takes these into account by introducing entities that measure, analyze, and make decisions [4]. Another challenge with SLA monitoring is that the notion of verifying SLA compliance usually emerges after services have been deployed. Xiao et al. designed a framework to monitor SLA when a service architecture is being designed rather than after it has already been deployed [23]. They built a prototype that allows system integrators to verify nonfunctional requirements defined in SLA when composing services [23].

Because carriers use different metrics and depending on the type of service (SLA with internet service providers use different metrics than data center providers), other work related to

SLA focuses on developing a common language and structure for SLA. Lamanna et al. investigated end-to-end quality of service and introduced a language for defining SLA [14]. Rana et al. focused on identifying different types of penalty clauses in SLA and how they can be enforced from SLA monitoring [18]. SLA specifications can also be based on the type of service such as the Web Service Level Agreement framework designed for web services [12]. SLA extends further than internet service providers. Cloud service providers also use SLA to specify the resources (e.g. computing and storage) and performance they offer [24]. Ye et al. developed an algorithm to detect whether cloud service providers violate SLA clauses related to the physical memory size in the virtual machine [24]. They find from real experiments that their algorithm can detect malicious cloud service providers who attempt to conceal SLA violation [24]. There has not been significant work in utilizing passive measurement for the specific application of monitoring SLA compliance.

In Section 4.2, we utilize a common data-plane switching technique to split arrays into multiple arrays indexed by d different hash functions. Sivaraman et al. designed a detection algorithm to identify flows with large traffic volumes which uses a pipeline of hash tables similar to the d-stage hash-indexed arrays used in Section 4.2 [21].

8. Conclusion

In this paper, we proposed an algorithm and data structure to passively measure RTT within the memory and processing constraints of hardware switches. Our algorithm measures RTT by using multiple hash-indexed arrays each indexed by a different hash function to store outgoing packets and matching incoming ACK packets. We take the difference between their timestamps to produce RTT samples. We built a prototype in Python and evaluated our method using a real university datacenter trace and an ideal algorithm as a ground truth, showing that we achieve high sampling rate and high true positive rate with respect to detecting high-latency packets. Furthermore, good performance indicates that our measurement scheme can be applied to monitor SLA compliance and determine whether the percentage of high-latency packets within a time frame is acceptable.

9. Future Work

Because we designed our algorithm and data structure to be hardware friendly, future work would require implementing the prototype on a programmable hardware switch (e.g. using P4) and deploying the switch in a real-world system. Further studies might explore the computational time and analyze the overhead required to process packet headers, hash them, and perform lookups. Since the number of packets that pass through a network switch in a given second is so large, the computational speed of our algorithm and how it is implemented could greatly impact performance.

Future work could explore a smarter allocation of memory in the *d*-stage hash-indexed array. Currently, we split a memory allocation in to *d* equally sized partitions. However, the probability that a hash collision occurs in each of stages decreases as a packet moves through the pipeline. One might design a way to take advantage of this property to make more efficient use of memory.

If this measurement scheme is deployed to heavy network traffic, further studies could explore adaptive sampling that first detect problematic flows and change the sampling rate based on whether the flow is likely to experience performance issues. Perhaps an algorithm could adjust its sampling rate and learn based network traffic behavior.

References

[1] How many packets per second per port are needed to achieve wire-speed?, 2009. Accessed: 2018-05-06.

[2] Service level agreement for wholesale dedicated internet, Comcast, 2018. Accessed: 2018-05-06.

[3] Service level agreements, Verizon, 2018. Accessed: 2018-05-06.

[4] D. Ameller and X. Franch. Service level agreement monitor (salmon). In Composition-Based Software Systems, 2008. ICCBSS 2008. Seventh International Conference on, pages 224–227. IEEE, 2008.

[5] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In Proceedings of the 10th ACM SIGCOMM conference on Internet measurement, pages 267–280. ACM, 2010.

[6] N. Duffield. Sampling for passive internet measurement: A review. Statistical Science, pages 472–498, 2004.

[7] K. R. Fall and W. R. Stevens. TCP/IP illustrated, volume 1: The protocols. addison-Wesley, 2011.

[8] N. Feamster. COS461 lecture notes, 2017. Accessed: 2018-05-06.

[9] M. Ghasemi, T. Benson, and J. Rexford. Dapper: Data plane performance diagnosis of tcp. In Proceedings of the Symposium on SDN Research, pages 61–74. ACM, 2017.

[10] J. Goo, R. Kishore, H. R. Rao, and K. Nam. The role of service level agreements in relational management of information technology outsourcing: an empirical study. MIS quarterly, pages 119–145, 2009.

[11] A. Gupta, R. Birkner, M. Canini, N. Feamster, C. Mac-Stoker, and W. Willinger. Network monitoring as a streaming analytics problem. In Proceedings of the 15th ACM Workshop on Hot Topics in Networks, pages 106–112. ACM, 2016.

[12] A. Keller and H. Ludwig. The wsla framework: Specifying and monitoring service level agreements for web services. Journal of Network and Systems Management, 11(1):57–81, 2003.

[13] C. Kim, A. Sivaraman, N. Katta, A. Bas, A. Dixit, and L. J. Wobker. In-band network telemetry via programmable dataplanes. In ACM SIGCOMM Symposium on SDN Research (SOSR), 2015.

[14] D. D. Lamanna, J. Skene, and W. Emmerich. Slang: a language for service level agreements. IEEE Computer Society Press, 2003.

[15] V. Mohan, Y. J. Reddy, and K. Kalpana. Active and passive network measurements: a survey. International Journal of Computer Science and Information Technologies, 2(4):1372–1385, 2011.

[16] V. Paxson, J. Mahdavi, M. Mathis, and G. Almes. Framework for ip performance metrics. Framework, 1998.

[17] M. Peuhkuri. A method to compress and anonymize packet traces. In Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement, pages 257–261. ACM, 2001.

[18] O. Rana, M. Warnier, T. B. Quillinan, and F. Brazier. Monitoring and reputation mechanisms for service level agreements. In International Workshop on Grid Economics and Business Models, pages 125–139. Springer, 2008.

[19] M. Roughan. Fundamental bounds on the accuracy of network performance measurements. In ACM SIGMETRICS Performance Evaluation Review, volume 33, pages 253–264. ACM, 2005.

[20] E. Rye and R. Beverly. Sdn as active measurement infrastructure. arXiv preprint arXiv:1702.07946, 2017.

[21] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford. Heavy-hitter detection entirely in the data plane. In Proceedings of the Symposium on SDN Research, pages 164–176. ACM, 2017.

[22] C. Williamson. Internet traffic measurement. IEEE Internet Computing, 5(6):70–74, 2001.

[23] H. Xiao, B. Chan, Y. Zou, J. W. Benayon, B. O'Farrell, E. Litani, and J. Hawkins. A framework for verifying sla compliance in composed services. In Web Services, 2008. ICWS'08. IEEE International Conference on, pages 457–464. IEEE, 2008.

[24] L. Ye, H. Zhang, J. Shi, and X. Du. Verifying cloud service level agreement. In Global Communications Conference (GLOBECOM), 2012 IEEE, pages 777–782. IEEE, 2012.