Network-Wide Heavy-Hitter Detection for Real-Time Telemetry

Qizhe Cai

master's thesis

Presented to the Faculty

of Princeton University

in Candidacy for the Degree

of Master of Science

Recommended for Acceptance

by the Department of

Computer Science

Adviser: Jennifer Rexford

June 2018

Abstract

Many network monitoring tasks identify subsets of traffic that stand out, *e.g.*, top-k flows for a particular statistic. We can efficiently determine these “heavy-hitter” flows on individual network elements, but network operators often want to identify interesting traffic on a *network-wide* basis. Determining the heavy hitters on a network-wide basis necessarily introduces a trade-off between the communication required to perform this distributed computation and the accuracy of the results. To perform distributed heavy-hitter detection in real time with high accuracy and low overhead, we extend the Continuous Distributed Monitoring (CDM) model to account for the realities of modern networks and devise practical solutions that detect heavy hitters with high accuracy and low communication overhead. We present two novel algorithms that automatically tune the set of monitoring switches in response to traffic dynamics. We implement our system using the P4 language, and evaluate it using real-world packet traces. We demonstrate that our solutions can accurately detect network-wide heavy hitters with up to 70% savings in communication overhead compared to existing approaches.

Acknowledgements

This work is collaborative work with Rob Harrison (Princeton University), Shir Landau Feibish (Princeton University), Arpit Gupta (Princeton University), Shan Muthukrishnan (Rutgers University) and Jennifer Rexford (Princeton University).

Contents

Abstract iii

Acknowledgements iv

List of Tables vii

List of Figures viii

1 Introduction 1

2 Related Work 5

2.1 Frequent and Top-k Item Detection 5

2.2 Distributed Detection 5

3 Monitoring Model 7

3.1 Continuous Distributed Monitoring Model 7

3.2 Selective Edge Monitoring 7

4 Heavy-Hitter Algorithms 9

4.1 CMY Upper Bound 9

4.2 Adaptive Local Threshold 10

4.3 Probabilistic Reporting Approach 12

4.3.1 The Basic Approach 13

4.3.2 Adding Spatial Locality 14

5 Implementation **15**

5.1 P4 Prototype 15

5.2 Memory Efficiency Considerations 16

6 Evaluation **17**

6.1 Experimental Setup 17

6.2 Error-Free Approaches Evaluation 19

6.2.1 Sensitivity to Site Affinity 20

6.2.2 Sensitivity to Threshold 21

6.2.3 Sensitivity to Number of Sites 21

6.2.4 Sensitivity to Traffic Changes 22

6.3 Probabilistic Reporting Approaches Evaluation 23

6.3.1 Sensitivity to Number of Switches 23

6.3.2 Sensitivity to L 24

6.3.3 Sensitivity to Approximation Factor 25

6.3.4 Sensitivity to Communication Factor 27

6.3.5 Comparing to ALT Approach 28

6.4 Evaluation Conclusion 29

7 Ongoing Work **30**

7.1 Memory-Efficient Heavy Hitters 30

7.2 Learn Edge Monitoring Switches 30

7.3 Heavy Distinct Counts 31

8 Conclusion **33**

References **35**

List of Tables

4.1 Notation 13

6.1 Threshold Message Reduction Precentage Table 19

6.2 Communication Reduction Table 20

List of Figures

1.1 Sampling Graph 2

3.1 CDM Model Graph 8

6.1 Affinity Graph 18

6.2 Threshold Graph 19

6.3 Sites Graph 20

6.4 Alpha Graph 22

6.5 PR: Number of Sites to Accuracy Graph 23

6.6 PR: Number of Sites to Cost Graph 24

6.7 PR: L to Accuracy Graph 25

6.8 PR: L to Cost Graph 25

6.9 PR: Epsilon to Accuracy Graph 26

6.10 PR: Epsilon to Cost Graph 26

6.11 PR: c to Cost Graph 27

6.12 PR: c to Precision Graph 27

6.13 PR: Number of Sites to Communication Cost Graph 28

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 sources with high precision in order to detect and mitigate DDoS attacks or TCP incast [3] in real time. In traditional networks, this heavy-hitter detection relies on analyzing packet samples or flow logs [4, 5]. Networks that forward high traffic volumes often resort to sampling $\frac{1}{n}$ packets, where n is operator-defined based on the needs of the specific network. However, sampling can result in substantially reducing accuracy on small time scales [19], even when traffic volumes are high. In Figure 1.1, we show the impact sampling has on accuracy while performing heavy-hitter detection on a link between two major ISPs [11] processing approximately 8,092 Mbps of traffic. The precision is quite low 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.

Programmable switches open up new possibilities for aggregating traffic statistics and identifying large flows directly in the data plane [16, 17, 24, 27]. These solutions use approximate data structures, which bound memory and processing overhead in

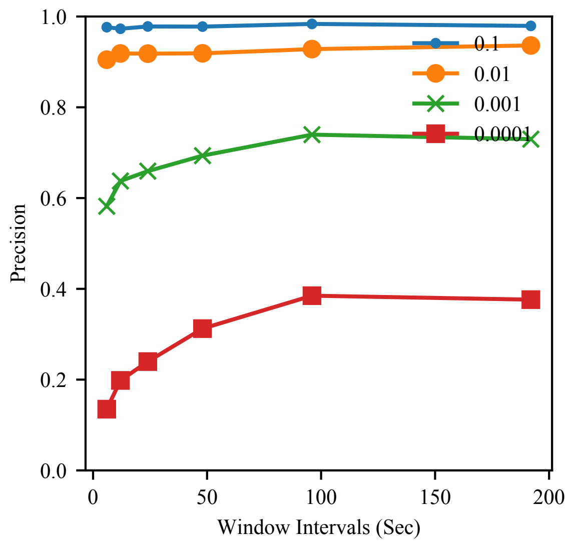

Figure 1.1: This graph shows the precision for detecting heavy hitters between two major ISPs [11] with different monitoring intervals. Precision quickly diminishes and worsens as the monitoring interval decreases.

exchange for some loss in accuracy, in order to deal with the limited resources available on the switches.

While prior work has focused on detecting heavy hitters at a single switch, network operators often need to track the *network-wide* heavy hitters. For example, port scanners [14] 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 could easily fall "under the radar" at multiple locations but still have sizable total volume. To accurately detect heavy hitters from a network-wide perspective we must introduce some *communication* between the nodes that could count a portion of a given flow. Since we must communicate for *each flow* that we wish to monitor, keeping the communication overhead low is important for scaling to a large number of flows or a large number of different heavy-hitter queries.

To quantify the amount of communication required to achieve accurate results, we frame detecting network-wide heavy hitters as an instance of the continuous distributed monitoring (CDM) problem [6]. Solutions [6, 26] to this problem demonstrate theoretical upper and lower bounds on the communication overhead by relaxing the accuracy of the results. However in the worst case, these solutions can actually degrade to $\frac{1}{n}$-sampling which leads to poor accuracy. Additionally, the abstract problem itself also fails to account for many properties of real networks, such as spatial locality, which we can exploit to further reduce the communication overhead while still achieving accurate results.

To perform network-wide heavy-hitter detection for real-time telemetry that achieves high accuracy with low communication overhead, we augment the CDM model:

Selective Edge Monitoring We augment the CDM problem to better model the network-wide heavy hitter detection problem based on the structure and locality properties of modern networks [21, 22]. While CDM treats all nodes in a network as being equally likely to observe the packets of a given flow, we note that only a subset of the entire network will actually observe the packets of a given flow. We enhance the CDM model to account for these observations.

Based on spatial locality of network traffic, we design two heavy-hitter detection algorithms:

Adaptive Local Threshold The first one is an error-free algorithm which we call Adaptive Local Threshold (**ALT**) approach. Inspired by prior work on distributed rate limiting [20], 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 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 future communication overhead.

Probabilistic Reporting Unlike **ALT**, Probabilistical Reporting (**PR**) does not need the global coordination. **PR** uses probabilistic reporting to detect network-wide heavy hitters with both high accuracy and low communication. Each flow's local threshold is set the same across all switches, but local thresholds for different flows may be different. Each switch probabilistically reports signals to the controller when the counts of any flow are greater than the local threshold. If the number of signals of any key is greater than a fixed threshold, the controller idenitifies the key as a heavy hitter.

Our main contribution of the paper is:

- We augment the CDM model by factoring the spatial locality of the network traffic into its decision.

- We design Adaptive Local Threshold approach which achieves perfect accuracy and reduces the communication cost comparing to the best known error-free approach in a small network.

- We design Probabilistic Reporting approach which balances accuracy and communication overhead in a large network.

- We prototype our solutions using the P4 [2] language. Experiments with ISP backbone traces [11] show that our methods substantially reduce the number of messages exchanged between the switches and the centralized controller while maintaining high precision and recall rate.

2. Related Work

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

2.1 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 [9] required to calculate these statistics. Several systems [12, 16, 17, 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.

2.2 Distributed Detection

Jain *et al.* [13] 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 [20] 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, 7, 8, 15]. 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.

3. Monitoring Model

3.1 Continuous Distributed Monitoring Model

Detecting network-wide heavy hitters is an instance of the continuous distributed monitoring (CDM) problem [6]. As shown in Figure 3.1, each of N sites sees a stream of observations (packets), and each 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 sites and the coordinator, while continuously computing the function in real time. For network-wide heavy hitters, we want to determine which flows exceed a global threshold (T) for a given statistic of interest. The best known solutions to this problem rely on setting per-site thresholds (t) and alerting the coordinator after the local threshold has been exceeded. After receiving a number of reports from the sites, the coordinator determines that the global threshold has been exceeded.

3.2 Selective Edge Monitoring

CDM 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. For

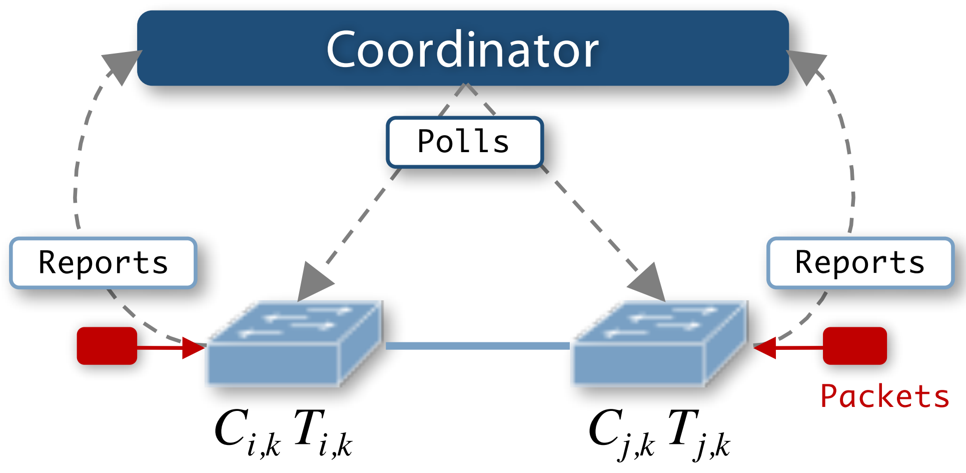

Figure 3.1: CDM Model. Each switch stores a count ($C_{i,k}$) and a local threshold ($t_{i,k}$) per key. Indices (i, k) refer to switch i and key k, respectively. Unless otherwise specified, k is the standard five-tuple.

example, in a datacenter, the traffic sent to a tenant's VMs can only enter VMs through a finite list of leaf switches, because cloud providers typically place a tenants VMs under the same leaf or spine switch to reduce bandwidth overhead for cross-VM communication [23]. 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. Based on this observation, we introduce a new symbol L_k to augment the CDM model, which refers to the number of monitoring switches for the flow k. Using this augmented model, we design two algorithms to further reduce the communication overhead.

4. Heavy-Hitter Algorithms

Rather than focusing on detecting heavy hitters with high memory efficiency, our approaches focus on maintaining high accuracy and at the same time reducing the overall *communication overhead* between the switches and the coordinator. In this chapter, we will first discuss error-free approaches and then focus on Probabilistic Reporting approaches that further reduce the communication cost by sacrificing the accuracy.

4.1 CMY Upper Bound

Previous work proved an upper bound $O(N \log \frac{T}{N})$ on the communication overhead between the N sites and a single coordinator [7, 8] which we will call the **CMY**

Algorithm 1: The CMY Algorithm: Controller

```
Input: $N$ switches, Global Threshold $T$, Count $C_{i,k}$, Round $R_k$
Output: Heavy Hitter Set (H), Local Threshold $t_{i,k}$
Func HandleReport($k$):
    $ReportedC_k \leftarrow ReportedC_k + 1$
    if $ReportedC_k \geq N$
        if $R_k \geq \lceil \log(\frac{T}{N}) \rceil$
            $H \leftarrow H \cup \{k\}$
        else
            SetLocalThresholds($\lfloor 2^{-i} \log(\frac{T}{N}) \rfloor, k$)
Func SetLocalThresholds($lt, k$):
    foreach $i \in N$ do
        $t_{i,k} \leftarrow lt$
```

(Cormode–Muthukrishnan–Yi) upper bound. **CMY** is the best known error-free approach we found for saving the communication cost which can be applied in PISA switches [7]. The algorithm proceeds over $\lceil \log \frac{T}{N} \rceil$ rounds. In the ith round, each switch sends a signal to the controller once its local count reaches $\lfloor 2^{-i} \log \frac{T}{N} \rfloor$. The controller will wait until it receives N signals and then advance to the next round. Once the controller receives N signals in the last round for a key k, k is identified as a heavy hitter. The algorithm is shown in Algorithm 1. This approach achieves the upper bound $O(N \log \frac{T}{N})$ of communication cost which we will refer as **CMY** upper bound [7]. As mentioned eariler, this algorithm ignores the distribution of keys. Our work shows the improvement upon this upper bound by utilizing spatial locality of network traffic.

4.2 Adaptive Local Threshold

ALT 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. 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 3.1. 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 2.

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), the coordinator *polls* all of the switches whose local thresholds are greater than zero to learn their current counts and produces 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 $\sum_i C_{i,k}$.

Algorithm 2: Adaptive Local Thresholds: Controller

```
Input: $N$ switches, Global Threshold $T$, 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 $(k) \geq T(k)$
        if GlobalPoll$(k) \geq T$
            $H \leftarrow H \cup \{k\}$
        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
        $frac \leftarrow \frac{(1-\alpha) \times EWMA_{i,k} + \alpha \times ReportedC_{i,k}}{\sum_{j=1}^{N} (1-\alpha) \times EWMA_{j,k} + \alpha \times ReportedC_{j,k}}$
        $t_{i,k} \leftarrow frac \times (T - \sum_{j=1}^{N} ReportedC_{j,k}) + ReportedC_{i,k}$

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

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.*, $t_{i,k} = \frac{T}{N}$, and is then recomputed by the coordinator based on subsequent reports. Algorithm 2 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 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.

4.3 Probabilistic Reporting Approach

In order to achieve the perfect accuracy, error-free approaches have to globally poll counts from each switch or reset local thresholds of all switches when certain conditions are met. However, when the number of monitoring switches increases, the cost of global polling increases sharply and negates the effect of adpatively tuning local thresholds. The probabilistic reporting approaches, which we focus on in this section, eliminate the need of global polling and lower the communication cost by probabilistically reporting signals to the controller. In this section, we first discuss the basic Probabilistic Reporting algorithm proposed by Cormode [6] and then factor the number of monitoring switches L_k into **PR**'s decision. Unless otherwise specified, we always use Probabilistic Reporting or **PR** to refer our augmented Probabilistic

Symbol	Meaning
N	Total sites in the network
T	Global threshold for flow k
$t_{i,k}$	Local threshold at site i for flow k
M	Number of reports controller expects to declare a heavy hitter
L_k	Number of monitoring switches for flow k
ϵ	Approximation factor
c	Communication factor
rp_k	Local reporting probability for flow k

Table 4.1: Notation

Algorithm 3: Basic Probabilistic Reporting Approach

```
Input: Count $C_{i,k}$
Output: Heavy Hitter Set (H)
$M \leftarrow \frac{c}{\epsilon}$; $t_{i,k} \leftarrow \frac{\epsilon * T}{N}$; $rp_k \leftarrow \frac{1}{N}$
Func Controller: HandleReport($k$):
    $ReportedC_k \leftarrow ReportedC_k + 1$
    if $ReportedC_k \geq M$
        $H \leftarrow H \cup \{k\}$
Func Switch i: RecvPkt($k$):
    if $Count_{i,k} \geq t_{i,k}$
        $Count_{i,l} \leftarrow 0$
        With Probabliity $rp_k$, Report ($k$)
```

Reporting approach, rather than the basic Probabilistic Reporting approach proposed by Cormode. The notation is shown in Table 4.1.

4.3.1 The Basic Approach

There are N switches in the network and a global threshold T. Each switch i maintains a local threshold $t_{i,k}$ and a counter per flow. Once the number of packets for a flow is equal to the local threshold in a switch, it resets the counter and sends a signal to the controller with probability rp_k. When the controller receives M signals for a flow, the

Algorithm 4: Augmented Probabilistic Reporting Approach

```
Input: Count $C_{i,k}$
Output: Heavy Hitter Set (H)
$M \leftarrow \frac{c}{\epsilon}$; $t_{i,k} \leftarrow \frac{\epsilon * T}{L_k}$; $rp_k \leftarrow \frac{1}{L_k}$
Func Controller: HandleReport($k$):
    $ReportedC_k \leftarrow ReportedC_k + 1$
    if $ReportedC_k \geq M$
        $H \leftarrow H \cup \{k\}$
Func Switch i: RecvPkt($k$):
    if $Count_{i,k} \geq t_{i,k}$
        $Count_{i,l} \leftarrow 0$
        With Probabliity $rp_k$, Report ($k$)
```

flow is identified as a heavy hitter. The algorithm is shown in Alg.3. This method reduces the communication cost by probabilitistically reporting while sacrificing the accuracy.

4.3.2 Adding Spatial Locality

However in the worst case, these solutions can actually degrade to $\frac{1}{N}$ sampling which leads to poor accuracy. For example, when $\epsilon = 0.1$, $T = 1000$ and $N = 100$, the local threshold becomes 1. This means for every packet, a switch may report a signal with probability = 0.01. The basic Probability Reporting approach does not adapt to natural differences in the portions of the traffic that enter (or leave) the network at different locations. In practice, a flow from a single source IP address typically enters the network at a limited number of sites. Based on this spatial locality of network traffic, we introduce a new parameter L_k. L_k refers to the number of monitoring switches that observes flow k. Reporting rate in our approach changes to $\frac{1}{L_k}$, rather than $\frac{1}{N}$ and the local threshold is now $\frac{\epsilon * T}{L_k}$. The algorithm is shown in Alg. 4. Comparing to the basic approach, our approach has higher accuracy and lower communication cost, shown in Chapter 6.

5. Implementation

5.1 P4 Prototype

Our prototype of data-plane algorithms 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 [18]. 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.

5.2 Memory Efficiency Considerations

Switches can maintain the per-key state (local counts and thresholds) using a hash-index array. 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.

While maintaining per-key state is expensive from a memory perspective, Chapter 6 shows that, in practice, we can store per-key state for a realistic query, based on real-world traffic traces. Our system identifies heavy hitters on a sliding time window (W), at the conclusion of which the counters for each key are reset. Only counting flows that appear on the window W reduces the memory for storing the per-key state. However, nothing about our approaches prevents us from employing space-saving algorithms and data structures [9, 24] if the memory constraints were prohibitive or we wanted to use much longer time windows. In Section 7, we will discuss how compact data structures can actually enhance our system beyond the base algorithm.

6. **Evaluation**

In this section, we quantify the reduction in the communication overhead using our algorithms. We first describe the experimental setup in Section 6.1 and then compare the performance of **ALT** to the **CMY** bound described by Cormode et al. [6, 7, 8] and the performance of our **PR** to the basic one in Section 6.2 and Section 6.3 respectively. We also quantify how sensitive the performance of **ALT** and **PR** is to various experimental parameters. For communication overhead, our evaluation shows that **ALT** improves upon the **CMY** upper bound [6] by up to 70% and our PR improves by up to 60% compared to the basic approach. We omit an explicit experiment to evaluate accuracy for our ALT approach because it achieves 100% precision and recall.

6.1 Experimental Setup

To quantify the performance of our approaches, we used CAIDA's anonymized Internet traces from 2016 [11]. 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.

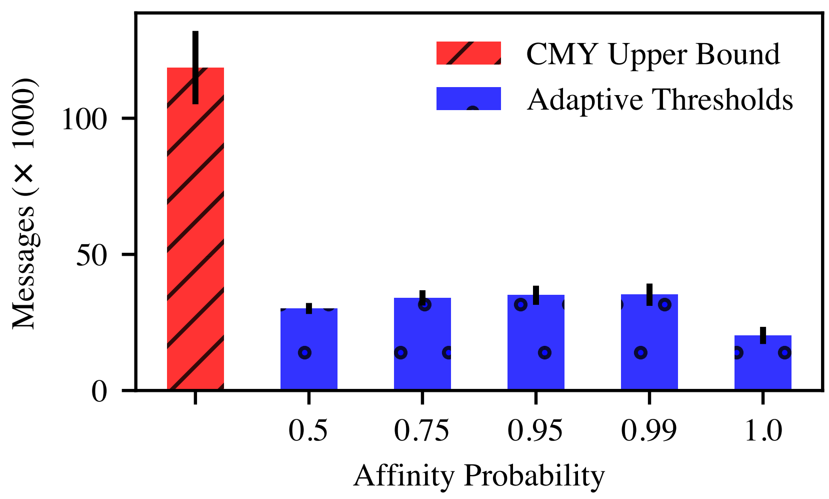

Figure 6.1: Communication overhead is reduced even when the sources' affinity for a preferred ingress switch is low.

We simulate a one-big-switch network consisting of N edge nodes (switches). The one-big-switch network is an abstraction that hides internal topology details and focuses on edge 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 $\frac{(1-p)}{(l-1)}$ 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) during a rolling time window (W). Unless otherwise specified, when evaluating error-free algorithms, we set $N = 4$, $l = 2$, $p = 0.95$, and $T = 600$. 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. When evaluating our **PR** algorithm, we set $N = 100$, $L = 2$, $c = 0.95$,

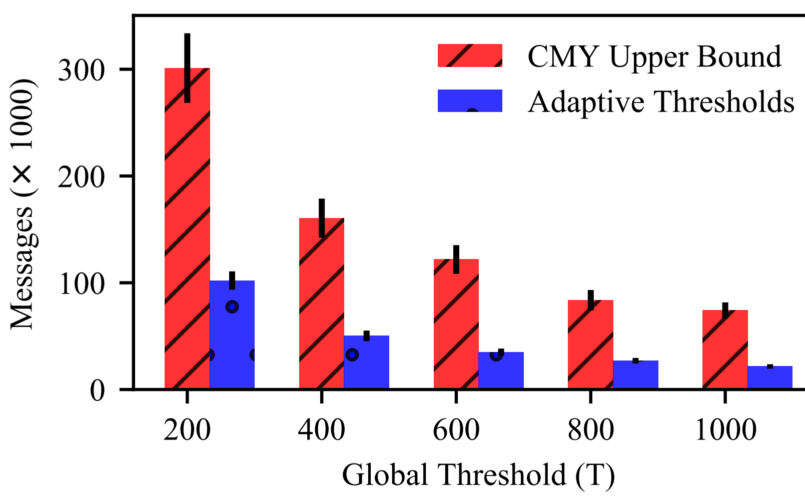

Figure 6.2: As the global threshold increases, the fraction of overall traffic that constitute heavy hitters decreases, reducing the communication overhead.

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

Table 6.1: Communication reduction over the **CMY** upper bound is not affected as the threshold increases.

$\epsilon = 0.1$ and $T = 1000$. In our experiment, we omit the symbol k in the L_k and set all L_k to the same number L and assume L is unchanged and known to the algorithm in our experiement. One of our ongoing work is to design an algorithm to learn the number of monitoring switches per flow at run time. In the future, when getting the network trace from multiple switches, we will evaluate how the learning algorithm improves our **PR** algorithm.

6.2 Error-Free Approaches Evaluation

We now use these traces to demonstrate how **ALT** reduces the communication overhead on continuous distributed monitoring for detecting heavy hitters. We compare the performance of **ALT** to the **CMY** upper bound [6]. **CMY** proceeds in a fixed sequence of rounds where each round reduces the per-site threshold exponentially

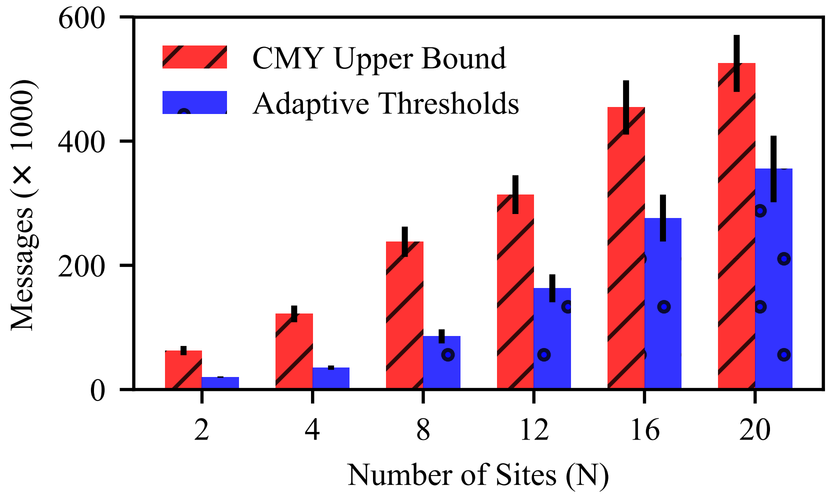

Figure 6.3: As the number of sites increases, the communication overhead increases due to the additional nodes communicating with the coordinator.

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

Table 6.2: Communication reduction over the **CMY** upper bound lessens as the number of sites increases.

until it reaches 1. We quantify the communication overhead in terms of the 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), number of sites (N), and smoothing factor (α).

6.2.1 Sensitivity to Site Affinity

In this experiment, we compare the performance of **ALT** to the **CMY** upper bound while varying the affinity for a given ingress switch. Figure 6.1 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.

6.2.2 Sensitivity to Threshold

In this experiment, we compare the performance of **ALT** for different threshold (T) values with the **CMY** upper bound. Figure 6.2 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 **ALT** compared to the **CMY** bound. Table 6.1 shows that our solution incurs 70% less communication overhead for $T = 1000$, which corresponds to a top-k=700 for this data set and query.

6.2.3 Sensitivity to Number of Sites

In this experiment, we compare the performance of **ALT** with the **CMY** upper bound as the number of sites (N) increases. Figure 6.3 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

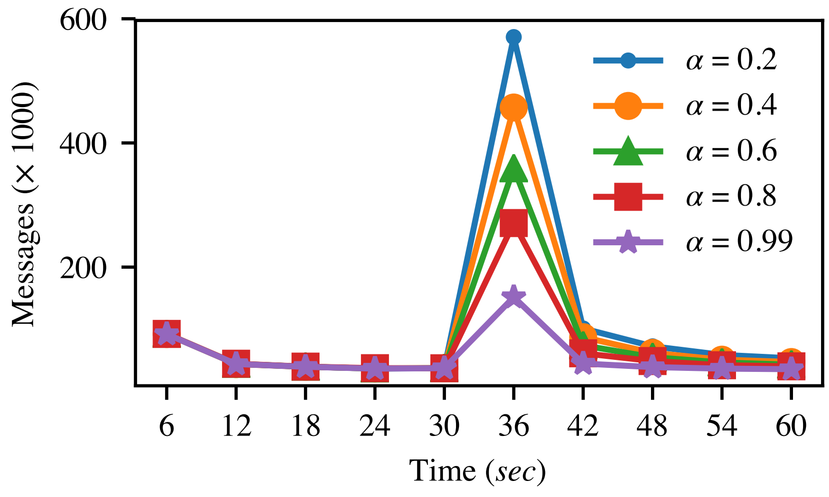

Figure 6.4: As alpha increases, the algorithm is able to more quickly adapt to changes in the traffic distribution which results in lower communication overhead.

globally poll all sites. However, Table 6.2 shows that our solution still reduces the communication overhead by 30–70% for up to $N = 20$.

6.2.4 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 6.4, we examine how well **ALT** 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.

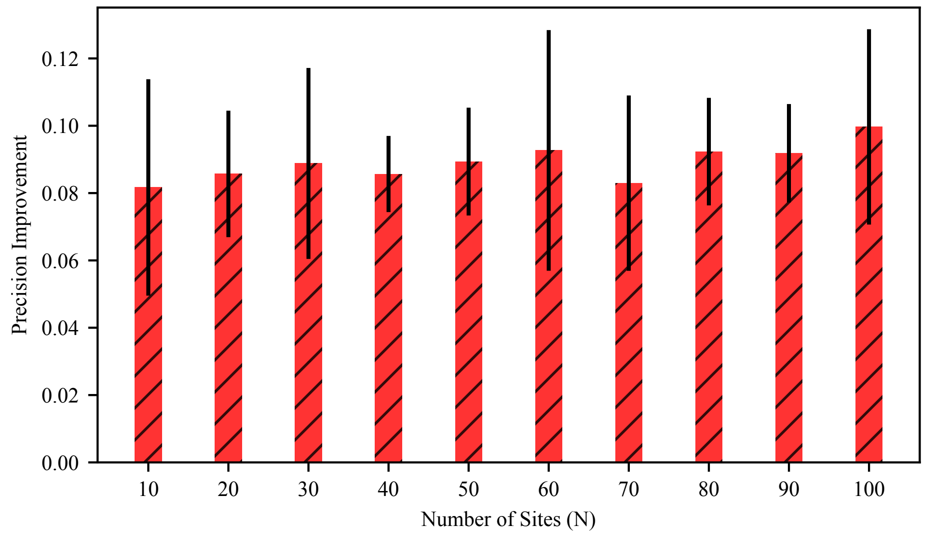

Figure 6.5: As number of switches increases, the precision of our **PR** still outperforms the basic **PR** algorithm by at least eight percent.

6.3 Probabilistic Reporting Approaches Evaluation

Similar to the error-free approaches evaluation, we quantify the communication overhead of two probabilistic reporting approaches in terms of the median number of communication overhead per window interval. Since both approaches cannot achieve perfect accuracy, we also measure and compare recall and precision of two algorithms. To demonstrate the sensitivity of the communication reduction and accuracy lost with respect to various parameters, we ran experiments varying one of four key parameters: number of monitoring switches (L), number of sites (N), the approximation factor (ϵ) and the communication factor (c).

6.3.1 Sensitivity to Number of Switches

In this experiment, we compare the performance of our **PR** with the basic **PR** as the number of sites (N) increases. Figure 6.6 shows how the performance, quantified

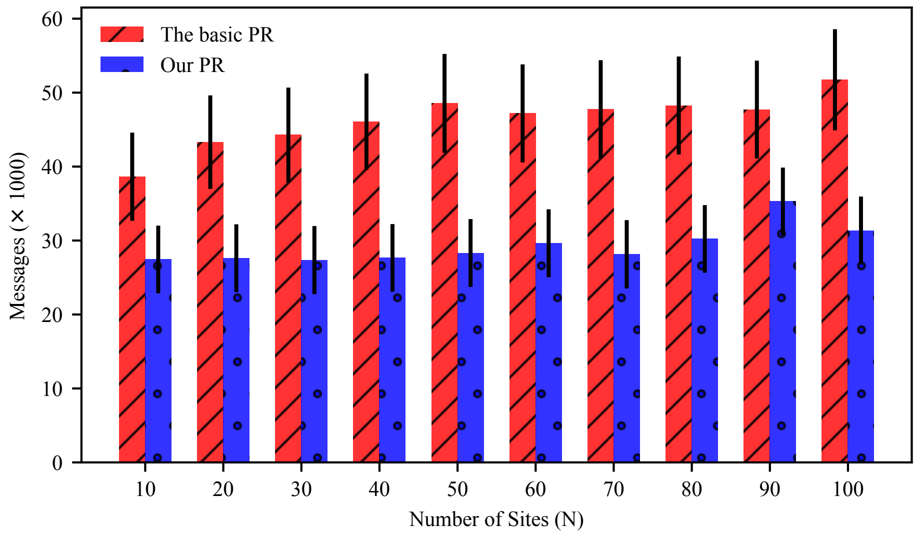

Figure 6.6: Regardless the number of switches increases, the communication cost of our **PR** outforms the basic **PR**.

as the total number of messages sent over the entire experiment duration, varies as we increase the number of sites. Unlike **ALT**, we observe that as the number of sites increases, our **PR**'s communication overhead stays almost the same. This observation demonstrates our **PR** scales well. Our **PR** solution reduces the communication overhead by 30–60% for up to $N = 100$, comparing to the basic **PR**. Besides saving the communication overhead, our **PR** also improves the precision by at least 8 %, as shown in Figure 6.5. The recall of two algorithms are almost the same.

6.3.2 Sensitivity to L

In this experiment, we evaluate the performance of our **PR** as the number of monitoring switches L increases. Figure 6.8 shows how the performance of the algoirthm changes as we increase L. We observe that as the number of sites increases, our communication overhead increases slightly. Figures 6.7 illustrates that the precision of our **PR** decrease slightly as L increases. The change of recall is similar as the change of precision.

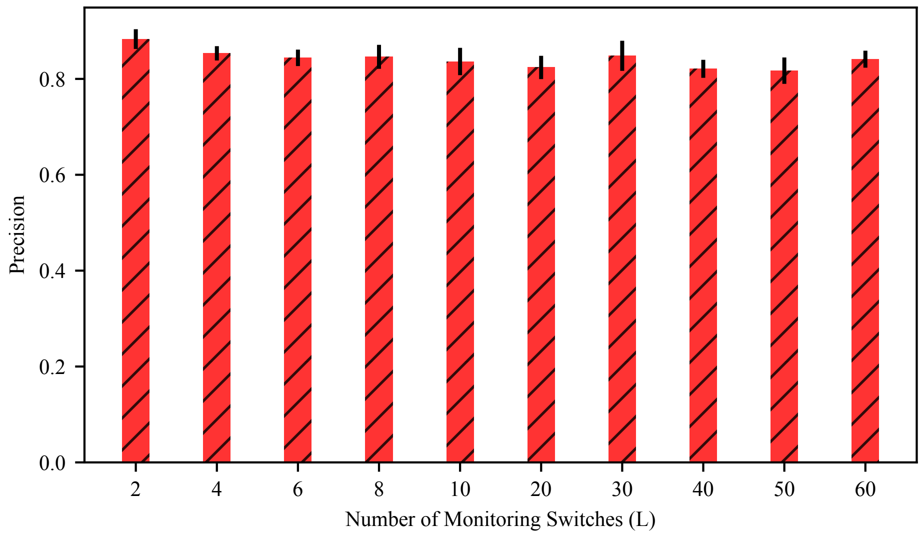

Figure 6.7: As number of monitoring switches increases, Our **PR** stil maintains a good precision rate.

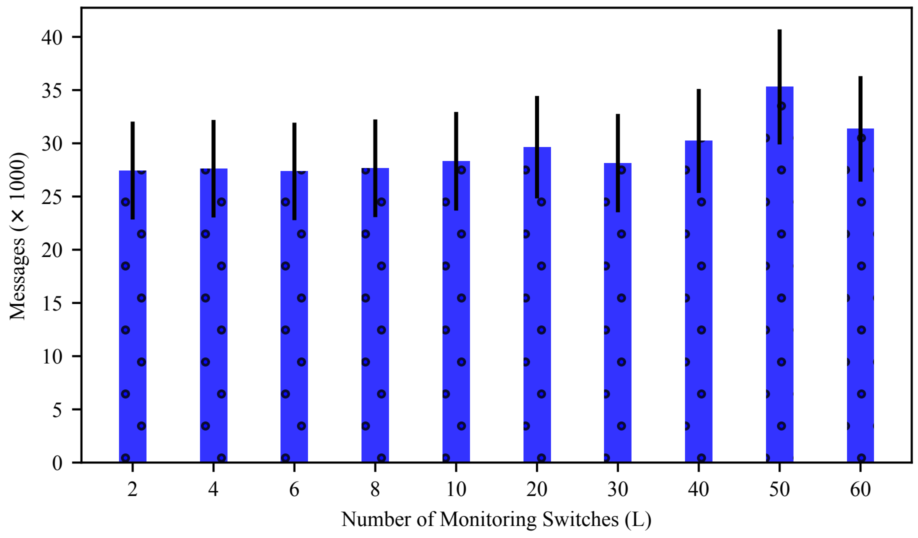

Figure 6.8: As number of monitoring switches increases, the communication cost of our **PR** keeps almost the same.

6.3.3 Sensitivity to Approximation Factor

In this experiment, we show the performance of our **PR** as epsilon ϵ increases, as shown in Figure 6.10. Again, ϵ is the approximation factor that determines the

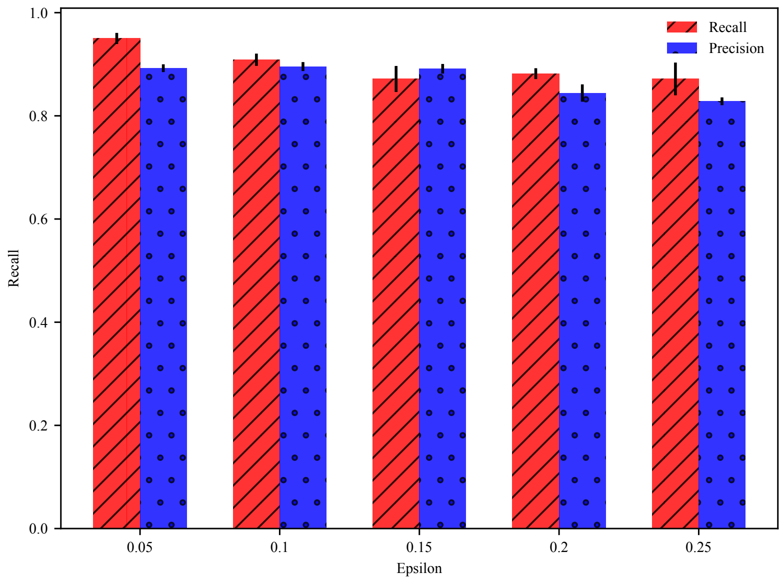

Figure 6.9: As epsilon ϵ increases, the median of recall of our **PR** decreases from 0.94 to 0.87 and the median of precision decreases from 0.89 to 0.82.

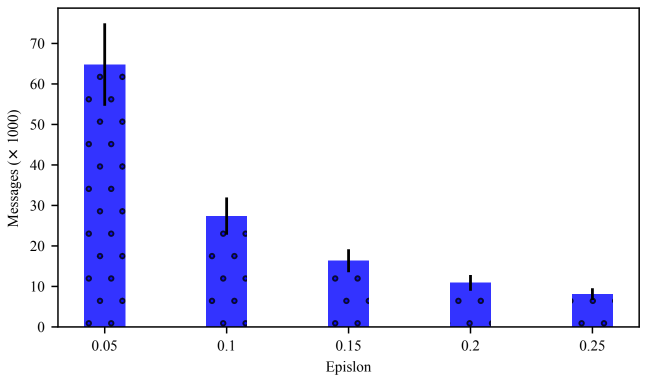

Figure 6.10: As epsilon ϵ increases, the median of communication cost of our **PR** decreases from 64000 to 8000 messages.

number of signals ($\frac{c}{\epsilon}$) required for the controller to decide a flow is a heavy hitter and the local threshold $\frac{\epsilon * T}{L}$ in the switch per flow.

As ϵ increases, the communication overhead decreases signifcantly from 64000 to 8000 messages. As shown in Figure 6.9, the recall of our **PR** approach decreases from 0.94 to 0.87 as ϵ increases and the precision decreases from 0.89 to 0.82.

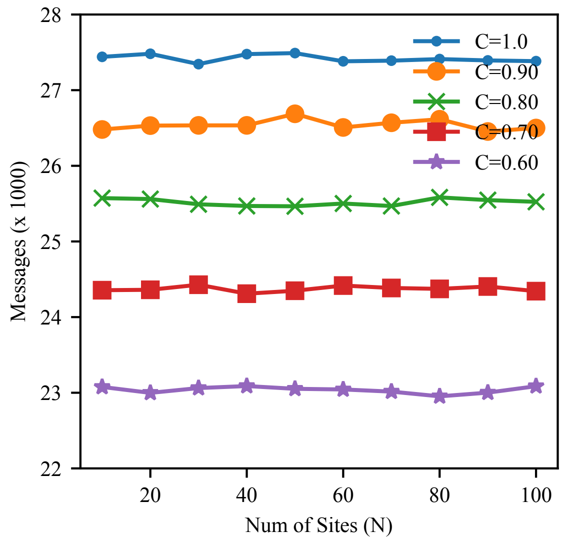

Figure 6.11: As the communication factor c increases, the median of communication cost of our **PR** gradually increases, regardless of the number of switches.

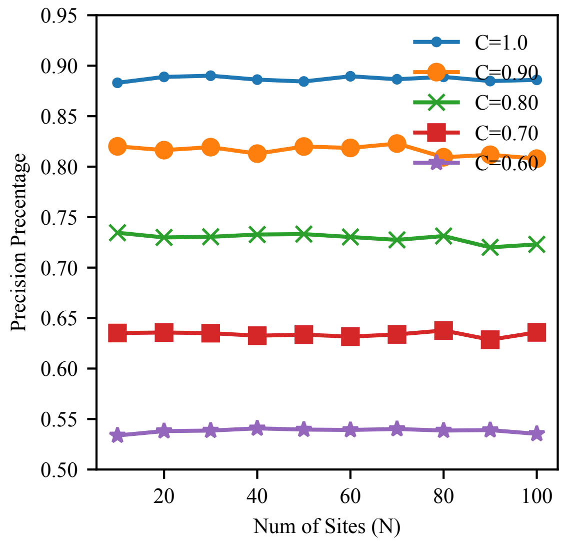

Figure 6.12: As the communication factor c increases, the median of precision of our **PR** gradually increases, regardless of the number of switches.

6.3.4 Sensitivity to Communication Factor

The communicaition factor c affects the number of signals $(\frac{c}{\epsilon})$ required for the controller to decide a flow is a heavy hitter. We measure the impact of c to the communication overhead and precision of our **PR** approach as the number of switches

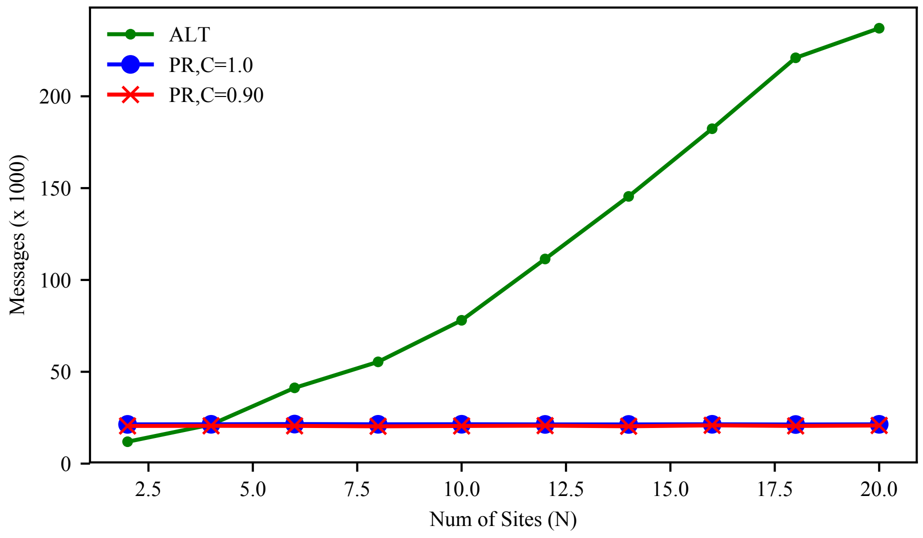

Figure 6.13: As number of switches increases, the overhead of our **PR** is significantly lower than the overhead of **ALT**.

increases in Fig 6.11 and Fig 6.12 respectively. The precision gradually decreases as c decreases because the number of false positves increases. The communication overhead also decreases as c decreases because the threshold $\frac{c}{\epsilon}$ required for the controller to detect a heavy hitter decreases.

6.3.5 Comparing to ALT Approach

We compare the performance of **ALT** to our **PR** as the number of switches increases. We find out that as the number of switches increases, the performance of **PR** is much better than the performance of **ALT**, as shown in Fig. 6.13. This figure shows that in a large network, **PR** is a better algorithm to deploy, because it balances accuracy and communication overhead very well. According to Fig. 6.12, the precision of **PR** is not perfect and when the number of switches is small, the overhead of **ALT** is similar as the overhead of **PR**. Therefore, in a small network, **ALT** is better.

6.4 Evaluation Conclusion

As our evaluation shows, **ALT** saves the communication overhead, comparing to the **CMY** bound and our **PR** algorithm reduces the communication cost and improves the accuracy of the heavy-hitter detection, comparing to the basic **PR** algorithm. Also, we find out that in a small network, **ALT** is a better option than **PR** to deploy because **ALT** achieves perfect accuracy and the cost of **ALT** is comparable to the cost of **PR**. In a large network, **PR** is a better option because **PR** balances accuracy and the cost by tuning the communication factor c.

7. Ongoing Work

While our evaluation demonstrated that our algorithms substantially reduce the communication overhead for detecting heavy hitters, our approaches can be improved in at least three ways: (1) by reducing the amount of state switches must store in the data plane, (2) learning edge monitoring switches per flow and (3) supporting distinct counts.

7.1 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 could maintain a count-min sketch [9] 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.

7.2 Learn Edge Monitoring Switches

Learning the set of edge monitoring switches per flow at run time is crucial to deploy our **PR** algorithm in the real world. The set of monitoring switches of a flow changes over the time. For example, some edge switches may fail, causing the traffic enter the network in a fewer ingress points. In this case, when the controller finds the failure happens, it should adjust L_k for the affected flows and send signals to all edge switches that monitor those flows. Misconfiuring L_k causes us to pay additional communication overhead. In the worst case, our **PR** algorithm may downgrade to the basic **PR** approach. One of our ongoing work is to design an algorithm to learn the edge monitoring switches per flow to better deploy our **PR** algorithm.

7.3 Heavy Distinct Counts

We have described count-based heavy hitters in Section 4.2. However, network operators often need to identify the heavy distinct counts. For example, operators try to identify the potential victims of a DDoS attack once vicitims receive DNS packets from more than T distinct senders. Counting is not effective when computing *distinct* counts.

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.

Fortunately, we can leverage a cardinality estimation algorithm known as HyperLogLog [10]. Hyperloglog can be performed at distributed sites and later merged without any loss of accuracy. Also, it is memory-efficient, so it can be implemented in switches. 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 Barefoot Tofino switch.

8. Conclusion

Detecting heavy hitters is an indispensable tool in managing and defending modern networks. We designed two efficient algorithms 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 (**ALT**) or the number of monitoring switches (**PR**), we can reduce the communication overhead required to detect network-wide heavy hitters while maintaining high accuracy.

As richer network traces become available from multi-switch networks, we can further explore the efficacy of our methods 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 communication overhead. We could improve **ALT** approach by starting with local thresholds learned from historical training data, given the availability of such data. Also, we could evaluate how the learning monitoring switches algroithm would improve the performance of our **PR** approach with additional data.

In conclusion, **ALT** approach achieves perfect accuracy and reduces the communication overhead in a small network comparing to the best-known error-free approaches. Our **PR** approach balances accuracy and communication overhead in a large network. In the real world, network operators can deploy these two algorithms to detect network-wide heavy hitters, depending on the scale of their networks, cost they like to pay and the accuracy they intend to achieve.

References

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

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

[3] Yanpei Chen, Rean Griffith, Junda Liu, Randy H. Katz, and Anthony D. Joseph. Understanding TCP incast throughput collapse in datacenter networks. In *ACM SIGCOMM Workshop on Research on Enterprise Networking*, pages 73–82, New York, NY, USA, 2009. ACM.

[4] Benoit Claise. Cisco systems Netflow services export version 9. RFC 3954, RFC Editor, October 2004.

[5] Benoit Claise. Specification of the IP flow information export (IPFIX) protocol for the exchange of IP traffic flow information. RFC 5101, RFC Editor, January 2008.

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

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

[8] Graham Cormode, S Muthukrishnan, Ke Yi, and Qin Zhang. Optimal sampling from distributed streams. In *ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 77–86. ACM, 2010.

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

[10] Philippe Flajolet and G. Nigel Martin. Probabilistic counting algorithms for database applications. *Journal of Computer and System Sciences*, 31(2):182–209, 1985.

[11] Center for Applied Internet Data Analysis. *The CAIDA UCSD Anonymized Internet Traces 2016*, 2018 (accessed November 1, 2017).

[12] Arpit Gupta, Rob Harrison, Ankita Pawar, Rüdiger Birkner, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven network telemetry. In *ACM SIGCOMM*. ACM, 2018.

[13] Ankur Jain, Joseph M Hellerstein, Sylvia Ratnasamy, and David Wetherall. A wakeup call for internet monitoring systems: The case for distributed triggers. In *HotNets-III*, 2004.

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

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

[16] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Flowradar: A better netflow for data centers. In *Usenix NSDI*, pages 311–324, 2016.

[17] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One sketch to rule them all: Rethinking network flow monitoring with UnivMon. In *ACM SIGCOMM*, pages 101–114. ACM, 2016.

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

[19] Peter Phaal and Sonia Panchen. *Packet Sampling Basics*, 2003 (accessed February 14, 2018).

[20] Barath Raghavan, Kashi Vishwanath, Sriram Ramabhadran, Kenneth Yocum, and Alex C. Snoeren. Cloud control with distributed rate limiting. In *ACM SIGCOMM*, pages 337–348, New York, NY, USA, 2007. ACM.

[21] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the social network's (datacenter) network. In *ACM SIGCOMM*, pages 123–137, New York, NY, USA, 2015. ACM.

[22] Brandon Schlinker, Hyojeong Kim, Timothy Cui, Ethan Katz-Bassett, Harsha V Madhyastha, Italo Cunha, James Quinn, Saif Hasan, Petr Lapukhov, and Hongyi Zeng. Engineering egress with edge fabric: Steering oceans of content to the world. In *ACM SIGCOMM*, pages 418–431. ACM, 2017.

[23] Muhammad Shahbaz, Lalith Suresh, Nick Feamster, Jen Rexford, Ori Rottenstreich, and Mukesh Hira. Elmo: Source-routed multicast for cloud services, 2018.

[24] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S Muthukrishnan, and Jennifer Rexford. Heavy-hitter detection entirely in the data plane. In *ACM SOSR*, pages 164–176. ACM, 2017.

[25] Cisco Systems. *Cisco Nexus 3600 NX-OS System Management Configuration Guide*, 2017 (accessed February 25, 2018).

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

[27] Minlan Yu, Lavanya Jose, and Rui Miao. Software defined traffic measurement with OpenSketch. In *Usenix NSDI*, volume 13, pages 29–42, 2013.