Carpe Elephants: Seize the Global Heavy Hitters

Rob Harrison
United States Military Academy
rob.harrison@westpoint.edu

Shir Landau Feibish
Princeton University
sfeibish@cs.princeton.edu

Arpit Gupta
UC Santa Barbara
arpitgupta@cs.ucsb.edu

Ross Teixeira
Princeton University
rapt@cs.princeton.edu

S. Muthukrishnan
Rutgers University
smewtoo@gmail.com

Jennifer Rexford
Princeton University
jrex@cs.princeton.edu

ABSTRACT

Detecting "heavy hitter" flows is the core of many network security applications. While past work shows how to measure heavy hitters on a single switch, network operators often need to identify *network-wide* heavy hitters on a small timescale to react quickly to distributed attacks. Detecting network-wide heavy hitters efficiently requires striking a careful balance between the memory and processing resources required on each switch and the network-wide coordination protocol. We present Carpe, a distributed system for detecting network-wide heavy hitters with high accuracy under communication and state constraints. Our solution combines probabilistic counting techniques on the switches with probabilistic reporting to a central coordinator. Based on these reports, the coordinator adapts the reporting threshold and probability at each switch to the spatial locality of the flows. Simulations using traffic traces show that our prototype can detect network-wide heavy hitters with 97% accuracy, while reducing the communication overhead by 17% and switch state by 38%, compared to existing approaches.

CCS CONCEPTS

• **Networks → Network monitoring**.

KEYWORDS

Network-wide monitoring, Heavy hitters

1 INTRODUCTION

Network operators continuously monitor their traffic to detect the *heavy-hitter* flows responsible for attacks and congestion. However, the recent proliferation of IoT (Internet of Things) and mobile devices makes network measurement more challenging. Even if each device sends a small volume of traffic, a large group of devices can easily create a substantial DDoS (Distributed Denial of Service) attack. Such traffic would not necessarily be "heavy" at any single vantage point, leading to the need to detect *network-wide* heavy hitters [5]. Furthermore, even short-term traffic bursts, such as TCP incast [6] or Slowloris attacks [21], can cause significant disruptions for interactive applications like videoconferencing, augmented/virtual reality, and cyberphysical systems. Supporting these applications requires effective ways to detect network-wide heavy hitters in *real time*.

In currently deployed solutions like sFlow [17] and NetFlow [7], each switch sends a small sample of traffic (e.g., 1 out of 4000 packets [15]) to a central collector for network-wide analysis. Unfortunately, the low sampling rates needed for reasonable overhead also lead to low accuracy for detecting heavy hitters on a small timescale. To achieve higher accuracy without excessive overhead, the switches must play a bigger role in identifying the heavy hitters. Indeed, approximate streaming algorithms can identify heavy-hitter flows on a single switch, despite limited memory and processing resources [3, 16, 20, 26]. Nonetheless, flows that generate a large volume of traffic for the network in total, but are not heavy at any one switch, can go undetected.

To achieve high accuracy at reasonable overhead, we need *coordination* between the collector and the distributed set of switches [8]. The central coordinator should aggregate the information observed at each switch to identify flows whose aggregate count exceeds a global threshold. Deciding what a switch should report, and when, determines the communication overhead and the accuracy of the results. For example, recent work [2, 4, 15] shows how to periodically collect and combine sketches from multiple locations to compute a network-wide estimate of the traffic. However, these techniques fail to strike the balance between *real-time* analysis and low communication overhead. Previous theoretical work [9] presented techniques that can reduce communication overhead, but requires per-flow state in the switches.

We present Carpe (Counting And Reporting Probabilistically for Estimation), a practical monitoring system for detecting network-wide heavy hitters in real time, with high accuracy, and under communication and state constraints. We extend the standard taxonomy of mice and elephant flows to more accurately describe the costs of performing heavy-hitter detection in a network-wide setting. Carpe instructs each switch to probabilistically identify and report potentially important flows, while accounting for the locality of flows to minimize communication. Our solution extends the probabilistic reporting techniques presented in [25] by combining them with a probabilistic flow count mechanism, to report measurements of individual flows in real time. Using this approach we are able to get the benefits of the probabilistic reporting while limiting the required memory on the switches. We summarize our contributions as follows:

Continuous, communication-efficient coordination. We developed a new coordination protocol for detecting network-wide heavy hitters that uses adaptive thresholds to account for flow

SPIN'20, August 14, 2020, Virtual Event, NY, USA

ACM ISBN 978-1-4503-8041-6/20/08…$15.00
https://doi.org/10.1145/3405669.3405820

locality. This protocol probabilistically reports when switches observe a non-trivial contribution from a monitored flow and infers network-wide heavy hitters at the coordinator from these reports. Probabilistic reporting can scale to large networks, as it achieves an error bound on the network-wide count that is independent of the number of switches in the network [25]. Our analysis shows that this protocol reduces the communication cost by 17% for achieving 97% accuracy compared to sampling.

Memory-efficient switch data structure. We present a data structure that efficiently stores the counters for flows that make an important contribution to a network-wide count. This data structure probabilistically determines the subset of flows to monitor at the switch from a larger traffic stream. This data structure can be implemented in modern PISA (Protocol Independent Switch Architecture) switches. Our evaluation shows that our solution requires 40% less switch memory at the expense of 3% degradation in detection accuracy when compared to counting all of the flows.

In § 2, we present a taxonomy of flows relevant to detecting heavy-hitters in a network-wide setting. We present the design of the coordination protocol to detect these flows in § 3, and the switch data structure and prototype in § 4. We evaluate Carpe in § 5. Related work is discussed in § 6, and a summary in § 7.

2 CARPE: WHO'S WHO IN THE ZOO

The Carpe system consists of a distributed set of tens, hundreds, or even thousands of switches and a centralized coordinator. The coordinator aggregates the partial information observed at each switch to identify the network-wide elephant flows whose aggregate counts exceed a threshold. To minimize overhead, the switches use probabilistic techniques to decide both when to maintain state about a given flow and also which flows to report to the coordinator. Figure 1 shows a simple example where two switches (A and B) communicate with a central coordinator (C) to determine the network-wide elephants. The tables below each switch show the actual counts observed for flows f_1-f_5 at each switch.

Mice (no state or communication): Each switch will observe a large number of *mice*, i.e., flows that will never become local, let alone network-wide, heavy hitters. Traditional approaches have typically employed approximate data structures to deal with the numerous mouse flows. However, the many mice flows can compromise the accuracy of the counts stored in an approximate data structure–even those for other non-mouse flows. Rather than overprovisioning an approximate data structure to achieve high accuracy in the face of numerous mouse flows, Carpe switches do not waste scarce resources maintaining state for mice. Instead, Carpe samples the flows observed at each switch and only stores counts for the sampled flows. Rather than storing *more* state to cope with these numerous, tiny flows, we ignore them entirely. In Figure 1, we consider flows of fewer than 5 packets as mice (colored red).

Moles (local state, no communication): Sampling flows reduces the amount of state required at each switch by filtering out most mice, but not all sampled flows will ever become a global heavy hitter. A flow that is sampled becomes a *mole* that the switch counts locally while waiting to determine whether the flow's count grows large enough to consider reporting it to the coordinator. By requiring that moles reach some threshold before reporting them to the coordinator, each switch can sample packets aggressively

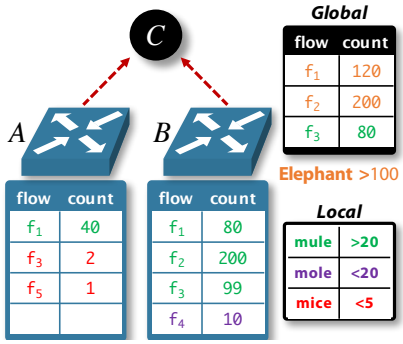

Figure 1: Example with mice, moles, mules, and elephants.

enough to catch flows that *may* have global significance, while still limiting the number of reports sent to the coordinator. Each switch maintains a "sample and hold" data structure [11] for all moles. On each incoming packet, switches check whether the packet's flow is already stored in the data structure. If so, the switch increments the corresponding counter; otherwise, the switch creates a new counter for the flow with sampling rate s. This approach inherently defines the set of moles as those whose count is greater than $(1/s)$, in expectation. For example, a sampling rate of 0.2 would, on average, result in a set of mole flows which all consist of five or more packets, as shown by the purple flows in Figure 1.

Mules (state and communication): When a mole becomes large enough to impact a global heavy-hitter count, the flow becomes a *mule* and the switch considers reporting the flow to the coordinator. Mules are counted locally at the switch *and* globally at the coordinator based on the reports from one or more switches. We set the reporting threshold $\tau > 1/s$ to ensure that the set of mule flows is strictly smaller than the set of moles. A switch could report to the coordinator each time a flow reaches a count of τ packets. However, this would cause the residual counts at each switch (which are not reported) to make the coordinator's count have an error proportional to the number of switches. This would limit our system's ability to support large networks. Instead, Carpe generates a report with probability r each time τ packets are observed for a sampled flow, allowing Carpe to achieve an error bound that is independent of the number of switches [25]. Going further, Carpe sets τ based on the number of switches that actually observe a flow to capitalize on spatial locality in network traffic. In Figure 1, flows with a count of at least 20, in expectation, are considered mules (see green flows).

Elephants (network-wide counts): The goal of Carpe is to identify the *elephant* flows that exceed a global threshold and estimate their packet counts. Based on reports sent from the switches, the coordinator determines when a mule becomes a network-wide elephant. The coordinator identifies a mule as an elephant after receiving a total of R reports for a flow from any of the switches.

Example: As seen in Figure 1, flows that exceed a global count of 100 are elephants (orange flows). The tables below each switch in the figure show the actual packet counts observed for each flow at that switch. The coordinator's count for each flow is based on reports sent by the switches. The coordinator is aware of all the mules (f_1, f_2, and f_3) from both switches, but determines that only

Symbol	Meaning
T	Global threshold
k	Total number of ingress switches
l	Number of switches which observe a flow
ϵ	Approximation factor
τ	Local (mule) threshold
r	Reporting probability to coordinator
s	Sampling probability at a switch
R	Number of reports expected at coordinator

Table 1: Network-Wide Heavy Hitter Parameters

f_1 and f_2 are elephants. In the case of f_3, notice how both switches and the coordinator all have *different* views of the total count for this flow. Since f_3 is a mouse at switch A, the switch should actually store no information about the flow's count at all. At switch B, flow f_3 is a mule locally, but since the switch reports to the coordinator only once every 20 counts (the mule threshold), the coordinator believes the global count of f_3 is only 80. In fact, the global count of f_3 meets the network-wide threshold of 100, but our taxonomy of flows and their reporting requirements would not identify f_3 as a network-wide elephant; this is by design.

3 COORDINATION PROTOCOL

Switches must efficiently communicate to the coordinator when they have observed counts that could be significant network-wide. In this section, we describe a coordination protocol that determines when a switch should report a flow, and how the coordinator uses these reports to determine the network-wide elephant flows. We then describe an extension to this protocol that leverages the spatial locality of network traffic to reduce the communication cost of the protocol. Table 1 summarizes the relevant system parameters.

3.1 Probabilistic Reporting of Local Counts

Scaling to Large Networks: A switch could report to the coordinator, each time a *bundle* of τ packets from a mule flow is observed at each switch. However, the communication cost of this strategy grows proportional to the number of switches in the network. Because a switch only reports a flow once for every τ packets it observes, other switches may have residual flow counts smaller than τ which have not yet been reported. In aggregate, these residual counts represent a "blind spot" for the coordinator and necessarily cause inaccuracies in the global estimate that it maintains.

As τ increases or the number of switches grows large, the inaccuracy of the final results will increase. One possible way to reduce the inaccuracy is to significantly lower τ. However, that would increase communication between the switches and the coordinator, because each switch will produce many more reports for each mule flow. Prior work proposed a probabilistic reporting approach that scales to networks with a large number of switches and proved its efficiency [9]. Specifically, they show that for a given error probability, the communication complexity of their algorithm is independent of k. However, this technique requires maintaining unlimited state and performing advanced computations, such as an arbitrary number of coin tosses with varying probabilities. Therefore, implementing this technique has proven to be challenging, and has yet to be implemented in production systems [8]. We adapt this technique to account for flow locality and enable it to be executed on modern programmable switches.

Probabilistically Separating Elephants and Mules: For every packet received by the switch, a flow identifier f is extracted. Our algorithm then updates the counter for f and checks if the counter has exceeded the local threshold (τ) (see Section 4 for further details), in which case the switch reports the flow to the coordinator with probability r. A simple key-value store (D), indexed by flow identifier, stores the counts at the switch and the parameters τ and r apply to all flows. By reporting with probability r, each reported bundle now represents a count of τ/r in expectation, which reduces the total number of reports that must be sent. When the coordinator receives a report for flow f, it increases the number of reports for f by 1. When the number of reports received for flow f exceeds the threshold R, the coordinator determines that it is an elephant flow.

Configuring Parameters: Configuring the parameters (τ, r, and R) to strike a balance between accuracy and communication cost is non-trivial. For example, we want to set τ high enough such that it reduces the number of reports sent to the coordinator for mule flows but low enough so that the size of the bundles does not reduce the accuracy of the global count estimates. Previous work [9] demonstrated tight bounds on communication and error by selecting an approximation factor (ϵ) and reporting with a probability $r = 1/k$. Their results show that this approach can achieve high accuracy with modest communication overhead that does not grow proportionally to the number of switches in the network. We generalize the results from prior work by setting $r = 1/k$, $\tau = \epsilon T/k$, for $0 < \epsilon < 1$; the coordinator then determines that a flow is an elephant after receiving $R = kr/\epsilon$ reports. When $r = 1/k$ this threshold simplifies to $R = 1/\epsilon$, which is independent of the number of switches in the network.

3.2 Locality-aware Reporting Parameters

The protocol we described in the previous section implicitly assumes that all of the k switches in the network are equally likely to observe a portion of the traffic for a given flow. This assumption results in lower local thresholds (τ) as networks grow large. A smaller τ will result in the switch determining that more mole flows are mules, and, ultimately, increase the communication cost. However, in practice, most flows exhibit spatial locality, i.e., only a subset of switches observe traffic for a given flow. If a flow is only observed at $l << k$ locations, then we should configure the parameters based on this smaller number of switches. Accounting for this locality would increase τ, which, in turn, ensures that fewer moles are unnecessarily promoted to mules, thus reducing the communication cost.

Configuring Parameters: We now require an additional parameter l_f to account for the spatial locality. Here, l_f denotes the *number* of switches that observe flow f. For now, we can assume that we know the locality parameters for all flows *a priori* because forwarding state can be used to infer this information. Accounting for this locality parameter, we adjust the local threshold as $\tau_f = \epsilon T/l_f$ and reporting probability as $r_f = 1/l_f$ for each flow at the switch. The coordinator reports flow f as a heavy hitter when it receives $R = 1/\epsilon$ reports from the switches. The key-value store

D in the switch must be augmented to maintain the values for τ_f and r_f for each flow.

Example: Let us consider an example topology with $k = 10$ switches, global threshold $T = 2500$ and we choose an approximation factor of $\epsilon = 0.1$. Without considering locality, switches would report every $\tau = 25$ packets to the coordinator with probability $r = 0.1$, and the coordinator would declare any mule an elephant after receiving $R = 10$ reports. Here, a single τ and r apply to all flows at all switches in the network. If we account for locality, considering a particular flow f that is observed only at $l_f = 2$ switches, we can increase both our bundle size to $\tau_f = 125$ and reporting probability to $r_f = 0.5$. The coordinator would still declare a mule an elephant after receiving $R = 10$ reports for the flow, but there are now fewer mules sending reports to the coordinator due to the larger bundle size. Here, τ and r can vary based on the number of switches that observe the flow.

Tracking Spatial Locality: However, the locality of flows is not static; it changes due to failures and routing updates. Carpe tracks changes in the spatial locality for flows directly in the data plane. At all times, a switch has knowledge of which flows it expects to observe. Upon receiving a packet from an unexpected flow, the switch sends a `Hello` message to the coordinator. The coordinator extracts the flow (f) and switch identifier (s) from the `Hello` message, and updates the value of l_f. To avoid putting that computation in the data plane, the coordinator calculates the parameters r_f and τ_f accordingly. Finally, it sends the updated r_f and τ_f parameters to the switch that sent the `Hello` message. In addition, the coordinator maintains a mapping of flows to switches. The set S_f of switches that also observe the flow should be notified of the updated parameters. However, to avoid updating the parameter due to spurious or transient conditions, we update the locality parameters for a flow only when the size of S_f doubles.

Security Benefits of Spatial Locality: Most DDoS attacks are launched from huge botnets, causing attack traffic to be widely dispersed [23], and difficult to detect. If the local threshold is a predefined constant [1], an attacker could potentially learn that threshold and send their traffic so that the count observed at each switch stays below the threshold. Carpe decreases reporting thresholds as a flow's traffic becomes more widespread making this type of evasion more difficult.

Coordinator Structures and Security: As shown in Figure 2, the coordinator tracks the set of switches that observe each reported flow f[1], as well as the number of reports received for f. While the switch may observe a large number of flows, the coordinator is only required to maintain state for a smaller number of mule flows reported by the switches. The coordinator data structures are resilient to DDoS attacks for two main reasons: first, an adversary would need to create a very large amount of traffic in order to trigger an overwhelming number of reports to the coordinator. Second, the coordinator only receives and handles reports from the switches in its own network; it should not be publicly accessible to the rest of the Internet.

[1]This information will be maintained at the granularity of groups as described in Section 4.

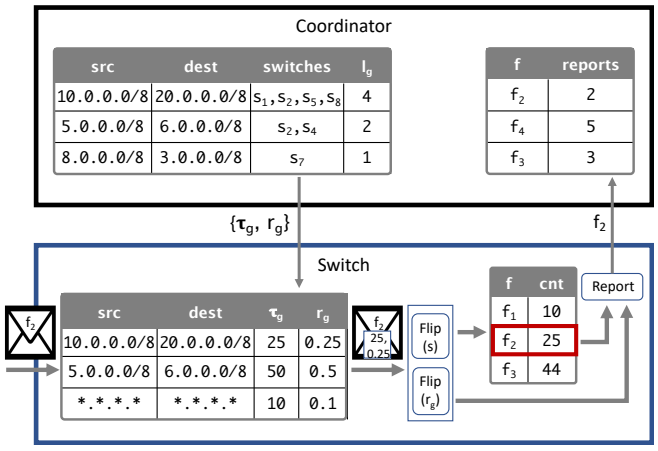

Figure 2: Data structures at a single switch and the coordinator, $T = 1000$, $\epsilon = 0.1$, $k = 10$.

4 SWITCH DATA STRUCTURE

Carpe's coordination protocol assumes that switches store the reporting probability (r_f) and local threshold (τ_f) for each flow based on the locality parameter (l_f). However, the memory in modern programmable switches is orders of magnitude too small to store per-flow state. Specifically, in PISA switches [22], packet processing is done in a multi-stage pipeline. Each stage consists of one or more match-action tables (MAT) that consume the finite TCAM, SRAM, and ALUs to match on and transform fields in packet headers. Carpe efficiently uses the finite switch memory by decoupling the granularity at which flows must be counted from the granularity at which locality may be observed. We store reporting parameters at the granularity of *groups* of related flows, while storing counters separately for individual flows that have been sampled.

Retrieving Flow Parameters: We observe that routing and forwarding decisions affecting flow locality are usually made at a coarser granularity, e.g., source-destination IP prefix pairs, while individual flows are often monitored at a finer granularity. Therefore, we have the switch store reporting parameters at this coarser granularity, while storing traffic *counts* at a finer granularity (e.g., five-tuple). We define a group ($g_{src,dst}$) based on source-destination IP prefix pairs, such that $g_{src,dst} = \{f | f.srcIP \in src, f.dstIP \in dst\}$. We store a group ($g$) at a switch if at least one flow $f \in g$ is observed at the switch. In addition, default parameters are stored for groups that are observed but have not yet been stored. The coordinator keeps track of the groups that are observed at each switch and updates the switch with the appropriate parameters.

Each flow group g stored in the switch has an associated match-action table (MAT) rule, implemented using TCAM to match on source and destination IP prefixes. The action of each rule reads the associated local threshold (τ_g) and reporting probability (r_g). As seen in Figure 2, when a packet enters the PISA pipeline, we first determine to which group (g) the packet belongs. The group identifier of each packet is matched against the relevant MAT rule to identify the correct reporting parameters for that group. Note that for a network with k switches, each switch only needs to store k different sets of locality parameters.

Coin Toss: We use sampling to effectively filter out the small flows, in expectation. As a packet traverses the switch, once the group identifier and locality parameters are determined, two independent but biased coin flips are performed; the first coin flip is based on the sampling probability (s) and the second based on the reporting probability (r_g) values. Both coin flip results are stored as packet metadata values $flip_1$ and $flip_2$, respectively, for use later in the pipeline. Existing PISA switches do not support floating point operations. Therefore, each $flip$ is performed similarly to [3], by representing a floating value as an unsigned integer ($0 < i \leq 1.0$), then using packet header fields to compute a 32-bit hash value and testing if it is smaller than $\lceil 2^{32} i \rceil$. Note that this approach introduces a small quantization error since we can only represent probabilities as multiples of $\frac{1}{2^{32}}$.

Count and Reset: For flows that we do sample, we store an exact counter. The switch checks if f is currently in the key-value store D, which is a hash-indexed register array of limited size. To reduce collisions, D may be implemented as a multi-stage hash-table that uses a different hash function in each stage. When collisions do occur, one possible approach would be to offload treatment of those flows to the control plane — trading communication cost for accuracy. If the flow is in D, its counter is incremented by 1. Otherwise, if $flip_1$ == `true`, i.e. the flow is "sampled", the switch inserts f into D and initializes the count in D to an initial value $v = \frac{1}{s}$. If the counter is greater than τ_g, it is reset to 0.

Report: If $flip_2$ == `true`, a report is then sent for this flow. Note that we reset the count regardless of whether the report had been sent or not. As seen in Figure 2, the coordinator receives the report and updates the number of reports received for that flow, which is used by the coordinator to classify the flow into the appropriate category.

5 PERFORMANCE EVALUATION

In this section, we quantify how Carpe makes efficient use of limited communication and memory resources to detect network-wide heavy hitters with as high accuracy as possible. We use real-world packet traces [18] to demonstrate how combining probabilistic counting with probabilistic reporting reduces Carpe's memory footprint by 38% and bandwidth footprint by 17% (compared with sampling) to report network-wide heavy hitters with up to 97% accuracy.

5.1 Trace-Driven Simulation Model

To quantify Carpe's performance, we run a simple network-wide 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) during a sliding time window.

Simulation experiments. For our experiments, we assume that we monitor at the edge switches of the network where the number of edge switches (k) is 10—representative of a wide-area network connecting multiple data centers for cloud providers [14]. For all experiments, each flow shows affinity for two ingress switches, i.e., $l = 2$, based on the source IP address. For the purpose of evaluation, we choose a global threshold that corresponds to the 99.99^{th} percentile flow count in the packet trace. In practice, an operator could select whatever threshold is important to them. We also note that our simulations use a real-world data trace from a single point-to-point link in a production Internet service provider (ISP) network. Therefore, we limit our evaluation to a network of 10 simulated switches because dividing this traffic across more simulated switches would only create smaller, unrealistic workloads on each switch. Instead we seek to preserve, as much as possible, the real-world flow size distributions and minimize the artifacts created by spreading the traffic across simulated switches. Furthermore, experimenting with a larger network would not provide additional insight. Since spatial locality is often limited to a subset of nodes in a network, adding more switches to the network should not affect the subset of nodes that display locality.

Technique	Prob. Count	Prob. Report	State
Strawman	✗	✗	345K
RLA	✗	✓	345K
Sampling	✗	✓	N/A
Carpe	✓	✓	211K

Table 2: Comparison with other Heavy-Hitter Detection Techniques. Carpe uses both probabilistic counting and reporting where other approaches use only one.

Packet traces. To emulate real-world traffic distributions, we used CAIDA's anonymized Internet traces from 2016 [18]. 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 use a time window of five seconds resulting in around 5 million packets and hence $\approx$ 270K unique flows per window.

Since the packet traces are collected from a single link only, 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 the other switches with probability $(1 - p)$. For the case of $l = 2$, this distribution simulates a primary/alternate relationship on ingress for a single source and $p = 0.95$.

5.2 Carpe Accuracy and Overhead

To demonstrate the benefits of combining probabilistic counting and reporting, we quantify the memory and communication overhead for Carpe and compare it with the existing heavy-hitter detection techniques that either employ probabilistic counting or reporting, but not both.

Alternative Approaches. First, we consider a strawman solution that makes use of neither probabilistic counting or reporting; each switch maintains counters for every flow (all flows are moles) and reports all the counters to a central coordinator at the end of a window. Second, we consider RLA *(randomized locality-agnostic)*, a solution based on the randomized reporting technique [9]; where the switch still treats all flows as moles, but it probabilistically

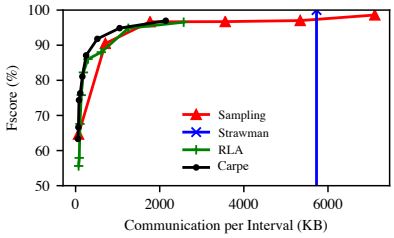

Figure 3: Accuracy comparison

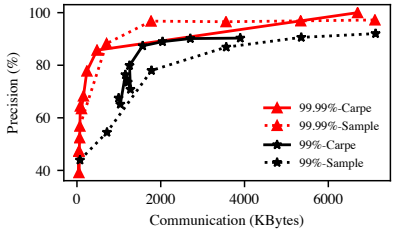

Figure 4: Precision vs. overhead

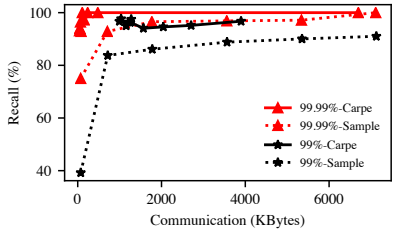

Figure 5: Recall vs. overhead

reports mules to the coordinator with parameters that ignore locality. Finally, we consider a solution based on packet sampling (e.g., sFlow) [17], which probabilistically samples packets and reports all of those samples to the coordinator. Performance of sampling-based solutions depend on sampling rate; high sampling rates result in higher accuracy but prohibitively costly communication overhead, and vice versa. For our experiments, we chose sampling rate of 0.001 as the communication overhead was comparable to Carpe and *RLA* at this rate. We run Carpe with an approximation factor of $\epsilon = 0.1$.

Communication and State Savings. We quantify the state overhead as the number of stateful counters required at the switch and the communication overhead as kilobytes sent to the coordinator for each window interval. We quantify accuracy in terms of both precision (*PR*) and recall (*RE*) and present them as a single F_1 score calculated as $\frac{2\times PR\times RE}{PR+RE}$. In Table 2, we see that Carpe achieves 38% savings in the state required for alternative approaches. In Figure 3, we compare how much communication is needed to reach a certain level of accuracy. We see that to achieve an F_1 score of 97%, Carpe communicates 17% less than sampling packets with a probability of 0.075.

Sensitivity to Heavy-Hitter Threshold. As the threshold determining heavy-hitters decreases, this advantage becomes more pronounced. In Figures 4 and 5, we show the communication/accuracy tradeoff compared with sampling for three different heavy-hitter thresholds ranging from the 99.99^{th} to the 99^{th} percentile thresholds. Carpe performs strictly better than the sampling approach in all cases, except in the 99.99^{th} percentile threshold where the sampling probability is greater than 0.05—an unrealistically-high sampling probability for modern data centers.

6 RELATED WORK

Scalable measurement techniques. NetFlow [7], a standardized approach for collecting telemetry data from network devices, requires significant CPU overhead or specialized hardware to run efficiently. Packet sampling [17] emerged as the de facto technique to cope with both the memory and communication limitations. However, packet sampling can introduce significant inaccuracy to detecting heavy hitters on small timescales [17]. FlowRadar [15] reduces the memory and communication overhead using a novel encoding of flow counters. Similarly, CSamp [19] provides a sampling mechanism for network-wide measurements. While both of these works are general-purpose solutions for performing network-wide measurement of most flows, we offer a tailored solution for continuous, network-wide monitoring of a global threshold distributed across several switches.

Heavy hitters with limited state. ElasticSketch [24] offers a technique to avoid maintaining state for mouse flows in the data plane by offloading computation to the control plane. Other solutions, such as SketchVisor [13], rely on *software* packet processing which limits their abilities to handle high data rates. Our approach, based on the sample-and-hold [11] technique, uses sampling to filter mice flows in the data plane and only maintains per-flow state for potential heavy hitters. Similar techniques have been used for identifying heavy hitters in a partial deployment setting [10].

Network-wide heavy hitters with low communication overhead. Analysis of the continuous distributed monitoring (CDM) problem [8] provides bounds on communication complexity [9, 25] for both deterministic and randomized solutions. We extend the model to account for flow affinity and the capabilities of programmable data planes. Recent work [12] showed that adaptive local thresholds can reduce communication overhead for computing network-wide heavy hitters. While this solution reports network-wide heavy hitters deterministically and without errors, its communication cost grows in proportion to the number of switches in the network. That solution also reports its cost savings against a deterministic algorithm not widely used in practice. In contrast, our solution reports network-wide heavy hitters probabilistically, which introduces modest inaccuracy, but our solution's communication costs do not grow proportionally with the number of switches as a result.

7 CONCLUSION

Carpe detects network-wide heavy hitters with high accuracy under communication and state constraints. We combined probabilistic counting on the switches with probabilistic reporting to the central coordinator. Based on these reports, the coordinator adapts the parameters at each switch to the spatial locality of the flows to compute network-wide heavy hitters accurately and efficiently.

ACKNOWLEDGMENTS

The views expressed in this article are those of the authors and do not reflect the official policy or position of the Department of the Army, Department of Defense or the U.S. Government. This research was also supported by NSF Grant CCF-1535948.

REFERENCES

[1] Yehuda Afek, Anat Bremler-Barr, Shir Landau Feibish, and Liron Schiff. 2018. Detecting heavy flows in the SDN match and action model. *Computer Networks*

136 (2018), 1–12.
[2] Ran Ben-Basat, Xiaoqi Chen, Gil Einziger, Shir Landau Feibish, Danny Raz, and Minlan Yu. 2020. Routing Oblivious Measurement Analytics. In *IFIP Networking Conference*.
[3] Ran Ben-Basat, Xiaoqi Chen, Gil Einziger, and Ori Rottenstreich. 2018. Efficient Measurement on Programmable Switches Using Probabilistic Recirculation. In *IEEE International Conference on Network Protocols ICNP*. 313–323.
[4] Ran Ben-Basat, Gil Einziger, Shir Landau Feibish, Jalil Moraney, and Danny Raz. 2018. Network-wide routing-oblivious heavy hitters. In *Symposium on Architectures for Networking and Communications Systems ANCS*. 66–73.
[5] Theophilus Benson and Balakrishnan Chandrasekaran. 2017. Sounding the Bell for Improving Internet (of Things) Security. In *Workshop on Internet of Things Security and Privacy, IoT S&P@CCS*. 77–82.
[6] Yanpei Chen, Rean Griffiths, David Zats, Anthony D. Joseph, and Randy H. Katz. 2012. Understanding TCP Incast and its Implications for Big Data Workloads. *;login* 37, 3 (June 2012).
[7] Benoit Claise. 2004. Cisco Systems NetFlow Services Export Version 9. *RFC 3954* (2004).
[8] Graham Cormode. 2011. Continuous Distributed Monitoring: A Short Survey. In *International Workshop on Algorithms and Models for Distributed Event Processing*.
[9] Graham Cormode, S Muthukrishnan, and Ke Yi. 2011. Algorithms for Distributed Functional Monitoring. *ACM Transactions on Algorithms* 7, 2 (2011), 21:1–21:20.
[10] Damu Ding, Marco Savi, Gianni Antichi, and Domenico Siracusa. 2020. An Incrementally-Deployable P4-Enabled Architecture for Network-Wide Heavy-Hitter Detection. *IEEE Transactions on Network and Service Management* 17, 1 (2020), 75–88.
[11] Cristian Estan and George Varghese. 2003. New Directions in Traffic Measurement and Accounting: Focusing on the Elephants, Ignoring the Mice. *ACM Transactions on Computer Systems* 21, 3 (2003), 270–313.
[12] Rob Harrison, Qizhe Cai, Arpit Gupta, and Jennifer Rexford. 2018. Network-Wide Heavy Hitter Detection with Commodity Switches. In *ACM SIGCOMM Symposium on SDN Research SOSR*. 8:1–8:7.
[13] Qun Huang, Xin Jin, Patrick P. C. Lee, Runhui Li, Lu Tang, Yi-Chao Chen, and Gong Zhang. 2017. SketchVisor: Robust Network Measurement for Software Packet Processing. In *ACM SIGCOMM*. 113–126.
[14] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. 2013. B4: Experience with a Globally-deployed Software Defined WAN. In *ACM SIGCOMM*. 74–87.
[15] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. 2016. FlowRadar: A Better NetFlow for Data Centers. In *USENIX NSDI*. 311–324.
[16] 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*. 101–114.
[17] P. Phaal, S. Panchen, and N. McKee. 2001. InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks. *RFC 3176* (2001).
[18] report [n. d.]. The CAIDA Anonymized Internet Traces 2016 Dataset. https://www.caida.org/data/passive/passive_2016_dataset.xml. ([n. d.]).
[19] Vyas Sekar, Michael K. Reiter, Walter Willinger, Hui Zhang, Ramana Rao Kompella, and David G. Andersen. 2008. cSamp: A System for Network-Wide Flow Monitoring. In *USENIX NSDI*. 233–246.
[20] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer Rexford. 2017. Heavy-Hitter Detection Entirely in the Data Plane. In *ACM SIGCOMM Symposium on SDN Research SOSR*. 164–176.
[21] slowloris 2009. Slowloris HTTP DoS. https://web.archive.org/web/20150426090206/http://ha.ckers.org/slowloris. (June 2009).
[22] url [n. d.]. Barefoot's Tofino. https://www.barefootnetworks.com/technology/. ([n. d.]).
[23] An Wang, Wentao Chang, Songqing Chen, and Aziz Mohaisen. 2018. Delving Into Internet DDoS Attacks by Botnets: Characterization and Analysis. *IEEE/ACM Transactions on Networking* 26, 6 (2018), 2843–2855.
[24] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. 2018. Elastic Sketch: Adaptive and Fast Network-Wide Measurements. In *ACM SIGCOMM*. 561–575.
[25] Ke Yi and Qin Zhang. 2009. Optimal Tracking of Distributed Heavy Hitters and Quantiles. In *ACM SIGMOD-SIGART-SIGACT Symposium on Principles of Database Systems PODS*. 167–174.
[26] Minlan Yu, Lavanya Jose, and Rui Miao. 2013. Software Defined Traffic Measurement with OpenSketch. In *USENIX NSDI*. 29–42.