Network-Wide Heavy Hitter Detection with Commodity Switches

Rob Harrison, Qizhe Cai, Arpit Gupta, and Jennifer Rexford
Princeton University

ABSTRACT

Many network monitoring tasks identify subsets of traffic that stand out, *e.g.*, top-k flows for a particular statistic. A Protocol Independent Switch Architecture (PISA) switch can identify these "heavy hitter" flows directly in the data plane, by aggregating traffic statistics across packets and comparing against a threshold. However, network operators often want to identify interesting traffic on a *network-wide* basis. To bridge the gap between line-rate monitoring and network-wide visibility, we present a distributed heavy-hitter detection scheme for networks modeled as one-big switch. We use adaptive thresholds to perform efficient threshold monitoring directly in the data plane. We implement our system using the P4 language, and evaluate it using real-world packet traces. We demonstrate that our solution can accurately detect network-wide heavy hitters with up to 70% savings in communication overhead compared to an existing approach with a provable upper bound.

1 INTRODUCTION

Network operators often need to identify outliers in network traffic, to detect attacks or diagnose performance problems. A common way to detect unusual traffic is to perform "heavy hitter" detection that identifies the top-k flows (or flows exceeding a pre-determined threshold), according to some metric. For example, network operators often track destinations receiving traffic from a large number of distinct sources with high-precision in order to detect and mitigate DDoS attacks or TCP incast [4] in real time. In traditional networks, this heavy-hitter detection relies on analyzing packet samples or flow logs [5, 6]. Programmable switches open up new possibilities for aggregating traffic statistics and identifying large flows directly in the data plane [17, 18, 24, 27]. These solutions use approximate data structures, that bound memory and processing overhead in exchange for some loss in accuracy, in order to deal with the limited resources available on the switches.

SOSR '18, March 28–29, 2018, Los Angeles, CA, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5664-0/18/03...$15.00
https://doi.org/10.1145/3185467.3185476

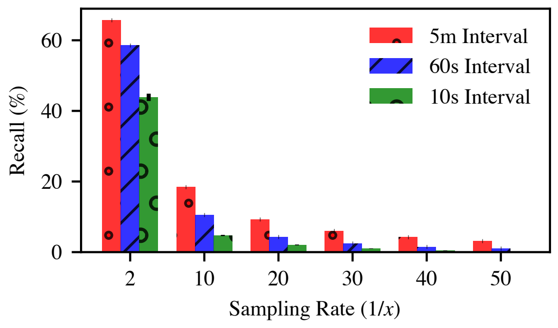

Figure 1: *This graph shows the recall for detecting heavy-hitters between two major ISPs [12] with different monitoring intervals. Even under high sampling rates, recall quickly diminishes and worsens as the monitoring interval decreases.*

While prior work has focused on heavy-hitter detection at a single switch, network operators often need to track the *network-wide* heavy hitters. For example, port scanners [15] and superspreaders [27] could go undetected if the traffic is monitored only at one location. Detecting the heavy hitters separately at each switch and then combining the results is not sufficient. Large flows can easily fall "under the radar" at multiple locations but still have sizable total volume. Applying a lower detection threshold at each switch reduces the chance of missing large flows, at the expense of higher communication overhead to report counts to a central coordinator.

Additionally, networks that forward high traffic volumes often resort to sampling $1/x$ packets, where x is operator-defined based on the needs of the specific network. However, sampling can result in substantially reduced accuracy on small time scales [21], even when traffic volumes are high. In Figure 1, we show the impact sampling has on accuracy while performing heavy-hitter detection on a link between two major ISPs [12] processing approximately 1 GBps of traffic. Even with high sampling rates, recall is quite low on short time intervals and it quickly diminishes as sampling rates decrease. In modern datacenter networks where switches commonly sample one packet out of 1,000–30,000 [25], we need new, network-wide techniques that are both efficient and accurate for real-time monitoring.

Detecting network-wide heavy hitters is an instance of the continuous distributed monitoring (CDM) problem [7]. In this model, each of n sites sees a stream of observations (packets), and each observation (packet) is observed at precisely one site. These sites work with a central coordinator to compute some (commutative and associative) function over the global set of observations. The objective is to minimize the communication cost between the observers and the coordinator, while continuously computing the function in real time. For network-wide heavy hitters, we want to determine which flows exceed the threshold (τ) for a given statistic of interest. Previous work proved an upper bound $O(n \log \tau/n)$ on the communications overhead between the n observers and a single coordinator [9, 10] which we will call the CMY (Cormode–Muthukrishnan–Yi) upper bound. This approach relies on setting per-site thresholds (*e.g.*, τ/n), and alerting the coordinator when these thresholds are violated. Other approaches that relax accuracy constraints or behave non-deterministically can also improve upon this upper bound [7, 26] but we will focus on the deterministic, error-free case.

Unfortunately, these theoretical results have not led to practical solutions for computing network-wide heavy hitters [7]. In real networks, the coordinator is capable of general-purpose computations, but the individual sites (switches) have a more restrictive computational model. For example, switches cannot iterate over arrays of values and typically support only simple arithmetic and bitwise operations. Also, the existing CDM work does not adapt to natural differences in the portions of the traffic that enter (or leave) the network at different locations. In practice, the traffic from a single source IP address typically enters the network at a limited number of sites, such as from a finite list of peering routers [23] or from fixed positions in a data center rack for east-west traffic. Similarly, the traffic to a unique destination IP address or prefix usually leaves the network at just a few locations. This spatial locality of network traffic provides opportunities to reduce the overhead of detecting heavy hitters if the coordinator can efficiently adapt monitoring thresholds to the actual volumes of traffic experienced.

In this paper, we propose a communication-efficient method for identifying network-wide heavy hitters using Protocol Independent Switch Architecture (PISA) switches [2, 3]. Inspired by prior work on distributed rate limiting [22], we apply adaptive thresholds to adjust to skews in the traffic volumes across different edge switches. Each switch identifies which traffic to report to the coordinator, using *different local thresholds for different monitored keys*. The coordinator

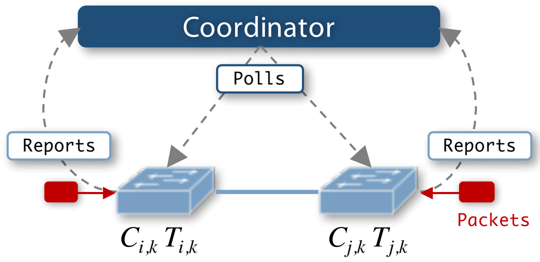

Figure 2: *CDM Dynamics. Each switch stores a count ($C_{i,k}$) and a threshold ($T_{i,k}$). Indices (i,k) refer to switch i and key k, respectively. Unless otherwise specified, k is the standard flow five-tuple.*

combines the reports across the switches to aggregate statistics and identifies the heavy hitters. Also, the coordinator *selectively polls* switches for additional counts and *updates the local thresholds* for relevant keys to reduce overhead. We prototype our solution using the P4 [2] language and compile to the Barefoot Tofino chipset [20]. Experiments with ISP backbone traces [12] show that our method substantially reduces the number of messages exchanged between the switches and centralized controller while maintaining 100% precision and recall.

2 NETWORK-WIDE HEAVY HITTERS

Rather than focusing on detecting heavy hitters with high memory efficiency, our approach focuses on reducing the overall *communication overhead* between the switches and the coordinator. Our network-wide algorithm counts the traffic entering the network at each edge switch, and applies local, per-key thresholds to trigger reports to a central coordinator. The coordinator adapts these thresholds to the prevailing traffic to reduce the total number of reports.

2.1 Distributed, Adaptive Thresholds

Edge switches count incoming traffic across packets with the same key k, such as a source IP address, source-destination pair, or five-tuple. Each edge switch i maintains a count ($C_{i,k}$) and a threshold ($T_{i,k}$) for each key k, as shown in Figure 2. The switch computes the counts, and the central coordinator sets the thresholds. When the local count for a key reaches or exceeds its local threshold, the switch sends the coordinator a report with the key and the count, which triggers the controller to `HandleReport`$(i, C_{i,k})$ as shown in Algorithm 1.

The coordinator combines the counts for the same key across reports from multiple switches. Since switches only send reports after the count for a key equals or exceeds its local threshold, the coordinator has incomplete information about the true global count. A switch i that has *not* sent a report for key k could have a count, at most, just under

$T_{i,k}$, which allows the coordinator to make a conservative estimate of the global count. The controller computes this estimate (`Estimate(`k`)`) by aggregating the counts ($C_{i,k}$) from switches that sent reports and assuming a count equal to one less than the *local threshold*, *i.e.*, $C_{i,k} = T_{i,k} - 1$, for the non-reporting switches. If the estimated total equals or exceeds the global threshold for the key ($T_G(k)$), the coordinator *polls* all of the switches whose local thresholds are greater than zero to learn their current counts and produce a more accurate estimate. If the total calculated after polling equals or exceeds the global threshold, then the coordinator reports key k as a heavy hitter with the count $\sum_i C_{i,k}$.

The coordinator adapts the per-key local thresholds based on past reports. Each local threshold starts as a fraction of the global threshold and the number of sites, *i.e.*, $T_{i,k} = T_G(k)/n$, and is then recomputed by the coordinator based on subsequent reports. Algorithm 1 describes the actions taken by the coordinator after receiving a `Report(`$i, C_{i,k}$`)`. Inspired by distributed rate limiting [22], the coordinator adapts the local thresholds based on the exponentially weighted moving average (EWMA) of the local and global counts. We use the EWMA to reflect the intuition that if a particular key was a heavy-hitter in the past, it is likely to be a heavy hitter in the future. We, therefore, adjust local thresholds (`Reset_Threshold(`k`)`) to reflect each site's fraction of the global EWMA for a particular key. This adjustment ensures that switches which observe the majority of the traffic for a given key apply a higher local threshold. By tuning these local thresholds based on the local and global EWMA, we further reduce the communication overhead between the switches and the coordinator.

Algorithm 1: Adaptive Local Thresholds: Controller

```
Input: N switches, Global Threshold $T_G(k)$, Count $C_{i,k}$,
Output: Heavy Hitter Set (H), Local Threshold $T_{i,k}$
Func HandleReport($i, C_{i,k}$):
    $ReportedC_{i,k} \leftarrow C_{i,k}$
    if Estimate ($k$) $\geq T_G(k)$
        if GlobalPoll($k$) $\geq T_G(k)$
            $H \leftarrow H \cup \{k\}$
        Reset_Threshold ($k$)

Func Estimate($k$):
    return
    $\sum_{i=1}^{N} (ReportedC_{i,k} \geq T_{i,k} \,?\, ReportedC_{i,k} : T_{i,k} - 1)$

Func Reset_Threshold($k$):
    foreach $i \in N$ do
        $frac \leftarrow \frac{(1-\alpha) \times EWMA_{i,k} + \alpha \times ReportedC_{i,k}}{\sum_{j=1}^{N} (1-\alpha) \times EWMA_{j,k} + \alpha \times ReportedC_{j,k}}$
        $T_{i,k} \leftarrow frac \times (T_G(k) - \sum_{j=1}^{N} ReportedC_{j,k}) + ReportedC_{i,k}$

Func GlobalPoll($k$):
    $Total \leftarrow 0$
    foreach $i \in N$ do
        if $T_{i,k} > 0$
            $Total \leftarrow Total$ + Poll ($i, k$)
    return $Total$
```

2.2 Implementation

Switches can maintain the per-key state (local counts and thresholds) using a hash table. In the simplest case, the switch would keep per-key state in registers, storing the current count $C_{i,k}$ and threshold $T_{i,k}$ for each key k. Upon receiving a packet, the data plane hashes the key to identify the corresponding register entry and updates the associated count (*e.g.*, a count of bytes or packets). If the local count equals or exceeds the local threshold, the switch generates a report to the coordinator.

Prototype. Our prototype of the data plane algorithm consists of approximately 200 lines of P4 code to monitor per-key counts with adaptive thresholds. We allocate two registers (hash tables) to store the count and the threshold for each key. The maximum number of entries that can be stored in registers across all stages of a particular PISA target determines the maximum number of keys that can be monitored in the data plane. When a packet arrives, match-action tables determine if it corresponds to a monitored key and, if so, looks up the current count and threshold in each register. Alternatively, the threshold could also be stored as a parameter to a match-action table entry. Depending on the cost of updating match-action table entries or the availability of register memory in different data plane targets, one could choose whichever implementation is appropriate for that target and forwarding logic. The switch generates a report for the coordinator by cloning the original packet that triggered the report and embedding the count into the clone. All of this logic can be performed in as few as seven logical match-action tables using P4 and such a small program could easily run alongside sophisticated forwarding logic with the resources available on targets like Tofino [20]. However, the precise amount of memory used on the switch depends on the aggregation level of the monitored keys (*e.g.*, five-tuple flow or single IP), the timescale of monitoring, and the distribution of keys in the underlying network traffic.

2.3 Memory Efficiency Considerations

While maintaining per-key state is expensive from a memory perspective, Section 3 shows that, in practice, we can store

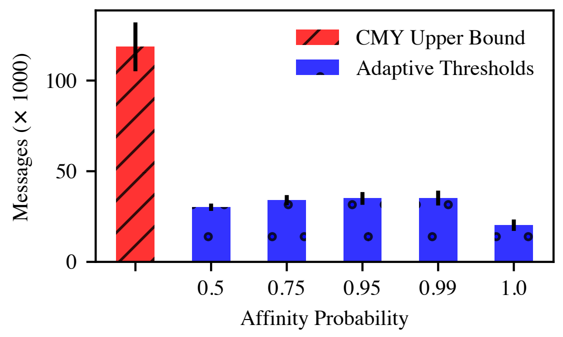

Figure 3: *Communication overhead is reduced even when the sources' affinity for a preferred ingress switch is low.*

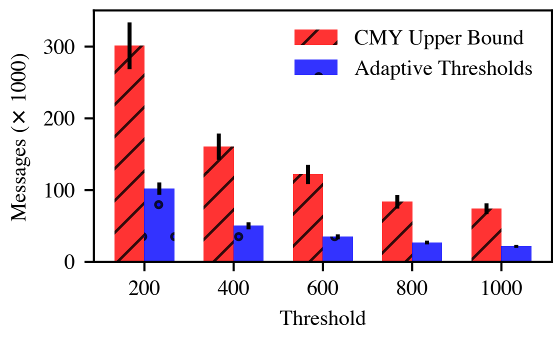

Figure 4: *As the global threshold increases, the fraction of overall traffic that constitute heavy hitters decreases, reducing the communication overhead.*

per-key state for a realistic query, based on real-world traffic traces. Our system identifies heavy hitters on a rolling time window (W), at the conclusion of which the counters for each key are reset. In our experiments, we choose a value of W that keeps the number of keys per window manageable. However, nothing about our approach prevents us from employing space-saving algorithms and data structures [8, 24] if the memory constraints were prohibitive or we wanted to use much longer time windows. In Section 4, we will discuss how compact data structures can actually enhance our system beyond the base algorithm.

3 EVALUATION

In this section, we quantify the reduction in the communication overhead using our algorithm. We first describe the experimental setup in Section 3.1 and then quantify the performance of our solution using the CMY bound described by Cormode et al. [7, 9, 10] as the baseline. We then quantify how sensitive our results are to various experimental parameters in Section 3.2. Our evaluation shows that our algorithm improves upon the CMY upper bound for the countdown problem [7] by up to 70%. We omit an explicit experiment to evaluate accuracy because both algorithms achieve 100% precision and recall.

3.1 Experimental Setup

To quantify the performance of our approach, we used CAIDA's anonymized Internet traces from 2016 [12]. These traces consist of all the traffic traversing a single OC-192 link between Seattle and Chicago within a major ISP's backbone network. Each minute of the trace consists of approximately 64 million packets. For our experiments, we consider all IPv4 packets for analysis using a rolling time window of $W = 6$ seconds. On average, 6 million packets are processed in each window which consists of approximately 300,000 flows and 250,0000 unique source and destination pairs. For calculating EWMA, we used a smoothing factor, $\alpha = 0.8$ for all our experiments. This factor must satisfy $0 < \alpha < 1$ where smaller values of α react to changes in the average slower than larger values do.

We simulate a one-big-switch network consisting of n edge nodes (switches). In order to model spatial locality of network traffic using data from a point-to-point link, we associate packets from the trace with a given ingress switch based on a hash of the source IP address. For each source IP address, we assign an affinity for a specific ingress switch with probability p. Packets from a given source IP are, therefore, processed at a "preferred" switch with probability p and at l other switches with probability $(1-p)/(l-1)$ where $n, l \geq 2$. On this distribution of traffic, we run a simple heavy hitter query to determine which flows (based on the standard five-tuple of source/destination IP address, source/destination port, and transport protocol) send a number of packets greater than a global threshold (T_G) during a rolling time window (W). Unless otherwise specified, we set $n = 4$, $l = 2$, $p = 0.95$, and $T_G = 600$ for our experiments.

Thresholds	200	400	600	800	1000
Message Reduction (%)	66.1	68.7	71.3	67.7	70.8

Table 1: *Communication reduction over the CMY upper bound is not affected as the threshold increases.*

3.2 Communication Overhead

We now use these traces to demonstrate how our approach reduces the communication overhead on continuous distributed monitoring for detecting heavy hitters. We compare the performance of our algorithm to an implementation based on the countdown problem for threshold monitoring [7]. This monitoring approach proceeds in a fixed sequence of rounds where each round reduces the per-site

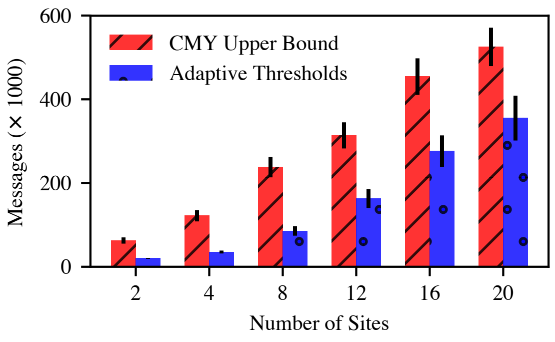

Figure 5: *As the number of sites increases, the communication overhead increases due to the additional nodes communicating with the coordinator.*

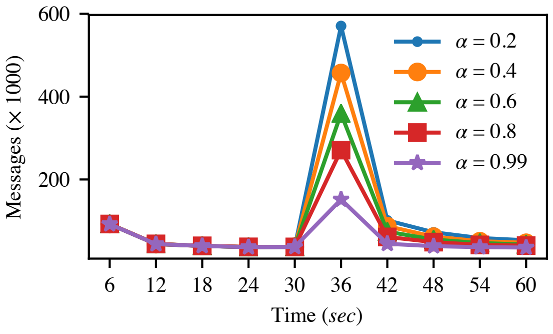

Figure 6: *As alpha increases, the algorithm is able to more quickly adapt to changes in the traffic distribution which results in lower communication overhead.*

Number of Sites	2	4	8	12	16	20
Message Reduction(%)	70.0	71.1	64.0	48.0	39.2	32.4

Table 2: *Communication reduction over the CMY upper bound lessens as the number of sites increases.*

threshold exponentially until it reaches 1. We quantify the communication overhead in terms of median number of messages per window interval. To demonstrate the sensitivity of the communication reduction with respect to various parameters, we ran experiments varying one of four key parameters: affinity probability (p), global threshold (T_G), number of sites (n), and smoothing factor (α); for each experiment.

3.2.1 Sensitivity to Site Affinity. In this experiment, we compare the performance of our algorithm to the CMY upper bound for the countdown problem while varying the affinity for a given ingress switch. Figure 3 shows how the performance, quantified as the number of messages sent over time, varies as we increase the site affinity probability from $p = \{0.5, 0.75, 0.95, 0.99, 1.0\}$. Here, an affinity probability of $p = 0.5$ implies that a packet will be processed by the preferred site with probability 0.5 and $p = 1.0$ implies that a packet will only be processed at the preferred site. We see that our approach substantially reduces the number of messages exchanged between the sites and the controller regardless of the sources' affinity for a particular ingress switch. We do see a substantial drop between $p = 0.99$ to $p = 1.0$ because when $p = 1.0$ a given source *always* enters the network at the same location. The problem of determining network-wide heavy hitters has been reduced to determining which keys are heavy on each edge switch, which substantially reduces communication overhead.

3.2.2 Sensitivity to Threshold. In this experiment, we compare the performance of our algorithm for different threshold (T_G) values with the CMY upper bound for the countdown problem. Figure 4 shows how the performance, quantified as the total number of messages sent over the entire experiment duration (60 *sec*), varies as we increase the global threshold. The total number of messages decreases as the threshold value increases because the total number of heavy hitters necessarily decreases with the larger thresholds. However, the increase in threshold has little impact on the performance of our algorithm compared to the CMY bound. Table 1 shows that our solution incurs 70% less communication overhead for $T_G = 1000$, which corresponds to a top-k=700 for this data set and query.

3.2.3 Sensitivity to Number of Sites. In this experiment, we compare the performance of our algorithm with the CMY upper bound for the countdown problem as the number of sites (n) increases. Figure 5 shows how the performance, quantified as the total number of messages sent over the entire experiment duration, varies as we increase the number of sites. We observe that as the number of sites increases, our communication overhead increases as a result of the increased cost to globally poll all sites. However, Table 2 shows that our solution still reduces the communication overhead by 70–30% for up to $n = 20$.

3.2.4 Sensitivity to Traffic Changes. In real-world networks, changes to traffic patterns and distributions are a regular occurrence due to planned changes as well as failures. In Figure 6, we examine how well our algorithm responds to changes in traffic patterns. At time $t = 30s$, we change the set of ingress switches for all keys and evaluate how our algorithm performs for various choices of the smoothing factor (α). For all values of α, we see a sharp increase in the communication overhead after a disruption followed by a brief period of adjustment, depending on the value of α. The single, abrupt change in this experiment fails to account for the variety of traffic dynamics one might experience in a production network and selecting the best value of α depends on those specific conditions. For example, selecting a large α might perform worse in a network that experiences frequent, but brief, transient changes because it would frequently "over-correct" the thresholds for these brief changes.

4 FUTURE WORK

While our evaluation demonstrated that our algorithm substantially reduces the communication overhead for detecting heavy hitters, our approach can be improved in at least two ways: (1) by reducing the amount of state switches must store in the data plane, and (2) supporting distinct counts.

Memory-Efficient Heavy-Hitters Storing per-key state to support adaptive thresholds has high memory overhead, so using a compact data structure, like a sketch, would be more memory-efficient. To reduce the space requirements, the data plane could maintain a count-min sketch [8] to estimate the counts for *all* keys, and then only store counts and thresholds for keys with counts above some minimum size that would qualify them as a potential heavy hitter.

Heavy Distinct Counts For simplicity of presentation, we have described count-based heavy hitters for the majority of this paper. However, adaptive, per-key thresholds can be used with any heavy-hitter statistic when the function applied is both commutative and associative, *e.g.*, count, sum, max, *etc.* However, these techniques are not effective when computing *distinct* counts, such as the number of unique sources contacting a given destination. Fortunately, we can again leverage an approximate algorithm known as HyperLogLog [11] to solve this problem. This algorithm takes advantage of randomization in order to approximate distinct counts in a distributed fashion that can be merged by a central coordinator.

5 RELATED WORK

Our work lies at the intersection of several areas in the database, theory, and the networking research communities.

Frequent and Top-*k* Item Detection Calculating frequent and top-k items over data streams has been well-studied. However, much of this work has focused on theoretical bounds and reducing the space [8] required to calculate these statistics. Several systems [13, 17, 18, 24, 27] make use of these compact data structures to perform heavy hitter detection on a single switch. Our work is orthogonal to these approaches that reduce the memory overhead on a single device; instead, we focus on reducing the communication overhead required to perform network-wide heavy hitter detection.

Distributed Detection Jain *et al.* [14] make the case for using local thresholds to monitor a global property, but they focus on the design considerations for such solutions rather than a specific system. Our work demonstrates an actual prototype that uses adaptive, local thresholds inspired by distributed rate limiting [22] and calculated with local and global estimates. The problem of calculating frequent and top-k items over distributed data streams has also been well-studied. These works shift their focus from reducing memory overhead to reducing the communication overhead in the distributed context [1, 9, 10, 16]. However, these approaches ignore the impact of key distribution in the distributed streams. Our work focuses on exploiting the spatial locality of network traffic to improve upon these previous results.

6 CONCLUSION

Detecting heavy-hitters is an indispensable tool in managing and defending modern networks. We designed an efficient algorithm and implemented a prototype for detecting network-wide heavy-hitters with commodity switches. Our evaluation with real-world traffic traces demonstrates that by dynamically adapting per-key thresholds, we can reduce the communication overhead required to detect network-wide heavy hitters without compromising accuracy.

As richer network traces become available from multi-switch networks, we can further explore the efficacy of this method for detecting network-wide heavy-hitters. Simulating any multi-switch traffic dynamics with the available data would have been inherently synthetic. With additional data, we could better explore how the reactiveness of the EWMA to short-term fluctuations affects the overall communications overhead. We could also improve our approach by starting with local thresholds learned from historical training data, given the availability of such data.

Detecting network-wide heavy hitters in networks modeled by one big switch is also one component of a more general network telemetry system. Recent work combines the flexible processing of PISA switches and stream processors to perform query-based network telemetry, but only on a single switch [13]. We foresee adapting our work to detect network-wide heavy hitters along network paths [19] as well as on one big switch for inclusion in such a general network telemetry system.

ACKNOWLEDGMENTS

We would like to thank our anonymous reviewers and our shepherd, Petr Lapukhov, for their thoughtful feedback. We would also like to thank Shir Landau-Feibish and Mina Tahmasbi Arashloo for their helpful suggestions and discussions.

REFERENCES

[1] Brian Babcock and Chris Olston. 2003. Distributed Top-k Monitoring. In *ACM SIGMOD International Conference on Management of Data*. ACM, 28–39.

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

[3] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. In *ACM SIGCOMM*. ACM, New York, NY, USA, 99–110.

[4] Yanpei Chen, Rean Griffith, Junda Liu, Randy H. Katz, and Anthony D. Joseph. 2009. Understanding TCP Incast Throughput Collapse in Datacenter Networks. In *ACM SIGCOMM Workshop on Research on Enterprise Networking*. ACM, New York, NY, USA, 73–82.

[5] Benoit Claise. 2004. *Cisco Systems NetFlow Services Export Version 9*. RFC 3954. RFC Editor. http://www.rfc-editor.org/rfc/rfc3954.txt

[6] Benoit Claise. 2008. *Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Traffic Flow Information*. RFC 5101. RFC Editor. http://www.rfc-editor.org/rfc/rfc5101.txt

[7] Graham Cormode. 2011. Continuous Distributed Monitoring: A Short Survey. In *International Workshop on Algorithms and Models for Distributed Event Processing*. ACM, 1–10.

[8] Graham Cormode and Shan Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms* 55, 1 (2005), 58–75.

[9] Graham Cormode, S Muthukrishnan, and Ke Yi. 2011. Algorithms for distributed functional monitoring. *ACM Transactions on Algorithms (TALG)* 7, 2 (2011), 21.

[10] Graham Cormode, S Muthukrishnan, Ke Yi, and Qin Zhang. 2010. Optimal Sampling From Distributed Streams. In *ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. ACM, 77–86.

[11] Philippe Flajolet and G Nigel Martin. 1985. Probabilistic counting algorithms for data base applications. *J. Comput. System Sci.* 31, 2 (1985), 182–209.

[12] Center for Applied Internet Data Analysis. 2018 (accessed November 1, 2017). *The CAIDA UCSD Anonymized Internet Traces 2016*. http://www.caida.org/data/passive/passive_2016_dataset.xml

[13] Arpit Gupta, Rob Harrison, Ankita Pawar, Rüdiger Birkner, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. 2017. Sonata: Query-Driven Network Telemetry. *arXiv preprint arXiv:1705.01049* (2017).

[14] Ankur Jain, Joseph M Hellerstein, Sylvia Ratnasamy, and David Wetherall. 2004. A Wakeup Call for Internet Monitoring Systems: The Case for Distributed Triggers. In *HotNets-III*.

[15] Jaeyeon Jung, Vern Paxson, Arthur W Berger, and Hari Balakrishnan. 2004. Fast portscan detection using sequential hypothesis testing. In *IEEE Symposium on Security and Privacy*. IEEE, 211–225.

[16] Ram Keralapura, Graham Cormode, and Jeyashankher Ramamirtham. 2006. Communication-efficient distributed monitoring of thresholded counts. In *ACM SIGMOD International Conference on Management of Data*. ACM, 289–300.

[17] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. 2016. FlowRadar: A Better NetFlow for Data Centers. In *Usenix NSDI*. 311–324.

[18] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. 2016. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon. In *ACM SIGCOMM*. ACM, 101–114.

[19] Srinivas Narayana, Mina Tahmasbi, Jennifer Rexford, and David Walker. 2016. Compiling Path Queries. In *Usenix NSDI*. 207–222.

[20] Barefoot Networks. 2018 (accessed November 1, 2017). *Barefoot Tofino*. https://www.barefootnetworks.com/products/brief-tofino/

[21] Peter Phaal and Sonia Panchen. 2003 (accessed February 14, 2018). *Packet Sampling Basics*. http://www.sflow.org/packetSamplingBasics/index.htm

[22] Barath Raghavan, Kashi Vishwanath, Sriram Ramabhadran, Kenneth Yocum, and Alex C. Snoeren. 2007. Cloud Control with Distributed Rate Limiting. In *ACM SIGCOMM*. ACM, New York, NY, USA, 337–348.

[23] Brandon Schlinker, Hyojeong Kim, Timothy Cui, Ethan Katz-Bassett, Harsha V Madhyastha, Italo Cunha, James Quinn, Saif Hasan, Petr Lapukhov, and Hongyi Zeng. 2017. Engineering Egress with Edge Fabric: Steering Oceans of Content to the World. In *ACM SIGCOMM*. ACM, 418–431.

[24] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S Muthukrishnan, and Jennifer Rexford. 2017. Heavy-Hitter Detection Entirely in the Data Plane. In *ACM SOSR*. ACM, 164–176.

[25] Cisco Systems. 2017 (accessed February 25, 2018). *Cisco Nexus 3600 NX-OS System Management Configuration Guide*. https://www.cisco.com/c/en/us/td/docs/switches/datacenter/nexus3000/sw/system_mgmt/503_U4_1/b_3k_System_Mgmt_Config_503_u4_1/b_3k_System_Mgmt_Config_503_u4_1_chapter_010010.pdf

[26] Ke Yi and Qin Zhang. 2013. Optimal tracking of distributed heavy hitters and quantiles. *Algorithmica* 65, 1 (2013), 206–223.

[27] Minlan Yu, Lavanya Jose, and Rui Miao. 2013. Software Defined Traffic Measurement with OpenSketch. In *Usenix NSDI*, Vol. 13. 29–42.