A Scalable Architecture for Maintaining Packet Latency Measurements

Myungjin Lee†, Nick Duffield‡, Ramana Rao Kompella†
†Purdue University, ‡AT&T Labs–Research

ABSTRACT

Latency has become an important metric for network monitoring since the emergence of new latency-sensitive applications (e.g., algorithmic trading and high-performance computing). In this paper, to provide latency measurements at both finer (e.g., packet) as well as flexible (e.g., flow subsets) levels of granularity, we propose an architecture called MAPLE that essentially stores packet-level latencies in routers and allows network operators to query the latency of arbitrary traffic sub-populations. MAPLE is built using a scalable data structure called SVBF with small storage needs.

Categories and Subject Descriptors

C.2.3 [Computer Communication Networks]: Network management

General Terms

Measurement, data structures

1. INTRODUCTION

In recent years, latency has evolved into a metric that is as important as throughput in IP networks. While low latency is a desirable property for any network-based application, this obsession towards low end-to-end latency stems from the stringent requirements of many new kinds of datacenter, cloud, and wide-area applications that have become popular in the recent times. For instance, low-latency requirement exists for partition/aggregate type workloads that have become popular in the recent times. For instance, low-end-to-end latency stems from the stringent requirements of various places in the network that will help them identify root causes of SLA violations, determine offending applications that may hurt the performance of others, and so on. In light of the importance of these measurements, there has been some recent research on developing measurement mechanisms such as lossy difference aggregator (LDA) [5] and reference latency interpolation (RLI) [6]. A key limitation of these existing techniques is that they only obtain latency measurements at the granularity of a fixed pre-configured aggregate (e.g., across all packets in LDA). By making the granularity of the aggregates for latency measurements part of the architecture, these prior architectures are quite ossified, lacking flexibility to obtain arbitrary latency measurements (e.g., per-packet, flow subsets).

Such flexibility can be enabled with finer-grained latency measurements that are per-packet latencies. Thus, we propose an architecture called MAPLE that achieves these measurements in a scalable fashion. Then, any other forms of aggregation (per-flow, per-prefix, all packets), that may be of importance to network operators, are easily composable from these packet-level measurements. At its heart, MAPLE contains a scalable packet latency store (PLS) designed to simply store latencies of all packets in a scalable and efficient fashion. While we let readers refer to [7] for more details, we mainly discuss PLS in this paper due to interest of space.

2. PACKET LATENCY STORE

PLS stores per-packet latency measurements along with packet signatures. Thus, we first need to measure per-packet latencies somehow. For restricted settings, such as packets within a router, we can obtain approximate delays on a per-packet basis by using techniques in prior work [6]. For ease of exposition, we assume that packet headers can be modified to carry timestamps. Note that in all cases, we assume high precision time synchronization between the two measurement points, which has become feasible in modern times due to the increasing adoption of IEEE 1588 [3].

We consider two major factors in storing packet latencies: 1) storage efficiency and 2) insert/lookup times. Given a storage budget, we aim to store as many packet latencies as possible while providing fast insert and lookup operations. Simple hash table does not scale because it has high storage overhead (e.g., 32 bit packet hash and 20 bit delay for a given packet) and its collision resolution methods like chaining or probing make the hash table infeasible for a high speed link (e.g., an OC-192 link).

For the kind of applications we envision, such as performance diagnosis or detecting SLA violations, we can exploit the fact that the latency values for each and every packet need not be precise, and can be approximate instead. We can then significantly reduce the memory usage—this is the key intuition behind our approach that involves the following two steps.

Selecting representative delays. Within a given measurement interval, there are typically only a few dominant latency values (depending on the utilization) where most of the packet values are clustered. In the worst case, the latency values can be all over the entire permissible range, but in general, this is typically rare. Thus, instead of storing packets and their associated delays, we can first cluster packets into equivalence classes based on the delay values, and associate a single delay value, called center, for all packets within the cluster. We use an online k-median clustering algorithm [2] to select k cluster centers dynamically. With clustering, we reduce the problem of storing < pi, li > tuple to < pi, ci > where li is the actual latency of packet pi and ci is the ith center. Given a data structure, we store and lookup the center id
corresponding to a packet; the actual latency value corresponding to the center will need to be looked up in a separate table.

**Storing packet delays compactly.** For each cluster, we leverage approximate membership query data structures such as Bloom filters (BFs), that have gained significant prominence in networking applications recently, for better efficiency in storage (in terms of bits/packet) as well as implementation in hardware. There are several data structures to extend the basic Bloom filter such as Partitioned Bloom Filter (PBF) and Combinatorial Bloom Filters (COMB) [4]. PBF maintains BFs associated with individual centers, which makes BF provisioning difficult. In COMB, there is a trade-off between insert and lookup speeds. In order to keep up with a high line rate, insert operation must not slow down. Thus, sacrifice in lookup speed is inevitable. More detailed discussion about these data structures can be found in [7].

Our new data structure called Shared-Vector Bloom Filter (SVBF) addresses the limitations of these solutions by essentially preserving the simplicity of a single BF, but reducing the lookup complexity significantly. Specifically, we store the bits corresponding to different delay values for the same packet close-by so that during queries we can read all the bits in a burst instead of reading them sequentially from various bit positions.

**Insert:** The insert operation is quite similar to a regular BF, except for a small modification. In regular BF, each packet is hashed using multiple hash functions, and bits at those indices are set to 1. In SVBF, we use the hash function index as an offset into a vector of delay values. Thus, we set the bit corresponding to \( h_i(s_{pkt}) + g_i \), where \( g_i \in [0, k - 1] \) is the group number of the packet. This is shown in Figure 1 where a packet that matches the second center \( c_2 \) (group id 1) is added into the BF using hash functions \( H_1 \) and \( H_2 \). The offset at which the bit is set is 1 for this second center.

**Lookup:** Given a packet \( s_{pkt} \), we first hash the packet to obtain various hash indices \( h_i(s_{pkt}) \). From each of these bases, we read the next set of \( k \) bits, i.e., \( h_i(s_{pkt}) \) to \( h_i(s_{pkt}) + k - 1 \), to obtain bitmap \( B_i \). We compute the bit-wise AND across all these bitmaps, \( B = B_1 \land B_2 \land \ldots \). In the final bitmap \( B \), the offset where a bit is set to 1 is the group id.

The biggest gain of SVBF is that, it relies on ‘burst reads’ which are simpler than random reads that COMB suffers from. Thus, instead of \( k \) memory accesses, we only need \([k/w] + 1\) memory accesses for each hash index. For example, for \( k = 50 \), we can obtain the bitmaps in a total of \( 3 \times h \) memory accesses assuming a 32-bit machine word, where \( h \) is the number of hash functions.

3. **EVALUATION**

This section discusses how the various data structures discussed in §2 compare in terms of their accuracy, insert and lookup complexity for a given 5Mbit storage budget. We configure the data structures to support 50 centers \((k = 50)\). We use a trace in prior work [6] in which there are about 22.4M packets in a period of 60s.

**Accuracy of per-packet latency estimation.** Figure 2 shows comparison results in terms of absolute errors of per-packet latency estimates. We show mainly the upper quartile in this graph where the difference is the most pronounced. Clearly, at lower than 75 %tile, either SVBF or COMB would return the same (correct) group id if the packet is not misclassified. We can notice that COMB and PBF suffer from much higher discrepancies as early as the 70%tile onwards, while in contrast we can see that SVBF has an absolute error that is significantly lower in comparison. For example the 85%tile absolute error for COMB is close to 116µs while SVBF has an error of 19µs at the same percentile.

**Insert and lookup time complexity.** We study the complexity of insert and lookup time of each data structure. We tested 0.4 million packets for insert and lookup using a Linux machine with 2.66GHz Intel CPU. Figure 3 shows the complexity in microsecond precision. From the figure, we observe that SVBF works faster than COMB in both insert (45% gain on average) and lookup (28%) operations. While PBF achieves the same performance of SVBF for insert operation, PBF is four times slower than SVBF for lookup operation on average. Note that SVBF is implemented as software and can be optimized further in hardware platform.

4. **REFERENCES**