Computer Networks xxx (2010) xxx–xxx

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

CLAMP: Efficient class-based sampling for flexible flow monitoring

Mohit Saxena [a,1], Ramana Rao Kompella [b,*]

[a] Department of Computer Sciences, University of Wisconsin-Madison, 1210 West Dayton St, Madison, WI 53706, United States
[b] Department of Computer Sciences, Purdue University, 305 N. University St, West Lafayette, IN 47906, United States

ARTICLE INFO

Article history:
Received 10 July 2009
Received in revised form 23 February 2010
Accepted 28 February 2010
Available online xxxx
Responsible Editor J. Domingo-Pascual

Keywords:
Sampling
Bloom filter
Flow monitoring

ABSTRACT

With an increasing requirement to classify traffic and track security threats, newer flexible and efficient ways are needed for collecting traffic statistics and monitoring network flows. However, traditional solutions based on packet sampling do not provide the flexibility required for these applications. For example, operators are often interested in observing as many unique flows as possible; however, random packet sampling is inherently biased towards large flows. Operators may also be interested in increasing the fidelity of flow measurements for a certain class of flows; such flexibility is lacking in today's packet sampling frameworks. In this paper, we propose a novel architecture called CLAMP that provides an efficient framework to implement class-based sampling. At the heart of CLAMP is a novel data structure we propose called composite Bloom filter (CBF) that consists of a set of Bloom filters working together to encapsulate various class definitions. In particular, we show the flexibility and efficacy of CLAMP by implementing a simple two-class size-based sampling. We also consider different objectives such as maximizing flow coverage and improving the accuracy of certain class of flows. In comparison to previous approaches that implement simple size-based sampling, our architecture requires substantially lower amounts of memory (up to 80×) and achieves higher flow coverage (up to 8× more flows) under specific configurations.

1. Introduction

Flow monitoring is an essential ingredient of network management. Typical flow monitoring involves collection of flow records at various intermediate network boxes such as routers. These flow records can assist a network operator in various tasks such as billing and accounting [5], network capacity planning, traffic matrix estimation [16,5,9], tracking heavy hitters [7], and detecting the presence of adversarial traffic (*e.g.*, worms, DoS attacks).

While the basic task of flow monitoring appears simple, collecting flow records at high speeds under extremely resource-constrained environments is quite challenging. Particularly, memory and CPU resources in routers are often distributed among several critical functions such as route computation, forwarding, scheduling, protocol processing and so on. Thus, flow monitoring tasks have to contend themselves with only a small fraction of the overall pie. With such constrained resources and ever increasing link speeds, it is extremely difficult to record each and every packet on a given interface. In order to overcome this hurdle, routers record only a random subset of packets that they observe by *sampling* the packets that traverse the interface.

The rate at which packets are sampled typically depends on the resources available on the router. The three major router resources that the flow collection task needs to grapple with includes CPU, memory, and flow export bandwidth. Several sampling schemes exist to control the utilization of these resources. For example, Cisco's flow collection tool called NetFlow [4] implemented on routers

* Corresponding author. Tel.: +1 765 496 9369.
E-mail addresses: msaxena@cs.wisc.edu (M. Saxena), kompella@cs.purdue.edu (R.R. Kompella).
1 The work was performed when the author was at Purdue University.

1389-1286/$ - see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.comnet.2010.02.016

includes a simple stage of random packet sampling. It exports flow records computed on an unbiased sample of packets collected on each interface. These flow records are then used to estimate flow aggregate volumes and also feed into several other management tasks. For example, they are used to estimate the volumes of popular applications such as Web and Email, or volume of traffic going from one prefix to another prefix for traffic matrix estimation.

A major deficiency of uniform packet sampling in collecting flow records is its bias towards heavy-hitter flows, *i.e.*, flows that have a large number of packets. Given that Internet flow size distribution is heavy-tailed, a large majority of sampled packets typically belong to a few large flows. While such a bias does not affect volume estimation applications, it provides no flexibility to network operators to specify how to allocate their overall sampling budget among different classes of traffic. For example, an operator might want to specify that he is interested in collecting as many small-sized flows as possible to satisfy security applications such as tracking botnets, detecting portscans and so on. For such applications, packet sampling is exactly the wrong choice as it inherently fills up the sampling budget with a large number of packets from the so-called 'elephant' flows.

In general, we observe that while sampling budget is directly dictated by router constraints, the network operator should be able to specify how to use this sampling budget efficiently to satisfy monitoring objectives. The monitoring objectives themselves are context and location dependent, and as such, we desire a flexible architecture that can accommodate several ways of allocating the sampling budget among different types of flows. It is, however, neither feasible nor desirable to control the sampling on a per-flow basis due to two reasons. First, the number of flows is too large which means a lot of state needs to be maintained. Second, it causes tremendous administrative overhead to configure and manage the sampling rates.

Instead of fixing a per-flow sampling rate, it is beneficial to aggregate flows into classes and allocate specific sampling rates to these individual classes. Note that this is similar to how we achieve QoS in modern routers through aggregation into DiffServ classes. Indeed, newer versions of Cisco's NetFlow [3] allow setting input filters which allow specifying different filters for different classes of traffic. However, the definition of a class is based on an access control list and thus is based on the fields of the header. We cannot, for example, specify that small flows and large flows get different sampling rates to implement size-based sampling [11]—a serious limitation of this *static* class-based sampling.

In this paper, we propose an architecture, called CLAMP(short for CLass-based sAMPling), that provides network operators to define dynamic flow classes (*i.e.*, those that are not dependent only on static header fields) and perform efficient class-based sampling on these flow classes.Our architecture itself is quite general and is applicable in the more general context of runtime classification of flows. To illustrate the efficacy of our approach, however, we focus on implementing simple two-class size-based sampling in this architecture as a canonical example. Specifically, we focus on providing different sampling rates to two different classes of flows, elephants and mice, based on their flow sizes. Compared to prior research (*e.g.*, [11]), the simplicity of our architecture facilitates an implementation with significantly lesser amount of memory and higher accuracy.

Thus, the contributions of our paper are as follows:

- We describe the architecture of CLAMP that achieves dynamic class-based sampling with the help of a novel classification data structure called composite Bloom filter (CBF) to help network operators flexibly allocate different sampling budgets among different competing classes.
- We show how to perform simple two-class size-based sampling (with extension to multiple classes) using our architecture. We also consider how to specify different objectives such as increasing flow coverage and maximizing accuracy of a specific group of flows and so on.
- We present both theoretical as well as empirical analysis for the same. Our results using real backbone traces indicate that our architecture consumes up to $80\times$ smaller amount of memory and achieves larger flow coverage (up to $8\times$ more flows under specific configurations), as compared to existing solutions.

The rest of the paper is organized as follows. First, we provide background and related work in Section 2. We then describe the CLAMP architecture in Section 3 and implementation details and theoretical analysis of size-based sampling in Section 4. We outline evaluation details in Section 5.

2. Background and related work

The increasing importance of flow measurement as an essential ingredient in several network management tasks prompted router vendors such as Cisco and Juniper to support a simple flow measurement solution called NetFlow [4] in routers. The basic version of NetFlow observes each and every packet on a given router interface and checks to see if a flow record is already created for that given packet. If it is already created, the information from the packet is incorporated into the flow record. The basic form of NetFlow maintains flow statistics such as packet and byte counters and information about TCP flags, timestamps of the first and last packet among other information [4].

Basic NetFlow does not scale beyond a few flows and small link speeds and thus is not suitable for several core backbone links. Routers, therefore, support sampled NetFlow, a variant of NetFlow that works on packets that are sampled according to a configurable sampling rate (say 1 in 64). By randomly sampling packets, sampled NetFlow allows unbiased estimators to compute volume estimates for different types of traffic. Several flow monitoring solutions that exist in the literature are fundamentally based on this simple idea. For example, adaptive NetFlow [6] exploits a degree of freedom in NetFlow—the sampling rate—

to control resource usage in the presence of adversarial traffic mixes such as DoS attacks. Flow sampling [9] advocates the usage of flow sampling (using hash-based flow selection) instead of random packet sampling. FlowSlices [10] combines the two and advocates the use of different tuning knobs for different resources (memory, CPU)—packet sampling for controlling CPU usage and flow sampling to control memory. While solutions based on random packet sampling are relatively easy to implement, they are not necessarily the most effective form of sampling for specific applications. For example, the inherent bias of random packet sampling towards large flows is harmful if collecting as many flows as possible is the objective. More generally, they do not provide the necessary flexibility to operators to distinguish between different classes of traffic.

The requirement of flexibility in flow measurement has been noted in several prior research efforts. ProgME [15], for example, articulates the need for a programmable measurement architecture in which a network operator can specify the set of flows he is interested in monitoring. Using efficient binary decision diagrams, ProgME allows routers to encode network operators' intent directly into the routers. While this offers some flexibility, the requirement of ProgME to explicitly specify the 5-tuples associated with individual flows of interest makes it hard to specify dynamic flow definitions (such as 'small' and 'large' flows).

A seminal paper by Kumar and Xu [11], provides a way to perform size-dependent sampling using a sketch to estimate the flow size. By configuring the sampling rate to be inverse of the flow size, the relative error of flow size estimates remains similar irrespective of the sizes. Note that this is unlike random packet sampling in which flow size estimates for large flows exhibit much better accuracy as compared to smaller flows. Their main observation is that we can reduce the sampling rate for large flows and allocate more sampling budget towards smaller flows so that the accuracy of large flows only decreases by a little while small flows benefit significantly. Given they need flow size estimates, they advocate the use of an online sketch to obtain flow size estimates.

FlexSample [14] builds upon this basic idea and attempts to explicitly improve the flow coverage, as opposed to just accuracy of volume estimates. Both these solutions, however, use sketches or counting Bloom filters which are quite heavy-weight. Our architecture, described in next section (Section 3), generalizes this notion of flexibility by defining 'classes'. It also uses light-weight data structures to simplify the associated implementation overhead. For class-based sampling as a canonical example, it reduces the overall memory consumption significantly while allowing the ability to improve flow coverage or accuracy.

3. Design of CLAMP

Our goal is to design an architecture that provides flexibility in configuring different sampling rates for different types of flows. At a high-level, the design of CLAMP comprises of two basic components—a classification data structure and flow memory as shown in Fig. 1. The classification data structure identifies the class to which a packet belongs, which then is used to determine the sampling rate at which to sample the packet. If the packet is sampled (according to the sampling rate determined), then the flow memory is updated for the particular flow to which the packet belongs. Since the classification data structure needs to operate at high speeds, it needs to be small and simple, *i.e.*, it needs to perform very simple operations and must use very little high speed SRAM. On the other hand, the rate at which the flow memory needs to be updated can be controlled by the sampling rate, and thus, it can reside in slower off-chip DRAM memory.

In CLAMP, we use a novel light-weight classification data structure called composite Bloom filter (CBF). Unlike other prior approaches [11,13], which use lossy synopsis data structures maintaining large arrays of counters, CBF is composed of a few Bloom filters (BFs) working in tandem to identify the class to which an incoming packet belongs to. Since the CBF uses simple BFs, it is small enough to fit well in fast memory (SRAM) and can easily operate at link speeds such as OC-192 and OC-768. Generally speaking, the membership of a packet within a specific combination of BFs represents a class. Thus, upon each packet arrival, the packet's flow id (*e.g.*, the 5-tuple) is queried parallelly within each BF and the matching BF indices are obtained. The tuple of matching BFs (the class-membership bitmap shown in Fig. 1) is used to identify the class to which the packet belongs to. In the simplest case, each BF represents a unique class; thus, the number of classes equals number of BFs and the packet that matches a given BF is said to belong to the corresponding class. In more complex scenarios, class definitions include arbitrary combinations of BFs.

As shown in Fig. 1, the sampling rate corresponding to each packet is determined by first identifying the class to which the packet belongs. The network operators can choose the sampling rate for a class based on several criterion. For example, network operators may want to sample a given target fraction of packets for different classes subject to processing constraints. They may also want to increase the accuracy of volume estimates for certain types of flows. In addition, they can also choose to allocate flow and packet coverage over different flow classes in a 'fair' manner. We discuss a few such objectives in more detail in Section 4.2.

Once the sampling rate is determined, the packet is then probabilistically sampled based on this rate. If the packet is sampled and a flow record exists in the flow memory, then the flow record is updated with the contents of the packet (*e.g.*, packet counter is incremented, byte counter is incremented by the packet size, and special flags in the packet are noted). If not, a new flow record is created for this packet. We envision that the flow memory resides in the larger and less expensive memory (DRAM).[2] Both the flow memory as well as the CBF data structure is periodically reset every *epoch* when the flow records are reported to the collection center.

[2] In some cases, a limited amount of flow memory may reside in SRAM which is then flushed to the DRAM.

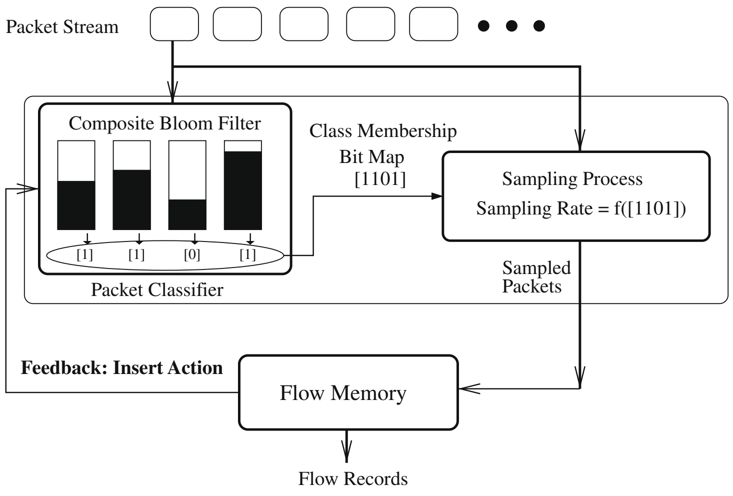


Fig. 1. Architecture of CLAMP.

The most important step is to decide which flows are inserted into a given BF. If classes are just based on pure packet headers, it is relatively straightforward to insert individual flows into the BFs. However, classes can be dependent on specific flow characteristics such as packet or byte counts. Such flow characteristics are not known *a priori* and must be determined on-the-fly. Even further, a flow may initially belong to a class X and then may transition into class Y at a later time. In order to account for such dynamics, we use the flow records in the flow memory to determine which class a given flow belongs, and when a flow transitions from one class to another. This is depicted as the feedback action in Fig. 1. Note that BFs do not allow deletion of flow records; thus, once a flow is entered in a given BF, it cannot be deleted from that BF. This constrains the class definitions to some extent.

Because records in the flow memory are based on sampled packets, they are subject to re-normalization errors. When flows are inserted or transitioned between different classes based on these sampled statistics, these re-normalization errors can affect the accuracy of the mapping between a flow and a class and thus may result in misclassification of certain flows. In addition, there could be misclassification due to the inherent false positives associated with BFs. To some extent, this is unavoidable due to the approximate nature of the data structure; the key is to ensure that such misclassification results do not significantly affect the overall flow monitoring objectives. While the framework itself is quite general, we focus on one specific example, namely size-based sampling, to illustrate the efficacy of CLAMP.

4. Size-based sampling using CLAMP

It is well known that sampling random packets for collecting flow statistics is prone to biases towards larger flows [11]. To alleviate this problem, an operator may like to, for example, set different sampling rates for 'mouse' and 'elephant' flows, where a flow is termed a mouse or an elephant depending on whether the number of packets in that flow is below or above a particular pre-defined threshold, *T*. By setting a higher sampling rate to mouse flows, we will be able to capture a larger number of unique flows, or in other words, increase flow coverage. This however, may result in a reduction of the accuracy of flow volume estimates for the elephant flows. Both of these might be perfectly valid objectives for a network operator; the choice of one over the other depends on the particular scenario. We consider both of these choices.

4.1. Configuring the CBF

In this section, we discuss how we can configure the CBF to implement different sampling rates for both the mouse and elephant classes. In this case, the CBF comprises of two Bloom filters naively, one for the mouse and the other for the elephant flows. However, since every flow starts off as a mouse flow as we do not know the flow size *a priori*, there is no explicit need for a mouse Bloom filter in the CBF.

The two basic operations on CBF are *find* and *insert*. For each packet, the five tuple corresponding to the source and destination IP addresses and ports along with the protocol

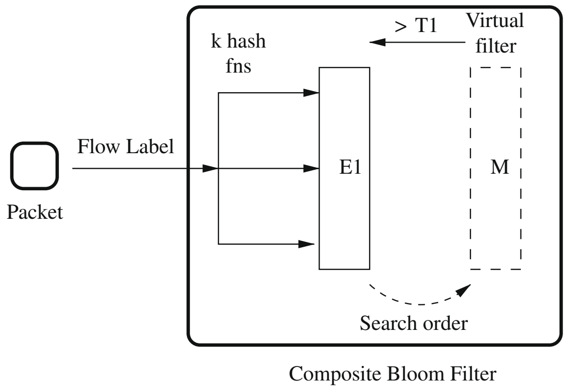


Fig. 2. CBF: finding the flow class to which a packet belongs.

field is used to uniquely identify a flow. As shown in Fig. 2, this flow five-tuple is hashed using k hash functions in parallel, and the hashes are queried in each Bloom Filter in parallel. The matched class with the highest priority is picked. In our case, if a flow belongs to the elephant class, then both the elephant and mouse BFs (if existed) will respond with a match, and since the priority of the elephant class is higher, the flow belongs to that class. More generally, all the BFs are arranged in priority of increasing flow sizes. Fig. 2 shows the case with three BFs for three classes—E_2, E_1 and M. E_2 and E_1 classes are distinguished with the threshold T_2 and E_1 and M by T_1. Since T_2 is greater than T_1, E_2 has higher priority than E_1, which in turn has higher priority than M.

Initially, all flows are entered in the mouse filter upon the arrival of the first packet. A new entry is created for a flow in the elephant filter, if the flow crosses the threshold to become an elephant. This particular operation can be seen as the feedback from the flow memory to the CBF in Fig. 1. Flow memory, however, is updated for every packet sampled for a flow.

4.2. Analysis

For our example of a two-class CBF, packets can be classified into either mouse or elephant packets, with some mouse packets misclassified as elephant packets and vice-versa. We use M_M and M_E to denote sampled mouse packets that are classified as mouse and elephant packets respectively (and hence sampled at mouse and elephant sampling rates). Similarly, we denote E_M and E_E as the elephant packets classified as mouse and elephant packets respectively.

If we used exact flow counters, M_E would be zero, since a mouse packet would never be misconstrued as an elephant packet. However, due to the inherent false positives associated with BFs, there might be an occasional match of a mouse flow within the elephant BF causing M_E to be non-zero. The probability of such collisions is $\beta = (1 - e^{kn/m})^k$ [8], where k is the number of hash functions, n is the number of elements supported by the filter, and m is the size of the filter. On the other hand, E_M exists because of the inherent online nature of our framework; a flow is deemed mouse until enough packets arrive to qualify for an elephant, thus initial packets are always construed as mouse. Note however that these false positives can be easily detected easily by accessing the particular flow record in the flow memory; the flow record for mouse flow will be either non-existent or will not be large enough to classify as an elephant.

For any sampling framework, the most fundamental constraint is applied by the processing limits. We define c as the capacity or the maximum number of packets which can be sampled at a given rate constrained by the processing power. More formally, this basic resource constraint inequality can be written as follows:

$$N = M_S + E_S \leqslant c,$$

where N is the total number of packets sampled and M_S and E_S are the actual number of mouse and elephant packets sampled per unit time respectively assuming an oracle which can perform perfect classification. Trivially, from our definitions, $M_S = M_M + M_E$ and $E_S = E_E + E_M$.

Due to misclassification of mouse packets as elephant, some of the mouse packets are sampled at elephant rate. Hence, we need to reduce this fraction from M_M and add it to M_E. Thus, we get the following equations:

$$\begin{aligned} M_M &= s_M \cdot M \cdot (1 - \beta), \\ M_E &= s_E \cdot M \cdot \beta, \end{aligned} \tag{1}$$

where, s_M and s_E are the sampling rates of the mouse and elephant classes and M is the total number of mouse packets. Generally, at larger sizes for the filters, $\beta \ll 1$, so M_M to be simply equal to $s_M \cdot M$.

Similarly, every elephant flow at the beginning of its evolution is treated as a mouse flow, until an estimated T/s_M packets have been encountered for it, assuming T represents the threshold number of *sampled* packets at which a flow changes from mouse to elephant. If F_E represents the total number of elephant flows, E_M is at most F_E times the threshold T, since a flow will be immediately labeled as an elephant once T packets are sampled for that flow.

$$E_M \leqslant F_E \cdot T.$$

We need to reduce these many packets sampled at mouse sampling rate, to get the expression of E_E:

$$E_E = s_E \cdot (E - E_M/s_M). \tag{2}$$

We now plug in the values for M_M, E_M and E_E to derive a more general inequality (assuming negligible values for β), as follows:

$$N \leqslant s_M \cdot M + F_E \cdot T + s_E \cdot (E - E_M/s_M) \leqslant c. \tag{3}$$

Using this general inequality, we will now show how a network operator can compute appropriate values for s_M and s_E to configure a *general* two-class size-based sampling, to satisfy the resource constraints and at the same time maximize his metric of interest—flow coverage or accuracy or both.

Maximizing flow coverage

One of the main objectives we consider is increasing the number of unique flows captured by CLAMP, either for a particular group or for all the flows. Due to the heavy-tailed nature of Internet traffic, there are many mouse flows with a few elephants. To increase the flow coverage of mouse flows, we need to increase the sampling rate s_M. However, for a given sampling budget and processing constraints, we cannot increase it indefinitely. The Eq. (3) is a quadratic in s_M as given below:

$$M \cdot s_M^2 + (F_E \cdot T + s_E \cdot E - c)s_M + s_E \cdot E_M \leqslant 0.$$

We can solve this equation to obtain two roots, of which the valid one for s_M is as follows:

$$\begin{aligned} s_M &= \left(t + \sqrt{t^2 + 4s_E \cdot E_M \cdot M}\right)/(2 \cdot M) \quad \text{where,} \\ t &= c - s_E \cdot E - E_M. \end{aligned} \tag{4}$$

All positive values of s_M less than the one defined above will satisfy the processing capacity c. We can get estimates for E, M and E_M historically, based on the past measurement cycle(s). Due to the heavy-tailed nature of Internet traffic, maximizing the value of s_M will automatically lead to increasing the overall flow coverage. Thus, we maximize the value of s_M with respect to s_E to obtain the values for s_M and s_E that will lead to maximizing the flow coverage.

$$\begin{aligned} s_M &= \min\{(c - E_M)/M, 1\}, \\ s_E &= \max\{0, (c - M - E_M)/(E - E_M)\}. \end{aligned} \tag{5}$$

In most cases, the first configuration with s_M set to $(c - E_M)/M$ and s_E to 0 would work fine because sampling capacity c is typically very small. Only when we have higher sampling capacity, we will need to configure s_M to 1 and s_E to a value less than or equal to $(c - M - E_M)/(E - E_M)$, so as not to underutilize the spare sampling capacity. We refer to this sampling scheme as *sample and block* as it is the opposite of *sample and hold* [7] that is designed to identify large flows. Sample and block samples packets belonging to mouse flows at the maximum possible sampling rate s_M, and as soon as a mouse flow becomes an elephant, it stops sampling further packets for that flow. Thus, it tries to allocate most of the sampling budget for identifying the mouse flows.

It is useful to compare the coverage gains of such an approach with that of random packet sampling with probability s_R. To perform fair comparisons, the invariant we maintain is the sampling budget c in both schemes. We compute the maximum packet coverage gain for *mouse* flows obtained at this configuration, as follows:

$$\begin{aligned} G_{max} &= (s_M \cdot M)/(s_R \cdot M) = \frac{(c - E_M)/M}{c/(M + E)} \\ &= (1 + E/M) \cdot (1 - E_M/c) \quad \text{for } E_M < c \leqslant (M + E_M). \end{aligned} \tag{6}$$

While G_{max} is actually packet coverage gain, it is directly related to the flow coverage gain for mouse flows, because all the mouse flows have less than or equal to T packets. We note that G_{max} is dependent on two terms: The first term, $1 + E/M$, is solely dependent on the traffic mix or the trace characteristics. However, the second term, $1 - E_M/c$, is dependent on the amount of misclassification occurring for elephant flows. Note that the gain in mouse coverage G_{max} is due to the reduction in number of elephant packets sampled and increase in the sampled mouse packets; the sum of both these sampled packets is the same as that of random packet sampling.

Along with increasing the packet and flow coverages, sample and block can also increase the accuracy of the flow size estimates for the mouse flows, as we discuss next.

Improving accuracy of specific types of flows

CLAMP when configured to sample and block, achieves the maximum packet and flow coverage for mouse flows. Because mouse flows consist of only a few packets, an increase in their coverage directly results in increased accuracy for their volume estimates. However, this increase in the flow coverage comes at the expense of reducing the accuracy of medium and large flows.

In order to achieve flow estimates as accurate as obtained using random packet sampling, we must sample at least those many elephant packets. This can be simply achieved by configuring both s_M and s_E equal to s_R. We can reduce s_M from its value for maximum coverage, and increase s_E so that larger number of samples are obtained for the elephant packets thus increasing the accuracy of flow size estimates for medium and large flows. In this way, the network operator has the flexibility to configure CLAMP for achieving maximal flow coverage for mouse flows, along with sufficiently accurate flow size estimates for medium and large flows. This can be attained by tuning to a point between the configurations for maximum coverage and maximum accuracy. Further, this analysis can be easily extended to multiple classes by computing the sampling rates corresponding to each class one at a time, based on the requirements of the network operator.

Multiple classes

While we have considered a two-class size-based sampling, we can trivially extend our analysis to three (or more) classes. Suppose there are three classes—mouse (M), medium flows (E_1) and elephant (E_2). We can first combine E_1 and E_2 into one *virtual* class E and perform the analysis for M and E to compute s_M and s_E. Once s_E is computed, we can use this sampling budget for E_1 and E_2 as s_R and do the analysis to get s_{E1} and s_{E2}. Basically, by using simple divide-and-conquer approaches, we can obtain the specific sampling rates required for different classes of flows.

Note that even as the number of classes increases, we do not need separate hash functions across different BFs. There are two subtle issues however. First, some BFs may be smaller than the others. In this case, one can imagine an independent stream of hashes (as many as the maximum that any particular BF uses) that are input to all the BFs and let the BFs choose which (set of) hash values it wants to use. Second, the sizes of all BFs need not be the same. Depending on the objective, one could choose to assign arbitrary sizes for different BFs. This does not, however, require new hash functions as one could just compute the index as h *modulo* m, where h is the hash and m is the number of bits in the BF. In this paper, we only focus on the two-class solution; we do not consider the multiple class scenarios further in this paper and is part of our future work.

5. Evaluation

While we envision CLAMP to be implemented in hardware for high speeds, we built a prototype software implementation of CLAMP for evaluation. The most important component of CLAMP is the packet classification data structure, which is implemented as a vector of binary Bloom filters (hash tables). Our prototype allows selecting the number of hash functions (k), number of entries (m) in each filter and even the epoch size (e). We use Bob Hash function as suggested by [12] for packet sampling at line speeds. Each hash function is initialized with a different 32 bit value.

Using this prototype implementation, we evaluate the efficacy of CLAMP over real-world traces. We also implemented other size-based sampling frameworks [11,13] and use these in our comparisons. The main metric we use for comparison is memory usage, although CPU usage is also an important metric. The reason for focusing mainly on memory is that characterizing the CPU usage depends heavily on which functions are implemented in software and hardware. For most of the flow management functions, the CPU costs are similar in both architectures. We show how to configure CLAMP to obtain better flow coverage using our theoretical analysis in Section 4.2. We also discuss how CLAMP can be configured for maximizing the mouse flow coverage and accuracy by employing the *sample and block* scheme. Finally, we show the results on accuracy of flow size estimates by CLAMP, for large flows and compare them with random packet sampling. (Table 1)

We used two real-world anonymized traces to analyze the performance of CLAMP. The first is a 10 minute OC-48 trace, referred to as ABIL, published by NLANR [2] with 34 million packets and about 2.2 million flows, while the second, referred to as CHIC, is a 1 minute OC-192 backbone trace of a tier-one ISP published by CAIDA [1] with 12 million packets and 1.1 million flows. Both traces exhibit heavy-tailed flow size distributions that we assume in our analysis. These traces are summarized in Table 2.

For all the experiments, we represent the constraint c using a random packet sampling probability of s_R. Recall that the processing constraint c reflects the number of packets that can be processed in a second. We can represent this equivalently as processing s_R fraction of the total packets, where $s_R = c/N$ where N is the total number of packets in the epoch. Thus, in all our results, the value of s_R directly governs the associated packet processing constraint c.

Table 1
Summary of notations (Note: All packet and flow terms are defined per epoch).

Name	Definition
β	Filter error rate
c	Capacity or maximum number of packets sampled satisfying processing constraints
s_R	Random packet sampling rate
s_M	Sampling rate for mouse class
s_E	Sampling rate for elephant class
T	Flow size threshold (number of packets - unnormalized) separating mouse and elephant classes
F_M	Number of mouse flows in the traffic
F_E	Number of elephant flows in the traffic
M	Number of packets in the traffic belonging to mouse flows
E	Number of packets in the traffic belonging to elephant flows
M_M	Number of mouse packets classified as mouse and sampled at mouse rate
M_E	Number of mouse packets misclassified as elephant and sampled at elephant rate
E_E	Number of elephant packets classified as elephant and sampled at elephant rate
E_M	Number of elephant packets misclassified as mouse and sampled at mouse rate
N	Total number of packets sampled

Table 2
Traces used for our measurements.

Name	Date/Time	Duration	Packets	5-tuple flows
ABIL	2002-08-14/ 09:20	600 s	34,573,317	2,195,366
CHIC	2008-04-30/ 13:10	60 s	13,046,322	1,174,965

5.1. Memory usage

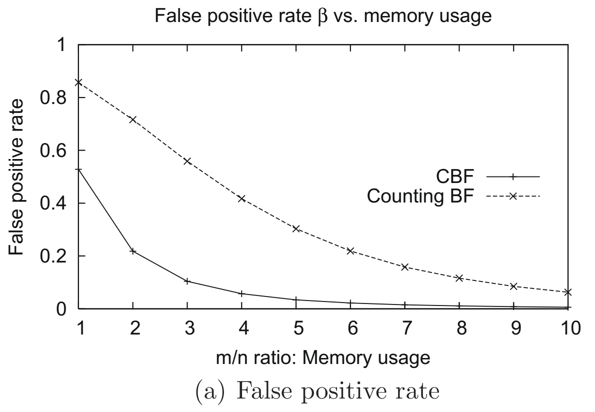
In this section, we compare CLAMP with other counter-based schemes for size-based sampling such as FlexSample [13] and sketch-guided sampling [11]. While the focus and usage of these solutions is different, they both share similar data structures (sketch and counting Bloom filters) to keep track of the flow sizes. In contrast, we rely on CBF that represents a much simpler data structure. In many cases, such simplicity comes at the cost of worsening some other metric such as, say, increasing the false positive rate of the filter. The results indicate, somewhat surprisingly, that CBF performs better in both memory consumption as well as false positives compared to counting BF alternatives, thus indicating that CBF achieves clear benefit and is not a tradeoff.

Why is higher false positive rate in the classification data structure bad for various sampling objectives? As an example, consider the case when we are interested in increasing flow coverage. According to Eq. (1), the mouse flow coverage is directly related to the number of mouse packets classified and sampled at mouse rate M_M. Mouse flow coverage, however, decreases as filter false positive rate β increases as these packets will be misclassified as elephant packets and thereby sampled at elephant rate (*i.e.*, less than the mouse rate for high flow coverage). Thus, low false positive rate for the classification data structure is important for such objectives.

Fig. 3(a) shows the false positive rate β (calculated using Eq. (1)) as we change the amount of over-provisioning m/n, where m is the number of bits or hash buckets provisioned and n is the number of elements inserted into the data structure. Even though we keep m/n consistent, the total amount of memory allocated to counting BFs is way higher than regular BFs, since each counter is much larger (*e.g.*, 32 bits) compared to the number of different BFs (two BFs for our two-class solution) we use in our architecture. Typically, both data structures increase the number of hash functions applied on each packet as a way to increasing this over-provisioning ratio.

From the Fig. 3(a), we show the false positive rate as we vary the m/n for ABIL trace. can observe that CLAMP exhibits much faster decrease in false positives (or misclassifications) compared to counting BFs. In addition, the granularity of over-provisioning is much higher in the counting BFs, as it employs counters as opposed to bits in regular BFs. Fig. 3(a) shows that CBF achieves a filter false positive rate of 6% at a m/n ratio of 4 (using 174.5 KB of memory). Counting BFs require an m/n ratio of about 10 and about two orders of magnitude (about 80×) more memory (using 13.96 MB of memory) in order to achieve the same false positive rate.

On the other hand, even considering only same number of entries (bits and counters) across both CBF and counting



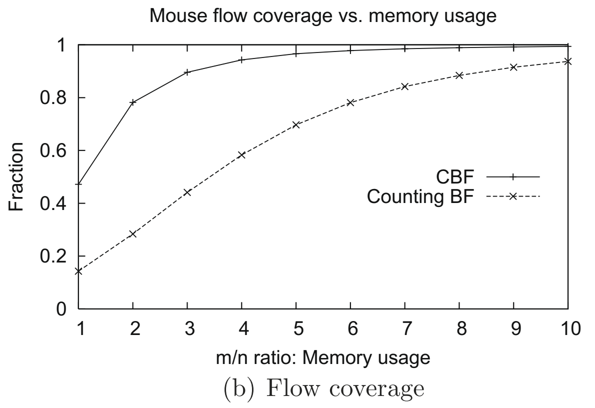


Fig. 3. Effect of varying memory usage for CBF and counting BF ($T = 1$, $s_M = 1$, epoch = 600 s).

Bloom filter, CBF still obtains at least up to 6× reduction in the number of false positives (at $m/n = 5$, CLAMP has about 4% false positives while counting Bloom filter experiences about 25% false positives). What makes these gains even more significant is the fact that we do not consider the extra overhead associated with size of the counters (in the counting Bloom filters). If we factor in this disparity, the benefit associated with CBF will increase even further.

In Fig. 3(b), we compare the mouse flow coverage (which is equal to M_M for $T = 1$) obtained using CLAMP as well as counting BFs. The configuration allowed mouse flow coverage to be maximum, *i.e.*, s_M set to 1, thus ensuring a worst case analysis for the filter with the maximum number of flows inserted in the Bloom filter over the trace. Typically, the false positive rate, β, associated with either of the two data structures will influence the maximum number of mouse flows that can be recorded, as the false positives will cause mouse flows to be misclassified as elephant flows and thus will be sampled at smaller rates. From the Fig. 3(b), we can observe that the number of mouse flows recorded with CBF is significantly higher (up to 3.5× at $m/n = 2$) for similar over-provisioning ratios. If we hold the amount of memory constant (not shown in figure), our gains are even more substantial.

Our results also indicate that beyond an over-provisioning ratio of 3–4 (i.e., about 3–4 hash functions), the marginal benefit is significantly reduced. However, the authors in [13] have used 10 hash functions for their evaluation. We fix m/n to 10 (with memory of 436.25 KB) for fair comparison with counting BFs for all the other experiments.

5.2. Effect of varying epoch size

In this experiment, we study the effect of choosing a particular epoch size for measurements. Recall that at every epoch, we reset both the Bloom filters as well as the flow memory as mentioned earlier in Section 3. Thus, a large epoch size allows effective warming up of the Bloom filters with appropriate state required to differentiate between mouse and elephant flows. However, it also increases the amount of memory that needs to be allocated towards the classification data structure since the number of flows is directly proportional (typically) to the epoch size. If the epoch size is small, then the data structure needs to be frequently warmed up before it can effectively classify flows, but has the advantage of using a smaller data structure.

Typically, the epoch size choice is dependent on many factors, one of which is the particular flow monitoring objective. For example, if our goal is to maximize the mouse flow coverage, we need to keep state about elephant flows long enough so that we do not keep retraining the classifier again and again. We show in Table 3, the break-up of E_M and s_R corresponding to achieving maximum mouse sampling rate (*i.e.*, $s_M = 1$) for different epoch sizes. The value of E_M directly corresponds to the misclassified elephant packets (at the beginning of a flow). We note that for each of the configurations in Table 3, packet and flow coverage for mouse flows is very close to 100% (not shown in table).

From the table, we can observe clearly that as we increase the epoch size, the effective contribution of E_M to the total sampling budget reduces. This is expected since as we increase the epoch size, the total sampling budget measured as number of packets that can be sampled in the epoch, increases. In addition, due to the fact that the classifier is not reset until the end of the epoch, the classifier is quite effective in filtering out elephant packets. Still, the contribution of E_M to c at an epoch of size 60s is high ($E_M/c \sim 0.676$) which results in a higher equivalent s_R (to obtain a mouse coverage of 1).

We note that this problem is inherent in *any* size-based sampling framework which is used for an online traffic analysis without the prior knowledge of elephant flows. However, a simple optimization to our CBF, namely not

Table 3
Effect of varying epoch size (CHIC trace, $T = 1$).

Epoch size	s_R	E_M	E_M/c
5 s	0.208	2,141,542	0.789
10 s	0.179	1,757,734	0.754
20 s	0.157	1,477,898	0.720
40 s	0.147	1,344,678	0.701
60 s	0.134	1,179,340	0.676

resetting the CBF in every epoch, but every *few* epochs, can solve this problem to some extent, as we discuss next.

Resetting the flow memory after every epoch is required to meet the constraints imposed by the memory (DRAM) of the sampling device. On the other hand, resetting CBF is useful especially if the amount of memory allocated to CBF is small. Otherwise, the Bloom filters will be filled up too fast rendering it almost useless for classification. Resetting the CBF frequently may result, however, in a considerable increase in E_M. Thus, while resetting both CBF and flow memory is important, a critical question is whether to reset one or both or none of these every epoch.

In Table 4, we consider three strategies—all reset every epoch, flow memory reset every epoch (of 10 s) but no CBF reset for this trace (of duration 60 s), and no reset. In each of these strategies, our aim is to achieve the best possible flow coverage; thus, we configure CLAMP with s_M equal to 1 and the epoch size is 10 s (OC-192 trace). We observe that only flow memory reset case is almost as good as no reset, in terms of E_M and mouse packet coverage M_M, but with slight increase in total number of sampled packets (can be seen from the calculated s_R to achieve the same level of coverage). Resetting the CBF too frequently (as in the all reset case) leads to a much larger number of E_M packets unnecessarily, thereby reducing the sampled mouse packets. In summary, our experiments indicate that depending on the particular flow measurement objective, it is advantageous to choose a different epoch size for resetting the CBF data structure as compared to the flow memory itself.

Table 4
Effect of resetting CBF and FM on coverage for flows of size 0–100 packets ($s_M = 1, T = 5$, epoch = 10 s).

Case	s_R	Packet sampling	CLAMP	Gain
All reset	0.307	53.8%	99.2%	1.84×
FM reset only	0.249	47.4%	99.9%	2.11×
No reset	0.239	46.4%	99.9%	2.15×

5.3. Flow coverage

According to Eqs. (5) and (6), the best possible coverage for mouse flows can be achieved by setting CLAMP in *sample and block* mode with $T = 1$. Table 5 shows the results for ABIL trace with epoch size of 600 s and CBF allocated 436.25 KB of memory. In Table 5, we show the gains for all flows of size 1–1,000 packets; this will be slightly less than those obtained for mouse flows alone according to Eq. (6), since we include some elephant flows also.

We can observe from Table 5 that, as we increase the random sampling probability s_R, the coverage gain of CLAMP increases from 3.74× (at $s_R = 0.001$) to almost 8.32× (at $s_R = 0.073$). Increasing s_R beyond 0.073 will reduce the gain, since the mouse sampling rate s_M can already be set to 1. As mentioned earlier, even with $s_M = 1$, we can see that CLAMP almost captures 99.4% of flows (with the remaining 0.6% attributed to the false positive probability associated with the elephant Bloom filter). Thus, we can clearly conclude that CLAMP provides an order of magnitude more flow coverage for this traffic mix as compared to random packet sampling.

For the CHIC trace, we show the results in Table 6. CLAMP achieves a maximum gain of 84% for this trace at a random packet sampling rate of 0.307, while at lower sampling rates ($s_R = 0.016$), the gains are just 11%. There are two important observations: First, at very low sampling rates (~0.001), CLAMP is almost as bad as random packet sampling since there is not enough sampling budget mouse flows can 'steal' from elephants to improve their coverage. Second, at high sampling rates such as 0.307, we get a gain of 84% which is not as good as we obtained for the ABIL trace (8.32× at $s_R = 0.073$). This is in part because of the flow size distribution of CHIC trace (see Eq. (6) for the $(1 + E/M)$ term); CHIC has a lesser fraction of elephant flows compared to the ABIL trace. But the major reason is attributed to the high E_M/c ratios for all of these configurations in Table 6, due to smaller epoch size (10 s)

Table 5
Coverage for flows of size 0–1K packets (ABIL trace, $T = 1$, epoch = 600 s, memory = 436.25 KB).

s_R	s_M	Packet sampling		CLAMP		Coverage gain
		# Flows	Percentage	# Flows	Percentage	
0.001	0.0043	6,524	0.3	24,454	1.1	3.74×
0.004	0.0267	22,472	1.0	114,188	5.2	5.08×
0.016	0.146	73,968	3.4	456,357	20.8	6.17×
0.064	0.845	233,825	10.7	1,894,616	86.4	8.10×
0.073	1.0	261,872	11.9	2,179,399	99.4	8.32×

Table 6
Coverage for flows of size 0–100 packets (CHIC trace, $T = 5$, epoch = 10 s, memory = 50 KB).

s_R	s_M	Packet sampling		CLAMP		Coverage gain
		# Flows	Percentage	# Flows	Percentage	
0.001	0.0010	5,003	0.4	4,984	0.4	0.99×
0.004	0.0041	19,079	1.6	19,697	1.7	1.03×
0.016	0.0179	69,623	6.0	76,960	6.7	1.11×
0.064	0.0965	216,170	18.7	293,108	25.3	1.36×
0.307	1.0	622,214	53.8	1,147,231	99.2	1.84×

and higher threshold ($T = 5$) that effectively decreases the coverage gain (see Eq. (6) for the $1 - E_M/c$ term). However, operating in *sample and block* mode for the CHIC trace (with $T = 1$, epoch = 60 s), a lower sampling rate ($s_R = 0.138$) results in a much better coverage gain of $3.05\times$ (with only flow memory reset after every 10 s), by reducing the share of E_M/c. For brevity, we omit those for CHIC trace.

5.4. Flow size estimation

The second objective we consider is to provide flexibility to improve flow accuracy for certain types of flows. We characterize the accuracy of flow size estimates obtained with CLAMP using the mean relative estimation error metric. We calculate the relative estimation error for each flow, and then average it for small groups of flows on the basis of their sizes. For our analysis, we classify flows into three groups—small (1–1K packets), medium (1K–10K packets) and large flows (greater than 10K packets).

Getting good estimates for flow sizes of the small flows is really difficult for random sampling because of high quantization errors involved. Increasing the packet coverage for small flows will require allocating a disproportionately higher budget to the small flows as compared to the larger ones. Thus, we configured CLAMP in the *sample and block* mode with $T = 1$, single epoch and memory 436.25 KB. Results are shown in Table 7 for flows of size 1–1K packets. For each of the points, we started with a base sampling budget (by fixing s_R) and allocating all this budget the mouse flows (s_M is the computed sampling rate) and ignoring completely the elephant flows. As we can see, at $s_R = 0.073$, we could sample all the mouse flows and thus $s_M = 1.0$.

We observe at all sampling rates s_R, CLAMP achieves much better accuracy than random packet sampling. Of course this result is expected because instead of sampling all the $s_R \cdot E$ elephant packets, our approach results in shifting some of this budget to sampling mouse packets. However, the comparisons at very low sampling rates ($s_R = 0.001$) are not very meaningful, because both CLAMP and random packet sampling are affected by huge quantization errors. However, for typical sampling rates ($s_R = 0.016$) and high sampling rates ($s_R = 0.073$), errors are highly reduced for CLAMP. At s_R equal to 0.073, for example, CLAMP gives an error of 7% with a standard deviation of 22%. This is almost $105.43\times$ better than random packet sampling. We also note that till a sampling rate of $s_R = 0.064$, all mean relative estimation errors are due to overestimation of flow size estimates. For $s_R = 0.073$, the mean error for CLAMP of 7% is due to underestimation. This is expected because of a high value of sampling rate used for normalization.

We show the results for medium (1K–10K) and large (10K–more) size flows in Table 8. In the table, we start with the *sample and block* configuration, *i.e.*, $s_E = 0$ and $s_M = 1.0$ and increase s_E and reduce s_M correspondingly such that the total number of packets sampled remains the same. By increasing s_E, we sample more packets corresponding to elephants and thus increase the accuracy of the flow size estimates. For example, at $s_E = 0.064$, the mean relative error of medium flows reduces to 0.3% from about 10.2% at $s_E = 0.001$. At this value of s_E, CLAMP performs similar to that of random packet sampling. In almost all the cases, the flow coverage remains at almost 100%. Interestingly, small values of s_E does not reduce the overall flow coverage as well, as can be seen under the coverage column in Table 8, except when we increase s_E beyond s_R. This is because large values of s_E lead to reducing the value of s_M significantly. Given all flows initially start off being mouse, with small s_M several medium flows fail to get even sampled to take advantage of higher s_E allocated to such flows.

Table 7
Accuracy for small flows of size 0–1K packets (ABIL trace, sample&block, $T = 1$, memory = 436.25 KB).

s_R	s_M	Packet sampling			CLAMP			Comparison	
		# Flows	Error	Dev.	# Flows	Error	Dev.	Error reduction	Coverage gain
0.001	0.0043	6,524	342.64	431.50	24,454	91.30	102.34	3.75×	3.74×
0.004	0.0267	22,472	96.87	109.91	114,188	18.39	16.48	5.27×	5.08×
0.016	0.146	73,968	28.82	27.64	456,357	3.69	2.69	7.81×	6.17×
0.064	0.845	233,825	8.37	6.55	1,894,616	0.07	0.28	119.57×	8.10×
0.073	1.0	261,872	7.38	5.65	2,179,399	0.07(u)	0.22	105.43×	8.32×

Table 8
Accuracy for medium and large flows (ABIL trace, $s_R = 0.073$, $T = 1$, memory = 436.25 KB).

CLAMP		1–10K			10K–more			Coverage	
s_E	s_M	# Flows	Error	Dev.	# Flows	Error	Dev.	1–10K	10K–more
0	1.0	3,152	0.999	0.0005	594	0.999	5.515	99.9%	99.5%
0.001	0.982	3,156	0.102	0.729	597	0.091	0.245	100%	100%
0.004	0.935	3,156	0.019	0.349	597	0.0024	0.121	100%	100%
0.016	0.75	3,156	0.0074	0.179	597	0.0055	0.059	100%	100%
0.064	0.148	3,156	0.003	0.084	597	0.0007	0.029	100%	100%
0.128	0.0004	2,307	0.372	0.694	592	0.0008	0.148	73.1%	99.2%
Packet sampling		3,156	0.0017	0.0756	597	0.0006	0.0262	100%	100%

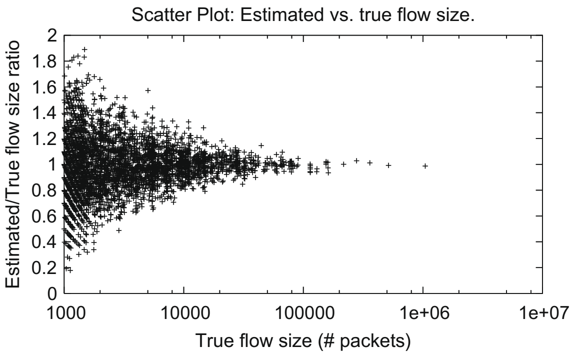


Fig. 4. Scatter Plot: flow size estimation accuracy for CLAMP.

5.5. Unbiasedness of estimates

We have already seen that CLAMP is able to achieve orders of magnitude better mean relative estimation error for small flows than random packet sampling and atleast as good as that for medium and large flows. However, unbiasedness of the flow size estimates is not verified by the absolute values for mean relative estimation error. Fig. 4 shows a scatter plot of the ratio of estimated and true flow sizes (in number of packets) on the *y*-axis with increasing true flow size on *x*-axis. We operate CLAMP at a typical sampling configuration, with $s_R = 0.01$ (s_M and s_E also 0.01) with $T = 1$ and epoch size of 600 s for CBF (with flow memory reset every 10 s) for ABIL trace. We only show the scatter plot for medium and large flows (flow size > 1000 packets).

From this scatter plot, we can see that most of the flows are accurately estimated at a sampling rate of s_R of 0.01 (within 20–30% of error). The estimates also converge as the true flow size increases. Finally, the presence of two-sided errors experimentally confirms the unbiasedness of flow size estimates for CLAMP.

6. Conclusions

Flow monitoring solutions in routers have evolved significantly over the years from their modest origins in simple NetFlow-like solutions. While most solutions revolve around better handling router resource constraints such as CPU, memory and flow reporting, there is little research on providing an efficient and flexible class-based sampling architecture, with dynamic class definitions that include specific flow properties such as the size. In this paper, we have discussed the architecture of CLAMP to address this challenge that involves the use of a set of simple Bloom filters for class membership and a feedback from the actual flow memory to record class-membership information about flows. Using this architecture, we have shown how to implement a simple two-class size-based packet sampling framework.

We have analyzed the architecture for this particular application both theoretically as well as empirically using a prototype software implementation over real ISP traces. Our results clearly indicate the simplicity of our approach over the previous solutions. For example, for some configurations, we achieve over 80× reduction in the amount of memory required for same rate of packet misclassifications. In addition, we have also shown how we can configure the CLAMP architecture for specific network monitoring objectives such as increasing flow coverage and improving the volume estimates of a specific class of flows (based on size). While we have primarily focused on size-based class definitions, our architecture itself is quite generic and lends itself to a wide variety of applications, some of which we intend to pursue in the future.

Acknowledgements

This work was supported in part by NSF Award CNS 08316547 and a grant from Cisco Systems.

References

[1] CAIDA Anonymized 2008 Internet Trace (equinix-chicago collection), ,<http://www.caida.org/data/passive/passive_2008_dataset.xml>.
[2] NLANR Abilene-I Internet dataset, <http://pma.nlanr.net/Traces/long/ipls1.html>.
[3] Cisco Systems. NetFlow Input filter, <http://www.cisco.com/en/US/docs/ios/12_3t/12_3t4/feature/guide/gtnfinpf.html>.
[4] B. Claise, Cisco Systems NetFlow Services Export Version 9, RFC 3954.
[5] N. Duffield, C. Lund, M. Thorup, Charging from sampled network usage, in: Proc. of IMW, 2001.
[6] C. Estan, K. Keys, D. Moore, G. Varghese, Building a Better NetFlow, in: Proc. of ACM SIGCOMM, 2004.
[7] C. Estan, G. Varghese, New Directions in Traffic Measurement and Accounting, in: Proc. of ACM SIGCOMM, 2002.
[8] L. Fan, P. Cao, J. Almeida, A.Z. Broder, Summary cache: a scalable wide-area web cache sharing protocol, IEEE/ACM Trans. Networking 8 (3) (2000) 281–293.
[9] N. Hohn, D. Veitch, Inverting Sampled Traffic, in: Proc. of IMC, 2003.
[10] R. Kompella, C. Estan, The Power of Slicing in Internet Flow Measurement, in: Proc. of IMC, 2005.
[11] A. Kumar, J.J. Xu, Sketch guided sampling - using on-line estimates of flow size for adaptive data collection, in: IEEE INFOCOM, 2006.
[12] M. Molina, S. Niccolini, N. Duffield, A comparative experimental study of hash functions applied to packet sampling, in: Technical report, AT&T.
[13] A. Ramachandran, S. Seetharaman, N. Feamster, V. Vazirani, Building a Better Mousetrap, in: Georgia Tech CSS Technical report GIT-CSS-07-01, 2007.
[14] A. Ramachandran, S. Seetharaman, N. Feamster, V. Vazirani, Fast monitoring of traffic subpopulations, in: Proc. of the Internet Measurement Conference (IMC), Vouliagmeni, Greece, October 2008.
[15] L. Yuan, C.-N. Chuah, P. Mohapatra, ProgME: towards programmable network measurement, SIGCOMM Comput. Commun. Rev. 37 (4) (2007) 97–108.
[16] Y. Zhang, M. Roughan, N. Duffield, A. Greenberg, Fast Accurate Computation of Large-scale IP Traffic Matrices from Link Loads, in: Proc. of ACM SIGMETRICS, 2003.

Mohit Saxena is a Ph.D. student at Department of Computer Sciences, University of Wisconsin-Madison. Previously he received his M.S. degree in Computer Science from Purdue University in 2008 and Bachelor of Technology in Computer Science from Indian Institute of Technology (IIT-Delhi) in 2006. His current research focus is on developing new operating system abstractions for flash memory and solid-state disks; and finding better ways to integrate them in computer systems. Earlier, he has researched scalable and memory-efficient packet sampling algorithms; analyzed the storage and content-distribution networks of video sharing services in Web 2.0; and investigated the problem of location estimation in the context of wireless sensor networks.

Ramana Kompella is an Assistant Professor of Computer Science at Purdue University. His main research area is computer networks. Particular topics of interest include scalable inference mechanisms for fault localization in enterprise as well as backbone networks, scalable streaming algorithms and architectures for various router functions such as traffic measurement, attack detection, packet classification and fair queuing, and finally, designing resource-efficient scheduling algorithms in wireless networks. Many of his past inventions resulted in direct industrial impact. His dissertation research resulted in the development of sophisticated fault localization tools that can pin-point the location of the failure in large-scale backbone networks.