Flow-Level Loss Detection with Δ-Sketches

Shir Landau Feibish
The Open University of Israel

Zaoxing Liu
Boston University

Nikita Ivkin
Amazon

Xiaoqi Chen
Princeton University

Vladimir Braverman
Rice University

Jennifer Rexford
Princeton University

ABSTRACT

Packet drops caused by congestion are a fundamental problem in network operation. Yet, it is difficult to detect *where* drops are happening, let alone *which flows* are most affected. Detecting the small-timescale drops caused by short bursts of traffic is even more challenging, and traditional monitoring techniques can easily miss them. To uncover packet drops as they occur inside a switch, the analysis must be real-time, fine-grained, and efficient. However, modern switches have distributed packet-processing pipelines that see either the arriving or departing traffic, but not the packet drops. Additionally, they do not have enough memory to store per-flow state. Our MIDST system addresses these challenges through a distributed compact data structure with lightweight coordination between ingress and egress pipelines. MIDST identifies the flows experiencing loss, as well as the bursty flows responsible, across different burst durations. Our evaluation with real-world traces and TCP connections shows that MIDST uses little memory (e.g., 320KB) while providing high accuracy (95% to 98%) under varying loss rates and burst durations. We evaluate a low-rate DDoS attack and demonstrate the potential use of our measurement results for attack detection and mitigation.

CCS CONCEPTS

• **Networks → Data path algorithms**.

KEYWORDS

Network Monitoring, Programmable devices, Sketches

ACM Reference Format:
Shir Landau Feibish, Zaoxing Liu, Nikita Ivkin, Xiaoqi Chen, Vladimir Braverman, and Jennifer Rexford. 2022. Flow-Level Loss Detection with Δ-Sketches. In *The ACM SIGCOMM Symposium on SDN Research (SOSR) (SOSR '22), October 19–20, 2022, Virtual Event, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3563647.3563653

SOSR '22, October 19–20, 2022, Virtual Event, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9892-3/22/10…$15.00
https://doi.org/10.1145/3563647.3563653

1 INTRODUCTION

Packet loss is one of the more puzzling problems that occur inside a network. Losses/drops can take place for any number of reasons, including congestion, hardware failures, software bugs, blackholes, and configuration mistakes, among others [22, 31]. Dropped packets can cause significant disruption to network operation, yet they are difficult to detect in a timely manner. Furthermore, even if a drop is eventually detected, it is hard to pinpoint exactly where and why the packet loss took place, let alone respond to the problem.

The losses caused by *microbursts* are especially difficult to detect, diagnose, and mitigate [17, 28, 33, 36]. Microbursts are bursts of traffic that are short-lived, yet they can cause massive congestion in switch queues, leading to immediate packet loss. Microbursts are often associated with data centers, yet such short bursts of traffic can occur in any network. For example, past measurement studies have shown microbursts in a campus network (due to active-monitoring tasks running concurrently in the network) [18] and a carrier network (due to customers simultaneously sending bursts) [3].

When bursts happen, timely and accurate measurements of both the flows responsible for the burst and those experiencing loss are critical. Based on this information, the network can target remedies to the responsible flows, *without* disadvantaging other flows. For example, the switch could mark, drop, rate-limit, or reroute the offending flows. In contrast, other countermeasures, such as congestion control and load balancing, typically affect *all* of the flows impacted by the congestion. Thus, an effective technique for measuring packet drops should satisfy three main goals:

- ***real-time*** (i.e., within a few microseconds) in detecting bursts to enable rapid mitigation;
- ***fine-grained*** in identifying the flows experiencing high loss and those responsible for congestion; and
- ***efficient*** in supporting analysis of many concurrent flows and varying loss rates in high-speed switch hardware.

Unfortunately, existing solutions compromise on one or more of these dimensions. Traditional NetFlow [8] and SNMP [4] only provide slow and coarse-grained loss statistics. Other approaches collect fine-grained statistics [2, 13–15, 22, 24, 26, 30, 38], but rely on *offline* analysis to detect packet loss or bursts "after the fact". More recently, NetSeer [37] supports real-time, in-switch loss detection through a "negative mirroring" feature that forwards dropped packets to an alternate egress pipeline for analysis. However, negative mirroring introduces high overhead under large bursts, essentially moving the heavy congestion elsewhere in the switch; some of the mirrored lost packets may themselves be lost. Plus, the results of the analysis are not readily available for taking action on the offending traffic as it streams by.

In this paper, we present **MIDST** (**M**onitoring **I**ntra-switch with **D**elta **S**ke**T**ch), which performs real-time measurement of the most lossy flows (the "heavy losers") and the heavy-hitters (the "bursty flows") within the data plane. Designing MIDST is challenging because modern high-speed switches are inherently *distributed*, with multiple ingress and egress pipelines that each have only a partial view of the traffic. In addition, each pipeline has *limited memory* for storing data structures; per-flow state is simply not an option. Plus, since detecting and reacting to packet drops must happen on a *small timescale*, the analysis cannot be offloaded to the control plane. We make several research contributions:

- **Estimating flow-level drops in modern switches (§2):** Dropped packets cannot be directly monitored in a distributed switch, as the ingress pipeline does not know they will be dropped and the egress pipeline does not see them at all. To overcome this limitation, MIDST has the ingress pipeline piggyback traffic counts so that the egress pipeline can compute per-flow loss statistics as packets that do *not* get lost depart. MIDST maintains a sketch of arriving packets and a sketch of departing packets. The *difference of these two sketches (called a Δ-sketch) is equivalent to a sketch of the difference*, thus the Δ-sketch measures the stream the lost packets, without even accessing them. To the best of our knowledge, Δ-sketch is the first data-plane sketch that can find the few "heavy loser" flows even if there are thousands of much heavier flows without loss.
- **Adapting measurement to burst duration (§3).** Traffic bursts vary in frequency and duration, from microbursts of a few milliseconds to heavy load over multiple seconds. Rather than impose an artificial time window on loss measurements, MIDST automatically tracks the traffic for the duration of the burst—initiating sketches as congestion builds and quickly resetting them as congestion wanes. Initiating and resetting the sketches in the data plane (via recirculated packets and subsequent traffic) is much faster (4ms) than relying on the control plane (730ms) to clean the data structures between consecutive burst events.

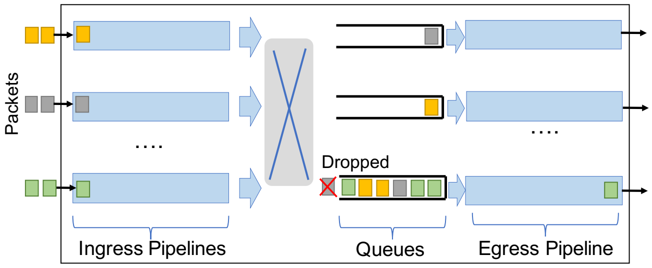

Figure 1: Packet drops within a multi-pipe switch.

- **Hardware implementation (§4):** We implement a MIDST prototype in a Tofino switch using the P4 language. Since Δ-sketches are essentially measuring the set of lost packets, each sketch instance can be small, enabling us to merge smaller counter arrays into larger ones to save data-plane resources (e.g., hashes and memory accesses). We perform a range of experiments on a real-world testbed of a 3.2Tbps switch and three commodity servers, shows that MIDST (1) offers high accuracy (95% to 98%) with small memory footprints (e.g., 320KB) under varying loss rates and burst durations, (2) accurately detects real-world lossy TCP and UDP flows during short congestion events, and (3) enables the detection and potential mitigation of low-rate DoS attacks by providing fine-grained heavy loser and bursty flow information.

2 COUNTING FLOW-LEVEL DROPS

MIDST identifies the flows experiencing the most packet loss (i.e., the *heavy losers*). Performing this analysis is challenging because high-speed switches process packets in a distributed fashion, as shown in Figure 1. Ingress pipelines see arriving packets, and egress pipelines see departing packets (after some queuing delay), but neither see the packets dropped due to congestion. So, MIDST computes drop statistics by *piggybacking* "counts" on packets traveling from ingress to egress (Section 2.1). The pipelines have too little memory to maintain per-flow state. So, MIDST maintains *approximate* counts in each pipeline and uses them to emulate a sketch of lost packets (Section 2.2).

2.1 Piggyback Counts on Delivered Packets

As a packet traverses the switch, MIDST estimates the loss of the packet's flow. If the loss is significant, the egress pipeline can take immediate action such as generating a report, marking the packet's Early Congestion Notification (ECN) bit, rerouting the packet, or relaying congestion feedback to the ingress port or upstream switch.

Monitoring packet drops is challenging because the loss occurs *inside* the switch, when packets encounter a congested queue. The packet-processing pipelines cannot "see"

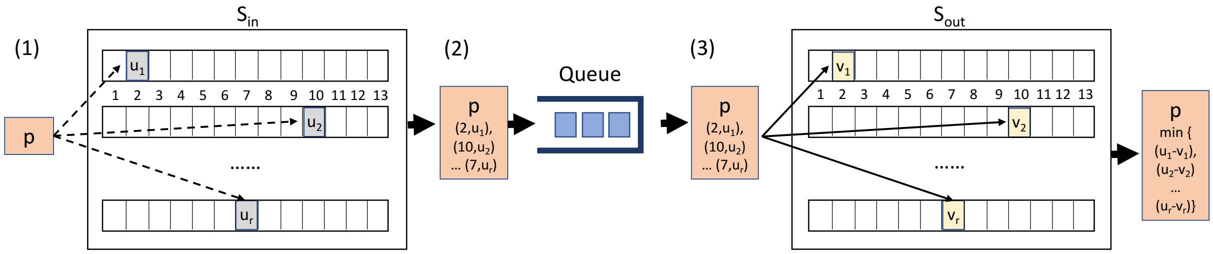

Figure 2: MIDST uses the difference between sketches to emulate the loss stream.

the dropping of the packets directly. Certain switches provide some visibility through a *negative mirroring* feature that directs a copy of dropped packets to a separate monitoring pipeline for analysis. However, negative mirroring is not efficient under high packet-loss rates, as it requires processing of every dropped packet, which may be infeasible in high loss rates. Plus, the loss statistics are collected in a separate pipeline, and are therefore not readily available to the egress pipeline serving that flow, thus hindering corrective action.

Instead, MIDST counts both the arriving packets (on ingress pipelines) and the departing packets (on egress pipelines), and computes the difference between the two to determine the number of lost packets. However, the ingress and egress counts are not synchronized, due to variable queuing delay. So, MIDST uses the packets themselves to carry and synchronize the counts between the ingress and egress pipelines. That is, the delivered (i.e., not dropped) packets allow MIDST to compute the statistics about the dropped packets of the flow to affect the scheduling or routing of future packets. In order to add the information on the packet, we would need to add an additional header to the packet. This header can be added in the ingress and removed in the egress so the packets exiting the switch remain unchanged.

The simplest approach would maintain a per-flow counter on the ingress and egress pipelines, and have each delivered packet carry the current ingress count to the egress. Then, the egress pipeline could compute the difference between the two counts to determine the number of lost packets.

2.2 Approximate Loss Using Δ-Sketches

Maintaining per-flow counters is too expensive, given the limited data-plane memory. Instead, each pipeline should maintain an approximate data structure, as shown in Figure 2. The ingress pipeline has a sketch S_{in} of arriving packets, and the egress pipeline has a sketch S_{out} of departing packets. For example, a count-min sketch [10] contains r rows, with c columns in each row, where each entry stores a count. When a packet arrives for flow f, the ingress pipeline updates S_{in} by computing r hashes on the flow identifier f to compute indices $a_1, ..., a_r$ and incrementing the associated counters to new values $u_1, ..., u_r$ before directing the packet to the appropriate queue. Similarly, in processing a packet from the head of the queue, the egress pipeline updates the sketch S_{out} of the number of departing packets with counts $v_1, ..., v_r$.

Subtracting sketches to find loss: Each pipeline could produce its own estimate of the number of arriving or departing packets for a flow, respectively, and combine the results, but that approach is not necessarily accurate. That is, the ingress pipeline could compute $u_{min} = \min\{u_1, ..., u_r\}$ and relay u_{min} to the egress for comparison with $v_{min} = \min\{v_1, ..., v_r\}$. However, both sketches may *over*estimate their counts due to hash collisions with other flows. Subtracting the two estimates could lead to large errors in estimating the number of lost packets. For example, suppose a packet from flow f observes the values $(u_1, u_2, u_3) = (25, 40, 30)$ in S_{in} and the values $(v_1, v_2, v_3) = (20, 10, 5)$ in S_{out}, respectively. The ingress pipeline would compute an estimate of $u_{min} = 25$ arriving packets, and the egress pipeline would compute an estimate of $v_{min} = 5$ departing packets, leading to an estimate of $u_{min} - v_{min} = 20$ dropped packets.

Instead, MIDST capitalizes on how sketches like the count-min sketch are *linear operators*, when the sketches have the same size (i.e., r and c) and the same random hash functions. In particular, MIDST can subtract one sketch from the other to compute the sketch S_{loss} of the stream of lost packets that neither pipeline can observe directly! That is, $S_{loss} = S_{in} - S_{out}$, where subtraction is the *matrix difference* of the sketches. We call such difference sketches a Δ-*sketch.*

Introducing Δ-sketches: More formally, let $A \subset B$ be two sets and let sketches S_A and S_B be the compressed summaries of A and B, respectively, of the same size. The matrix difference $\Delta = S_B - S_A$ is a sketch of $B - A$. Critically, we can compute the Δ-sketch even when we do not have access to the set of lost packets $B - A$.

Δ-sketches in MIDST: Returning to the example, the S_{loss} sketch would have entries $(u_1 - v_1, u_2 - v_2, u_3 - v_3)$, or $(5, 30, 25)$ for flow f. So our estimate would be 5, which is the minimum of these values, which is a better estimate of the number of lost packets than 20. In short, the "minimum of the differences" $(\min\{u_1 - v_1, u_2 - v_2, u_3 - v_3\})$ is a better estimate than the "difference of the minimums" $(u_{min} - v_{min})$. This example also shows how collisions may affect Δ-sketches. If for example, due to collisions u_{min} and v_{min} significantly overestimate the count of f, this over-estimation is likely to be cancelled out when computing the difference between the sketches. In this case, the accuracy of the Δ-sketch is influenced by the number of packets lost and not by the size of the flows.

MIDST does not need to materialize the S_{loss} sketch explicitly. Instead, the ingress pipeline simply tags each packet with the r associated counts (u_1, u_2, u_3), and the egress pipeline looks up (v_1, v_2, v_3) to compute $(u_1 - v_1, u_2 - v_2, u_3 - v_3)$ and the minimum of these three values. As an additional optimization, the ingress can tag the packet with the *indices* (a_1, a_2, a_3)—with values 2, 10, and 7 in the above example so the egress pipeline doesn't need to re-compute these hashes.

Finding the bursty flows. So far we have discussed how MIDST can identify packet losses. Surprisingly, MIDST can also identify the bursty flows, for free. Intuitively, the bursty flows are the flows which inserted many packets into the queue, meaning, the flows which had many packets at ingress. The sketch S_{in} maintains the aggregated measurement of the number of packets that entered the switch and should have been processed by the egress pipeline, so flows which are heavy hitters in S_{in} are, in fact, the bursty flows. Therefore we can identify if a packet is part of a bursty flow simply by computing its flow's count in S_{in}.

In-flow FIFO. Thus far we have assumed that each egress pipeline serves a single FIFO queue, meaning that each flow maintains a FIFO order of its packets for each ingress pipeline. In reality, each egress pipeline may have multiple FIFO queues. An ingress pipeline could map the same flow to different egress queues, however that may create out-of-order packets. In practice, packets of the same flow would normally be mapped to the same queue, and the relative order of packets within a flow is preserved on ingress and egress. This assumption can be violated if flow routing changes within a short time span. This can happen, for example, if there is a link failure causing the flow's route to be modified. In this case, MIDST may temporarily behave poorly as it may overestimate the loss. Once in-flow FIFO is restored, MIDST should quickly resume its regular behaviour.

3 REACTIVE MONITORING

The data structure in §2 estimates per-flow drop counts over a long time period. However, many congestion events are short-lived, requiring monitoring on a smaller timescale. The right time granularity depends on when the congestion event begins and ends, making it hard to adopt a fixed monitoring interval. Instead, MIDST triggers loss monitoring based on changes in the queue depth, and automatically "cleans" the data structures between congestion events.

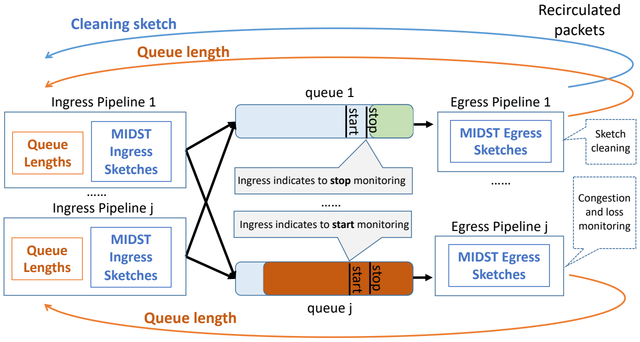

Figure 3: MIDST reactive monitoring.

3.1 Trigger on Changes in Queue Depth

Most network telemetry systems monitor traffic over long time intervals [22, 23, 29] or adopt fixed-length sliding windows [6]. However, congestion events can arise at any time, and last for any duration. As such, any fixed timescale for loss monitoring can introduce inaccuracy. Long time windows can obscure the effects of short-lived spikes in queue length caused by microbursts. Small time windows raise challenges in resetting the data structure between successive intervals, or alternating between multiple data structures. Given most network queues are underutilized most of the time, more efficient monitoring techniques should be possible.

Instead of resorting to fixed time intervals, MIDST introduces a novel *reactive monitoring* procedure. That is, MIDST identifies when congestion starts to build up and promptly starts monitoring. Reactive monitoring enables MIDST to adapt the monitoring to the duration of the burst, and provide information about an interval of time that tightly encapsulates the burst. So, MIDST only reports the "heavy losers" and "bursty flows" associated with the burst.

MIDST starts monitoring traffic destined to a queue when the queue depth crosses a high-water mark h, and stops monitoring when queue depth falls below a low-water mark l. The ingress and egress pipelines each have an important role to play in triggering a new monitoring interval. The ingress pipeline must initiate monitoring, to ensure that both pipelines maintain counts over the same time interval of packets; however, only the egress pipeline has direct access to the depth of the queue. For every fixed (small) interval, MIDST recirculates packets and piggybacks information on queue depth from the egress pipeline back to the ingress pipelines on the recirculated packets, as depicted in Figure 3.

If a queue exceeds length h, the ingress pipeline starts monitoring the traffic directed toward that queue. The ingress pipeline alerts the egress pipeline to start monitoring, by marking each packet destined to the associated queue. That is, any packet p "counted" in the ingress pipeline is marked so that the egress pipeline knows to "count" the packet as well.

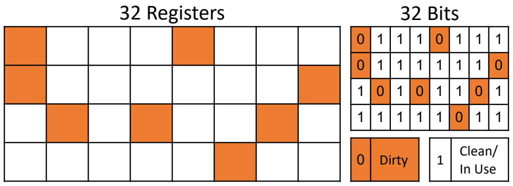

Figure 4: Auxiliary register for quick cleaning.

Upon learning that the queue depth has fallen below length l, the ingress pipeline stops monitoring by (un)marking packets accordingly.

3.2 Clean Sketches Between Monitoring

Once monitoring has stopped, all relevant sketches on ingress and egress need to be cleaned up. Yet, since the next microburst can arrive almost instantly, MIDST needs to clean up *quickly* to prepare for the next possible burst. One standard way to clear out and reset register arrays is to use the control-plane API (e.g., *counter write or reset* via Thrift [1]). However, as pointed out by prior work [25], accessing the data plane via control API presents a non-trivial delay. Instead, our implementation runs entirely in the data plane.

We use the packets being processed to clean the registers. Each packet that traverses through the switch is assigned a group of indexes to reset. To avoid a situation in which there are not enough packets going through the switch to reset all registers, MIDST also generates a packet which recirculates through the switch until the cleaning process has completed.

To expedite the cleaning process even further, we maintain an auxiliary register array that allows us to quickly "clear out" sketch memory. As seen in Figure 4, for each 32 register arrays, MIDST maintains a single 32-bit register. Each bit corresponds to one of the registers in the sketch, and indicates whether the corresponding register is clean or not. This array starts out with a 0 value in each bit. Once monitoring starts, when any register in the sketch needs to be updated for the first time in that monitoring interval, the corresponding bit is flipped to 1 and the register is reset before it is written to. Consequent packets writing to that register (during that interval) do not reset the bit or the register. As shown, "orange" registers are considered dirty and their corresponding bit is set to 0, whereas "white" registers are in use and their corresponding bit is set to 1. When monitoring stops, all 32 *bits* are reset to 0 by resetting a single 32-bit register to 0. This way MIDST can speed up the cleaning process by 32×.

We note that there is a certain trade-off between the overhead of sketch cleaning (i.e., the recirculation frequency of the cleaning packet) and how quickly we can treat the next burst. With this trade-off in mind, we set $l << h$, to ensure that, once cleaning has started, monitoring does not restart before the cleaning phase has completed.

Cleanup Mechanism	160KB	320KB	640KB	1280KB
Control Plane API	95.32ms	187.46ms	370.12ms	730.36ms
Recirculation only	4.53ms	9.06ms	18.13ms	36.47ms
10G traffic+recirculation	0.55ms	1.09ms	2.20ms	4.39ms
40G traffic+recirculation	0.13ms	0.27ms	0.54ms	1.07ms

Table 1: Comparing sketch cleaning delays.

4 PERFORMANCE EVALUATION

We evaluate MIDST under various algorithm parameters and hardware settings. Our major findings are:

- MIDST can accurately detect the flows with high intra-switch loss using small memory, under various loss rates.
- MIDST is able to estimate the loss with high accuracy in a real-world testbed. We evaluate a low-rate DDoS attack [20] and demonstrate the potential use of our measurement results for attack detection and mitigation.

Testbed: We have implemented a prototype of MIDST in P4 and conduct experiments in a real-world hardware testbed with one Intel Barefoot Tofino switch and three commodity servers. Each server is equipped with two Intel Xeon Silver 4110 CPUs and a Mellanox CX-4 Pro 2×100G NIC. Our testbed has three hosts as the data plane connected through the Tofino switch using 100Gbps links.

In particular, we run the following experiments:

Latency in reactive monitoring. We evaluate the latency in stopping the monitoring in MIDST and cleaning up the counters. Specifically, we measure the time it takes for a full sketch cleaning using different methods, as depicted in Table 1. Compared to counter cleaning using the control plane API (same as measured in [25]), our recirculation-assisted approach runs significantly faster and leverages the subsequent traffic to accelerate the cleaning process. When there is no current traffic, the recirculation still guarantees finishing the sketch cleaning in a few milliseconds.

LDoS Attacks. In low-rate denial of service (LDoS) attacks [34, 35], concentrated attack traffic is sent in short-lived pulses. That is, as opposed to volumetric distributed denial of service (DDoS) attacks which use a large volume of traffic to exhaust the network bandwidth or host resources, LDoS attacks target the flow adaptive mechanisms in the network (e.g. one representative LDoS attack known as the *shrew* attack [20] targets the TCP retransmission time out mechanism). Thus, in LDoS attacks, the average attack traffic volume is low in a large time window, therefore allowing such attacks to evade volumetric detection mechanisms.

Methodology: To demonstrate MIDST can be used effectively to analyze losses suffered by real-world flows, we set up a Shrew attack scenario on the hardware testbed:

- We use `iperf3` to generate a total of 100 TCP flows between a sender and a receiver as workload. The sender and receiver connect to the switch using 100Gbps links.

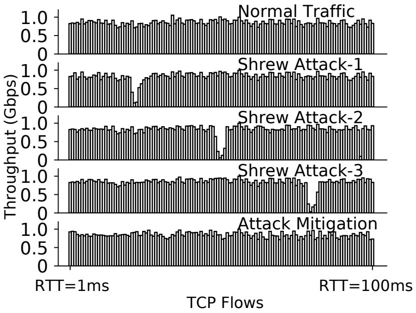

Figure 5: Shrew attack scenarios.

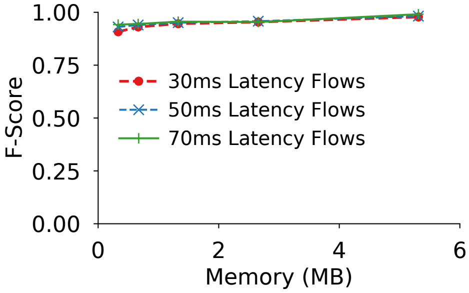

Figure 6: Detection in Shrew attacks.

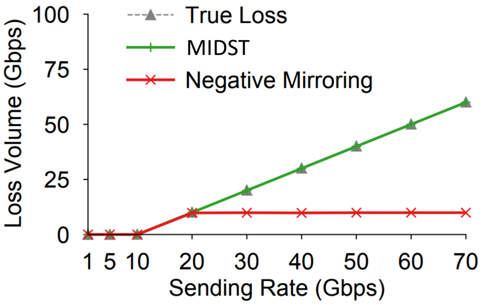

Figure 7: High-volume losses.

- Since the LDoS Shrew attack hurts the throughput of specific flows with a certain round-trip time (RTT), we use the `tc` utility to add varying latency to different flows using Linux kernel's queuing discipline. The added latency ranges from 1ms to 100ms for the 100 TCP flows created by `iperf3`. The testbed fabric only introduces 0.02ms latency between servers.
- We implement a custom script written in MoonGen [11] to generate periodic bursts from the attacker server, at an interval of 30, 50, 70ms, separately. Each burst contains about 2 Million UDP packets (64 bytes per packet) with a sending rate of 10Gbps.

Results: Under normal circumstances, the payload flows will run at the full 10Gbps line rate, and TCP congestion control guarantees minimal losses. When a shrew attack is launched, we can immediately observe the victim flow suffered from decreased throughput. For example, in Figure 5 (Shrew Attack-1), when we start the shrew attack with an interval of 30ms, only one TCP flow with that particular RTT suffered from significantly decreased throughput (dropped from 8.2Gbps to 0.97Gbps). We also observe that, due to the inherent measurement errors from injected latency, the TCP flows with RTTs around 30ms (e.g., 28ms to 32ms) are also affected with a decreased throughput.

We now show how network operators can potentially use MIDST to analyze packet loss and accurately identify the attacker and victim of an LDoS attack. Under normal conditions, the TCP flows in our testbed will lose no more than approximately 0.005% of packets due to congestion control. When we launch the attack with 30ms, 50ms, and 70ms interval (Attack-1 to 3 in Figure 5), the victim TCP flows start to lose as much as 98.1% of packets, while other flows (with varying RTTs) stay at a relatively stable throughput and we do not observe significant changes in their loss rates. MIDST accurately reports the victim TCP flow as the "heavy-loser", as shown in Figure 5. Out of ten repeated runs, MIDST identified LDoS victim flows correctly with no errors.

Meanwhile, we also correlate our heavy loser analysis with heavy flow analysis (which is a part of MIDST's ingress pipeline) to provide information about the culprit of the attack. By only looking at the ingress sketch S_{in}, we can find the heavy-hitter senders at the time when the victim flows are experiencing high losses and low volumes. This will help identify the potentially malicious sender that suddenly sends bursty traffic, and correlate it with the victim's high drop rate. Using this method, we identify the potential LDoS attacker flows with 0.912 to 0.981 F-scores under varying memory configurations, as shown in Figure 6. Note we define *F-score* as $\frac{FP}{FP+TN}$, where FP is the false positive rate (i.e., how many *non*-heavy loss flows are mistakenly identified as heavy lossy flows) and TN is the true negative rate (i.e., how many heavy loss flows were not identified).

Accuracy under high-volume drops. In this experiment, we generate a large number of dropped packets by sending an excessive volume of UDP traffic, leading to severe congestion. We show that the high volume of dropped packets overwhelmed the negative mirroring link, while MIDST still accurately tracks the amount of dropped packets. Specifically, an excessive amount of UDP traffic is sent to a bottleneck 10Gbps link. We use `pktgen-DPDK` to generate and send minimum-sized UDP packets to the switch via a 100Gbps link, varying the sending rate from 1Gbps to 70Gbps.

As shown in Figure 7, when the incoming traffic rate exceeds 10Gbps, the number of dropped packets starts to increase. The negative mirroring port is overwhelmed after the sending rate exceeds 20Gbps, and can only mirror a small fraction of all dropped packets. Meanwhile, MIDST accurately tracks the dropped packets. We note that the accuracy of MIDST only depends on the number of lossy flows tracked, and is not affected by the high volume of drops.

5 RELATED WORK

We summarize related work not covered in previous sections. **Network performance monitoring tools:** Existing solutions for network performance monitoring (e.g., Dapper [13], Marple [26], and Lean [24]) support detecting TCP/UDP packet loss by measuring the number of un-acked packets in the flows. Additionally, LossRadar [22] detects loss in a network by collecting flow counters of traffic as it enters and right before it leaves the network and analyzing these counters for differences. While MIDST can detect and analyze intra-switch packet loss, these tools can only detect the packet losses that did not happen within the switch (i.e., somewhere else along the flow path). Thus, the fundamental difference from our work lies in the new ability to efficiently measure the flow-level loss between the ingress and egress pipelines inside switches.

Sketch-based change detection: Intra-switch loss detection can be reformulated as a problem of change detection. Given two packet streams, change detection algorithms aim to find the flows whose sizes (e.g., the number of packets) have changed the most between two time intervals. Similarly, our goal is to identify the flow changes between the ingress and the egress as losses. While there are significant efforts in designing sketches for change detection [16, 19], we need to consider several subtle differences in the real-world setting, including hardware constraints and the limitations caused by the duration of the bursts.

Other efforts in switch measurement: In addition to loss detection, there are complementary telemetry capabilities that focus on switch-level measurement. For instance, ConQuest [6] can detect the heavy flows inside a queue when the queue is congested. While heavy flows in the queue can be the major contributor of the congestion, they may not be the victims that lose a significant portion of their packets. Instead, our work detects the victim flows with significant losses and provides further insights to understand the correlations between the heavy flows and heavy losers. Further, PacketScope [32] monitors the entire life-cycle of a packet in a switch and is capable of understanding the queuing delays and losses. However, PacketScope cannot aggregate packet-level information into the flow level and thus cannot report the lossy flows. Additionally, sketch-based algorithms can support others flow-level measurements, including heavy hitters [5, 10, 23, 29], flow size distribution [10, 21], entropy estimation [9, 27] and distinct flows [7, 12, 23].

6 CONCLUSION

MIDST detects and diagnoses congestion problems quickly and accurately, directly in the data plane of modern programmable switches. The design makes several novel contributions to compact data structures to monitor loss in a distributed setting, both within a high-speed switch and across the wide area. Through a combination of small sketches and lightweight synchronization protocols, MIDST can both (i) identify the “heavy losers" that neither the ingress nor egress pipelines can observe directly and (ii) monitor loss on the timescale of traffic bursts while quickly cleaning the data structures in between congestion events.

ACKNOWLEDGMENTS

We thank the anonymous SOSR reviewers and our shepherd Fernando Ramos for their valuable feedback. This work is supported in part by the Israel Science Foundation under grant No. 980/21, and NSF grants CNS-2106946, CNS-2107239, and CNS-1704077.

REFERENCES

[1] Apache Thrift. https://thrift.apache.org/.

[2] Paramvir Bahl, Ranveer Chandra, Albert G. Greenberg, Srikanth Kandula, David A. Maltz, and Ming Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *ACM SIGCOMM*, pages 13–24. ACM, 2007.

[3] Swapna Buccapatnam, Xiaoqi Chen, Ken Duell, Shir Landau Feibish, Kathleen Meier-Hellstern, Yaron Koral, Steven A. Monetti, Aswatnarayan Raghuram, Jennifer Rexford, Joe Stango, Simon T. Tse, John Tulko, and Tzuu-Yi Wang. Fine-grained P4 measurement toolkit for buffer sizing in carrier grade networks. *Workshop on Buffer Sizing*, 2019.

[4] J. D. Case, M. Fedor, M. L. Schoffstall, and J. Davin. RFC 1157: Simple network management protocol (SNMP), 1990.

[5] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. *ICALP*, 2002.

[6] Xiaoqi Chen, Shir Landau Feibish, Yaron Koral, Jennifer Rexford, Ori Rottenstreich, Steven A. Monetti, and Tzuu-Yi Wang. Fine-grained queue measurement in the data plane. In *ACM SIGCOMM Conference on Emerging Networking Experiments and Technologies*, pages 15–29. ACM, 2019.

[7] Xiaoqi Chen, Shir Landau-Feibish, Mark Braverman, and Jennifer Rexford. BeauCoup: Answering many network traffic queries, one memory update at a time. In *ACM SIGCOMM*, 2020.

[8] Benoit Claise. Cisco Systems NetFlow Services Export Version 9. *RFC 3954*, 2004.

[9] Peter Clifford and Ioana Cosma. A simple sketching algorithm for entropy estimation over streaming data. In *International Conference on Artificial Intelligence and Statistics*, 2013.

[10] Graham Cormode and Shan Muthukrishnan. An Improved Data Stream Summary: The Count-Min Sketch and Its Applications. *Journal of Algorithms*, 2005.

[11] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. MoonGen: A Scriptable High-Speed Packet Generator. In *ACM SIGCOMM Internet Measurement Conference*, Tokyo, Japan, October 2015.

[12] Philippe Flajolet, Eric Fusy, Olivier Gandouet, and Frederic Meunier. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. In *Conference on Analysis of Algorithms*, 2007.

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

[14] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, David A. Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *ACM SIGCOMM*, pages 139–152, 2015.

[15] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *USENIX Networked Systems Design and Implementation*, pages 71–85, 2014.

[16] Monika R Henzinger. Algorithmic challenges in web search engines. *Internet Mathematics*, 1(1):115–123, 2004.

[17] Raj Joshi, Ting Qu, Mun Choon Chan, Ben Leong, and Boon Thau Loo. BurstRadar: Practical real-time microburst monitoring for datacenter networks. In *Asia-Pacific Workshop on Systems (APSys)*, 2018.

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

[19] Balachander Krishnamurthy, Subhabrata Sen, Yin Zhang, and Yan Chen. Sketch-based change detection: methods, evaluation, and applications. In *ACM SIGCOMM Internet Measurement Conference*, pages 234–247, 2003.

[20] Aleksandar Kuzmanovic and Edward W. Knightly. Low-rate TCP-targeted denial of service attacks: The shrew vs. the mice and elephants. In *ACM SIGCOMM*, pages 75–86, 2003.

[21] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. FlowRadar: A better NetFlow for data centers. In *USENIX Networked Systems Design and Implementation*, pages 311–324, 2016.

[22] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. LossRadar: Fast detection of lost packets in data center networks. In *ACM SIGCOMM CoNEXT Conference*, pages 481–495. ACM, 2016.

[23] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One sketch to rule them all: Rethinking network flow monitoring with UnivMon. In *ACM SIGCOMM*, 2016.

[24] Zaoxing Liu, Samson Zhou, Ori Rottenstreich, Vladimir Braverman, and Jennifer Rexford. Memory-efficient performance monitoring on programmable switches with lean algorithms. In *Symposium on Algorithmic Principles of Computer Systems*, pages 31–44, January 2020.

[25] Hun Namkung, Daehyeok Kim, Zaoxing Liu, Vyas Sekar, and Peter Steenkiste. Telemetry retrieval inaccuracy in programmable switches: Analysis and recommendations. In *ACM Symposium on SDN Research*, 2021.

[26] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-directed hardware design for network performance monitoring. In *ACM SIGCOMM*, pages 85–98, 2017.

[27] George Nychis, Vyas Sekar, David G. Andersen, Hyong Kim, and Hui Zhang. An empirical evaluation of entropy-based traffic anomaly detection. In *ACM SIGCOMM Internet Measurement Conference*, 2008.

[28] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. Jupiter rising: A decade of Clos topologies and centralized control in Google's datacenter network. In *ACM SIGCOMM*, pages 183–197, 2015.

[29] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer Rexford. Heavy-hitter detection entirely in the data plane. In *ACM Symposium on SDN Research*, 2017.

[30] Joel Sommers, Paul Barford, Nick G. Duffield, and Amos Ron. Improving accuracy in end-to-end packet loss measurement. In *ACM SIGCOMM*, pages 157–168. ACM, 2005.

[31] Cheng Tan, Ze Jin, Chuanxiong Guo, Tianrong Zhang, Haitao Wu, Karl Deng, Dongming Bi, and Dong Xiang. NetBouncer: Active device and link failure localization in data center networks. In *USENIX Networked Systems Design and Implementation*, pages 599–614, 2019.

[32] Ross Teixeira, Rob Harrison, Arpit Gupta, and Jennifer Rexford. PacketScope: Monitoring the packet lifecycle inside a switch. In *ACM SIGCOMM Symposium on SDN Research*, pages 76–82, March 2020.

[33] Jackson Woodruff, Andrew W. Moore, and Noa Zilberman. Measuring burstiness in data center applications. In *Workshop on Buffer Sizing*, pages 5:1–5:6, 2019.

[34] Zhijun Wu, Xu Qing, Jingjie Wang, Meng Yue, and Liang Liu. Low-rate DDoS attack detection based on factorization machine in software defined network. *IEEE Access*, 8:17404–17418, 2020.

[35] Changwang Zhang, Zhiping Cai, Weifeng Chen, Xiapu Luo, and Jianping Yin. Flow level detection and filtering of low-rate DDoS. *Computer Networks*, 56(15):3417–3431, 2012.

[36] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. High-resolution measurement of data center microbursts. In *ACM SIGCOMM Internet Measurement Conference*, pages 78–85, 2017.

[37] Yu Zhou, Chen Sun, Hongqiang Harry Liu, Rui Miao, Shi Bai, Bo Li, Zhilong Zheng, Lingjun Zhu, Zhen Shen, Yongqing Xi, Pengcheng Zhang, Dennis Cai, Ming Zhang, and Mingwei Xu. Flow event telemetry on programmable data plane. In *ACM SIGCOMM*, pages 76–89. ACM, 2020.

[38] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert G. Greenberg, Guohan Lu, Ratul Mahajan, David A. Maltz, Lihua Yuan, Ming Zhang, Ben Y. Zhao, and Haitao Zheng. Packet-level telemetry in large datacenter networks. In *ACM SIGCOMM*, pages 479–491, 2015.