Network-Wide Heavy Hitter Detection with Commodity Switches

Rob Harrison
Princeton University

Qizhe Cai
Princeton University

Arpit Gupta
Princeton University

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 and approximate data structures to perform threshold monitoring and distinct counting directly in the data plane. We implement our system using the P4 language and Barefoot's Tofino hardware switch, and evaluate it using real-world packet traces. We demonstrate that our solution can accurately detect network-wide statistics with up to 75% savings in communication overhead.

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 to identify the top-k flows (or flows exceeding a pre-determined threshold), according to some metric. For example, network operators may want to know the top source-destination pairs by traffic volume, or the destinations receiving traffic from a large number of distinct sources. In traditional networks, heavy-hitter detection relies on analyzing packet samples or flow logs [4, 5]. Programmable switches open up new possibilities for aggregating traffic statistics and identifying large flows directly in the data plane [17, 18, 22, 26]. Since switches have limited resources, these techniques often involve approximate data structures, such as sketches, that bound memory, and processing overhead in exchange for some loss in accuracy.

These existing works focus on heavy-hitter detection at a single switch. However, network operators often need to track the *network-wide* heavy hitters. For example, port scanners [15] and superspreaders [26] could go undetected if the traffic is monitored at a single location. Detecting the heavy hitters separately at each switch, and combining the results, is not sufficient. Large flows can easily fall "under the radar" at multiple locations but still, have sizeable 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). Instead, we need new, network-wide techniques that are both efficient and accurate.

Detecting network-wide heavy hitters is an example of the continuous distributed monitoring (CDM) problem [6]. 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 [8, 9] which we will call the CMY upper bound. These approaches rely 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 [6, 25] 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 [6]. In our setting, 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,

SOSR'18, March 2018, Los Angeles, CA USA
2018. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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. Similarly, the traffic to a unique destination IP address or prefix usually leaves the network at just a few locations. The spatial locality of the traffic provides opportunities to reduce the overhead of detecting heavy hitters if the coordinator can efficiently adapt to the actual volumes of traffic.

In this paper, we propose a communication-efficient way to identify network-wide heavy hitters using Protocol Independent Switch Architecture (PISA) switches [2, 3]. Inspired by prior work on distributed rate limiting [20], we apply adaptive thresholds to adapt 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 combines the reports across the switches to aggregate statistics and identifies the heavy hitters. Also, the coordinator *selectively polls* switches for additional counts (to increase accuracy) and *updates the local thresholds* for the relevant keys (to reduce overhead). We prototype our solution using the P4 [2] language, for both the heavy-hitter and the heavy-distinct problems and compile to the Barefoot Tofino chipset [23]. Experiments with ISP backbone traces [24] show that adaptive thresholding achieves a substantial reduction in the communication overhead quantified as the number of messages exchanged between the switches and the centralized controller.

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

The edge switches count incoming traffic across packets with the same key k, such as the source IP address, source-destination pair, or five-tuple. Each edge switch i has a count ($C_{i,k}$) and a threshold ($T_{i,k}$) for each key k, as shown in Figure 1. The switch computes the counts, and the central coordinator sets the thresholds. When the local count for a key 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.

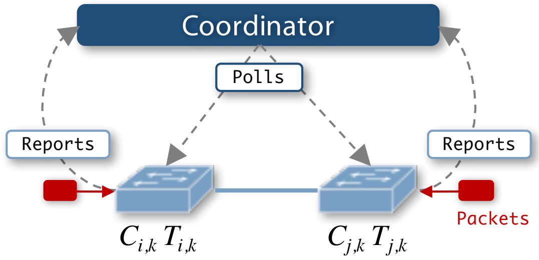

Figure 1: *CDM Dynamics. Each switch stores a count ($C_{i,k}$) and a threshold ($T_{i,k}$). Indices (i,k) refer to switch i and key k, respectively. Unless otherwise specified, k is the flow five-tuple.*

The coordinator combines the counts for the same key across reports from multiple switches. Since switches only send reports upon exceeding their local thresholds, the coordinator has incomplete information. A switch i that has *not* sent a report for key k could have a count just under $T_{i,k}$, allowing the coordinator to make a conservative estimate (Estimation(k)) of the total traffic by aggregating the *counts* $C_{i,k}$ for switches i that sent reports and the *thresholds* $T_{i,k}$ for the remaining switches. If the estimated total exceeds the global threshold T_G, the coordinator *polls* all of the switches to learn their current counts, to produce a more accurate estimate. If the total exceeds the global threshold,

```
Algorithm 1: Adaptive Local Thresholds: Controller
Input: $N$ switches, Global Threshold $T_G$, Count $C_{i,k}$,
Output: Heavy Hitter Set (H)
Func HandleReport($i, C_{i,k}$):
    $ReportedC_{i,k} \leftarrow C_{i,k}$
    if Estimation ($k$) $> T_G$
        if GlobalPoll($k$) $> T_G$
            $H \leftarrow H \cup \{k\}$
        Reset_Threshold ($k$)

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

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

the coordinator reports key k as a heavy hitter with the count $\sum_i C_{i,k}$.

The coordinator adapts the per-flow local thresholds based on past reports. Each local threshold begins at $T_{i,k} = T_G/n$, and then is recomputed by the coordinator based on past reports. Algorithm 1 describes the actions taken by the coordinator after receiving a `Report`$(i, C_{i,k})$. Inspired by distributed rate limiting [20], the coordinator adapts the local thresholds based on the exponentially weighted moving average (EWMA) of the local 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 to reflect a site's share 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.

2.2 Local Counts and Threshold Checks

The switches can maintain the per-key state (local counts and thresholds) using a hash table. In the simplest case, the switch could 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 appropriate register and updates the associated count (e.g., a count of bytes or packets). If the count exceeds the threshold, the switch generates a report to the coordinator.

Implementation Our prototype of the switch data structures consists of fewer than 200 lines of P4 code to monitor per-key counts with adaptive thresholds in the data plane. We allocate two registers (hash tables) to store the count and the threshold for each key. The maximum number of entries per register determines the maximum number of keys that can be monitored in the data plane. When a packet arrives, match-action tables determine if it contains a monitored key and, if so, looks up the current count and threshold in each register. The switch generates a report for the coordinator if the count exceeds the threshold. Alternatively, the thresholds can also be stored as a parameter in a match-action table entry. Depending on the cost of updating match-action table entries or availability of register memory in different data plane targets, one can choose whichever implementation is appropriate for that target and forwarding logic. To generate a local report in the data plane, the switch clones the original packet, modifies it to send to the coordinator and embeds the count into that clone. Our implementation uses less than 2% of available SRAM on the Tofino target [23] and could easily co-exist with complex forwarding logic in the remaining switch memory.

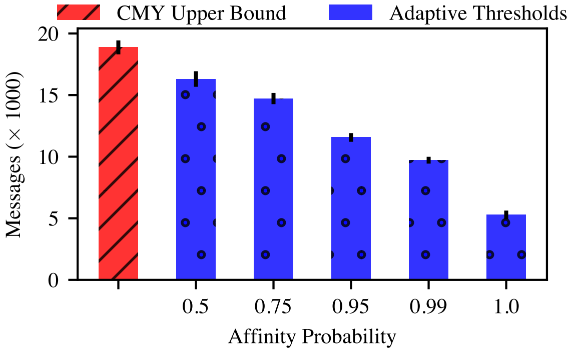

Figure 2: *Communication overhead decreases as a packet's affinity for a single ingress switch increases.*

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 per-key state for a realistic query and based on real-world traffic traces. Our system determines heavy hitters for a rolling time window (W). In our experiments, we choose a value of W=5 seconds which keeps the number of keys per window manageable. However, nothing about our approach prevents us from employing space saving algorithms and data structures [7, 22] if the memory constraints became prohibitive or we wanted to use much longer time windows. In Section 4, we will discuss how compact data structures can enhance our system beyond this 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. [6, 8, 9] as the baseline. We then quantify how the affinity for ingress switch affects the communication overhead of the solution in Section 3.2. Our evaluation shows that our algorithm improves upon the CMY upper bound for the countdown problem [6] by 8-75%. We also show how sensitive the reduction in the communication overhead is for different threshold values and number of edge sites (switches) in the network.

3.1 Experimental Setup

To quantify the performance of our approach, we used CAIDA's anonymized Internet traces from 2016 [24]. These traces consist of all the traffic traversing a single 10 Gbps link between Seattle and Chicago within a major ISP's backbone network. Each minute of the trace consists of approximately 640 million packets. For our experiments, we only considered UDP

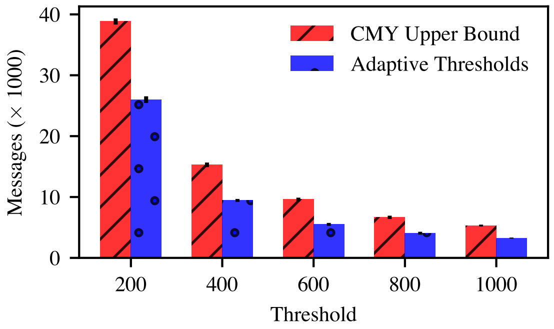

Figure 3: *As the global threshold increases, the fraction of overall traffic that constitute heavy hitters decreases, reducing the communication overhead.*

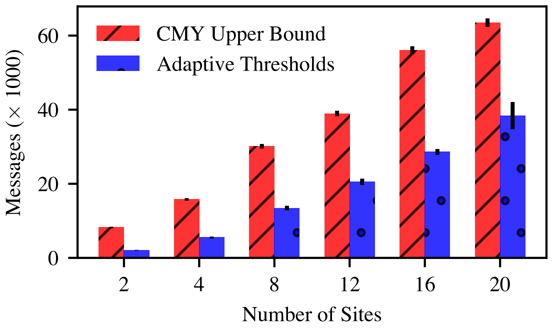

Figure 4: *As the number of sites increases, the communication overhead increases due to the additional nodes communicating with the coordinator.*

Thresholds	200	400	600	800	1000
Performance Gains (%)	33.0	38.2	42.6	38.9	38.7

Table 1: Performance gains over the CMY upper bound are not affected as the threshold increases.

Number of Sites	2	4	8	12	16	20
Performance Gains(%)	75.0	65.1	55.4	47.3	48.9	39.5

Table 2: Performance gains over the CMY upper bound reduce as the number of sites increases.

packets for the analysis and used a rolling time window of $W = 5$ seconds which results in processing approximately one million packets per window interval. For estimating the moving window average calculations, we used the weight factor, $\alpha = 0.8$ for all our experiments.

We simulate a one-big-switch network consisting of n edge nodes (switches). In order to model the spatial locality of the network traffic and to map the traffic from a single link onto the several switches, 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 the other $(n-1)$ switches with probability $(1-p)/(n-1)$. On this distribution of traffic, we run a simple heavy hitter query to determine which source IP, destination IP pairs send a number of packets greater than a threshold (T_G) during a rolling time window (W). Unless otherwise specified, we use $n = 4$, $p = 0.95$, and $T_G = 600$ for our experiments.

3.2 Communication Overhead

We now use realistic 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 [6]. We quantify the communication overhead in terms of median number of messages per window interval. To demonstrate the sensitivity of the performance improvement with respect to various parameters, we ran experiments varying three key parameters: (1) site affinity probability (p), (2) threshold (T_G), and (3) number of sites (n); 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 2 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 determined by its source IP address with probability 0.5 and $p = 1.0$ implies that packet will only be processed at the preferred site. As the affinity for a given site increases, i.e., as p increases, the performance of our algorithm that leverages this affinity improves significantly. We observe that up to 60% reduction is possible for $p = 1.0$. The reduction increases substantially from $p = 0.99$ 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 3 shows how the performance, quantified as the total number of messages sent over the entire experiment duration, varies as we increase the threshold value. 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 33% less communication overhead for $T_G = 600$, which corresponds to a top-k=100 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 4 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 the increase in number of sites has a small impact on the performance of our algorithm compared to the CMY bound. Table 2 shows that our solution incurs around 40% less communication overhead for $n = 20$.

4 COMPACT DATA STRUCTURES

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.

4.1 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 can keep a count-min sketch [7] to estimate the counts for *all* keys, and then only store counts and thresholds for keys with counts above some minimize size. A count-min sketch applies d hash functions on a key to calculate indices into an equal number of arrays. At those indices, the count-min sketch stores the count of the key. Since these counts are subject to hash collisions, each individual count is an overestimation of the true count, but the minimum of the the d counts will be an estimate of the true count within bounded error.

When an arriving packet matches an tracked key in the table, the switch updates the associated count and compares it to the threshold. When an arriving packet does not match a tracked key, the switch updates the count-min sketch and selectively creates a new entry in the table if the sketch shows that the count has exceeded some minimum threshold. Given that our approach from Section 2 has focused on providing exact counts, we would need to carefully choose minimum thresholds that bound the inaccuracy incurred using sketches.

4.2 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.

Destinations with distinct source counts exceeding a local threshold can be identified at distributed locations, but determining whether these switches see *different* sources or not is more challenging. Consider the case where a destination receives traffic from distinct sources s_1, s_2 at one switch and distinct sources s_1, s_3 at another switch. Both switches would report two distinct flows, but globally, there are three. Once the switch summarizes the distinct element set into a count, it cannot be combined with any other summary with any fidelity. Therefore, no threshold on the number of distinct flows could be evaluated locally at a switch.

Fortunately, we can again leverage an approximate algorithm known as HyperLogLog [11] to solve this problem. The count distinct problem is equivalent to the set cardinality problem; for this problem there exists an approximate solution that also performs with high accuracy [10, 11]. This algorithm can also be performed at distributed sites and later merged without any loss of accuracy. The key intuition behind this algorithm is that by storing a maximum value based on random inputs, a good estimate of the size of a set can be derived. These maximum values are stored in m buckets; when merging two estimates together, keeping the maximum value from each of the m buckets will produce a new estimate based on both previous estimates. Therefore, we can use this algorithm to generate local estimates, compare them against local thresholds, and merge the results at the controller just as with deterministic counts.

Implementation Implementing the HyperLogLog algorithm in the data plane is much more challenging than implementing adaptive thresholds due to the complexity of the algorithm. To implement the HyperLogLog algorithm, we require m registers to store a count. For each set that we want to estimate the cardinality of, i.e., count distinctly, we compute a hash value (h) of that item and break it into two components: an index i of p bits and a count c of leading zeros in the remaining $|h| - p$ bits. The maximum of c and the value stored in the i^{th} register is written back to that register. The harmonic mean of the values stored in these m

registers determines the estimate. Our implementation that uses adaptive thresholds and the HyperLogLog algorithm is less than 1600 lines of P4 code and uses less than 15% of available SRAM on the hardware target.

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 [7] required to calculate these statistics. Several systems [12, 17, 18, 22, 26] 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 single devices; 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 triggers 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 [20] and calculated using 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, 8, 9, 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.

Set Cardinality The set cardinality–also known as count distinct–problem has experienced a similar path of exploration. Heule *et al.* [13] improved upon the original HyperLogLog (HLL) algorithm [11] for effectively estimating set cardinality over streaming data in one pass. Sharma *et al.* [21] demonstrated that the HLL algorithm could be implemented on certain PISA switches using the Cavium Xpliant hardware target. We implement the HLL on the Barefoot Tofino hardware switch [23] which runs at line rate. Our work also focuses on using HLL, in conjunction with our adaptive thresholds mechanism, to detect network-wide heavy distincts.

6 CONCLUSION AND FUTURE WORK

Detecting heavy-hitters is an indispensable tool in managing and defending modern networks. We designed an efficient algorithm and implemented a real prototype for detecting network-wide heavy-hitters with commodity switches. Our evaluation with real-world traffic traces demonstrates that by leveraging the spatial locality of network traffic, 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 better data, we could explore how 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 [12]. 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 network telemetry system.

REFERENCES

[1] Brian Babcock and Chris Olston. 2003. Distributed top-k monitoring. In *Proceedings of the 2003 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*.

[4] B Claise. [n. d.]. RFC 5101: Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Traffic Flow Information, 2008. ([n. d.]).

[5] Benoit Claise. 2004. Cisco systems netflow services export version 9. (2004).

[6] Graham Cormode. 2011. Continuous distributed monitoring: a short survey. In *Proceedings of the First International Workshop on Algorithms and Models for Distributed Event Processing*. ACM, 1–10.

[7] 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.

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

[9] Graham Cormode, S Muthukrishnan, Ke Yi, and Qin Zhang. 2010. Optimal sampling from distributed streams. In *Proceedings of the twenty-ninth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, 77–86.

[10] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. 2007. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In *AofA: Analysis of Algorithms*. Discrete Mathematics and Theoretical Computer Science, 137–156.

[11] Philippe Flajolet and G Nigel Martin. 1985. Probabilistic counting algorithms for data base applications. *Journal of computer and system sciences* 31, 2 (1985), 182–209.

[12] 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).

[13] Stefan Heule, Marc Nunkesser, and Alexander Hall. 2013. HyperLogLog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm. In *Proceedings of the 16th International Conference on Extending Database Technology*. ACM, 683–692.

[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 *Proceedings of 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 *Proceedings of the 2006 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 *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 *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*. ACM, 101–114.

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

[20] Barath Raghavan, Kashi Vishwanath, Sriram Ramabhadran, Kenneth Yocum, and Alex C. Snoeren. 2007. Cloud Control with Distributed Rate Limiting. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '07)*. ACM, New York, NY, USA, 337–348. https://doi.org/10.1145/1282380.1282419

[21] Naveen Kr Sharma, Antoine Kaufmann, Thomas E Anderson, Arvind Krishnamurthy, Jacob Nelson, and Simon Peter. 2017. Evaluating the Power of Flexible Packet Processing for Network Resource Allocation.. In *NSDI*. 67–82.

[22] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S Muthukrishnan, and Jennifer Rexford. 2016. Smoking Out the Heavy-Hitter Flows with HashPipe. *arXiv preprint arXiv:1611.04825* (2016).

[23] url [n. d.]. Barefoot's Tofino. https://www.barefootnetworks.com/technology/. ([n. d.]).

[24] url 2017. The CAIDA UCSD Anonymized Internet Traces 2016. http://www.caida.org/data/passive/passive_2016_dataset.xml. (jun 2017).

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

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