Sketching the Delay: Tracking Temporally Uncorrelated Flow-Level Latencies

Josep Sanjuàs-Cuxart
UPC BarcelonaTech
Jordi Girona 1-3
Barcelona 08034, Spain
jsanjuas@ac.upc.edu

Pere Barlet-Ros
UPC BarcelonaTech
Jordi Girona 1-3
Barcelona 08034, Spain
pbarlet@ac.upc.edu

Nick Duffield
AT&T Labs–Research
180 Park Avenue
Florham Park, NJ 07932
duffield@research.att.com

Ramana Rao Kompella
Dept. of Computer Science
Purdue University
West Lafayette, IN 47907
kompella@cs.purdue.edu

ABSTRACT

Packet delay is a crucial performance metric for real-time, network-based applications. Obtaining per-flow delay measurements is particularly important to network operators, but is computationally challenging in high-speed links. Recently, passive delay measurement techniques have been proposed that outperform traditional active probing in terms of accuracy and network overhead. However, such techniques rely on the empirical observation that packet delays across different flows are temporally correlated, an assumption that is not met in presence of traffic prioritization, load balancing policies, or due to intricacies of the switch fabric.

We present a novel data structure called Lossy Difference Sketch (LDS) that provides per-flow delay measurements without relying on any specific delay model. LDS obtains a notable accuracy improvement compared to the state of the art with a small memory footprint and network overhead. The data structure can be sized according to target accuracy requirements or to fit a low memory budget.

We deploy an actual implementation of LDS in an operational research and education network and show that it obtains higher accuracy than temporal correlation-based techniques without exploiting any knowledge about the underlying delay model.

Categories and Subject Descriptors

C.2.3 [**Computer-Communication Networks**]: Network Operations—*network monitoring*; C.4 [**Performance of Systems**]: General—*measurement techniques*; G.3 [**Probability and Statistics**]: General—*Probabilistic algorithms*

General Terms

Algorithms, Measurement

Keywords

One-way packet delay, network latency, network measurement, delay sketch

1. INTRODUCTION

Packet delay has become a key network performance metric, together with other metrics such as throughput and packet loss. This growth in importance of packet delay is mainly due to the emergence of a new class of network-based applications that demand extremely low end-to-end latency. For instance, algorithmic trading applications require end-to-end latencies to not exceed few microseconds, otherwise they may lose significant amount of revenue in the form of lost arbitrage opportunities [21]. High-performance computing applications form another class of such applications with message latencies directly impacting the amount of time it takes for the job (e.g., weather simulation) to be finished. Finally, modern data center applications have soft real-time deadlines [3] that typically are in the order of milliseconds, but once backend computation requirements are factored in, very little time is left for network delays.

Now, consider a network operator that is running and managing a network environment that supports low-latency applications, such as a data center network. Typically, many data centers host several thousands of machines connected via a network fabric that is often constructed out of commodity networking equipment (e.g., switches and routers). Depending on the requirements (e.g., full bi-section bandwidth), the network is often connected in a multi-rooted tree topology (e.g., a fat-tree) with several thousand switches providing multiple paths between servers for load-balancing purposes. Further, the cluster itself may be shared across several tens to hundreds of customers running tens to hundreds of different applications with potentially very different network usage patterns. Given the complexity stemming from the sheer number of network elements as well as the variety of networking-based applications, it becomes extremely difficult to debug and troubleshoot latency anomalies (such as delay spikes) throughout the network *without* proper latency measurement tools at various points in the network.

Traditionally, such measurements have been obtained using active probing in wide-area ISP networks [5, 24, 6, 28]. However, end-to-end network delays are an order of magnitude smaller in data center networks—order of *microseconds* compared to milliseconds. To capture delay dynamics at such microsecond granularity, high probing frequency (e.g., 10,000Hz [17]) is required, which makes this approach prohibitively expensive in practical scenarios. Further, diagnosing end-to-end delay anomalies requires measurements at various vantage points in the network—ideally, at each pair of interfaces within each switch in the network, so that the root cause can be localized down to a router or a switch. The network operator could then conduct a more extensive analysis, such as study the set of customers or applications that may be routed through that switch to carefully investigate the root cause of the problem.

Unfortunately, native switch/router support for packet delay measurements is sorely lacking. Today, NetFlow and SNMP form the two main measurement solutions that a router supports. Neither, however, focuses on delay measurements. In some environments such as the London Stock Exchange, operators resort to specialized measurement boxes (e.g., Corvil [1]) that can detect these delays at high fidelity. However, because of the high costs and the hassles of administering a new box in the network, such an approach does not scale well. The complexity of packet latency measurements comes fundamentally from the fact that we cannot easily just store a packet timestamp at two monitoring points, without incurring high storage and communication complexity, since the complexity is linear in the number of packets. It is therefore important to overcome the linear relationship between number of collected timestamps and network overhead for any solution to be scalable.

Recent work [17] proposed the lossy difference aggregator (LDA) to overcome the linear relationship between sample size and network overhead by intelligently aggregating timestamps between the two measurements points. LDA, however, provides only aggregate latency estimates *across* all packets, which may be inadequate for diagnosing customer-specific or application-specific latency issues [18]. As pointed out by prior work [18], flows may exhibit significant diversity in their latency characteristics at a given router, and hence, per-flow measurements are important for network operators. Unfortunately, the problem of measuring per-flow delay is harder in the environments we consider, since the number of flows can be quite large; collecting and exchanging per-flow state becomes prohibitively expensive.

The problem of measuring per-flow delay has been very recently explored in [18, 19]. Both papers exploit the *key observation* that packets exhibit significant temporal delay correlation in specific settings, i.e., packets that are transmitted close in time experience similar delays, even if they do not belong to the same flow. RLI [18], the most recent of the two, exploits this observation to inject simple active probes periodically and uses linear interpolation to estimate per-packet delay. At the downstream monitoring point, these estimated per-packet delays can be aggregated into per-flow latencies using only three counters per-flow.

While the assumption that packets exhibit temporal correlation is valid in a restricted subset of systems, this assumption *does not* hold true in more general scenarios where there is prioritization across packets with two or more parallel queues. For example, many modern routers support different queuing for prioritizing real-time traffic (e.g., VoIP, video) over regular data transmissions (e.g., Web). Thus, in these cases, there exists very little correlation between the delays of packets that end up traversing two different queues. Similarly, in many modern data center networks, packets are routinely load balanced across multiple paths using ECMP—temporal delay correlation may potentially exist across any given path but *certainly not* across paths. Finally, modern switch fabrics (e.g., Clos-network-based switch fabrics used in Juniper's T-Series routers [2]) are often composed of intermediate stages of switching with each packet being sent to a random intermediate location; the latency of a packet through the router may be different depending on the path within the router. (In such switches, packets are re-sequenced back because TCP does not interact well with reordering, but such reordering needs to be only on a per-flow basis.)

Thus, the assumption of temporal delay correlation is not universally applicable; unfortunately, schemes such as RLI will produce grossly inaccurate latency estimates if the assumption does not hold, posing a major hurdle for deploying RLI on a global basis. Switch vendors do not wish to be bothered about the specifics of the deployment scenario, and instead would like to have one scheme that is universally applicable across *all* possible scenarios. Our objective in this paper is to accomplish this task. Specifically, we focus on devising a scalable *delay-model-agnostic mechanism* to obtain per-flow latency measurements at microsecond granularity across two measurement points in the network.

In this paper, we propose a technique called lossy delay sketching (LDS) that essentially combines the model independence nature of LDA with sketching techniques that do not rely on per-flow state to obtain model-free and scalable per-flow delay estimation. LDS essentially maintains a series of hash buckets, with each bucket consisting of a timestamp sum and the number of packets that hash to the bucket (similar to an LDA bucket). In accordance with the spirit of sketching, LDS maps each flow to a random *subset* of buckets, that are potentially shared (partially or fully) by other flows. To minimize the effect of interference, we randomize the fate-sharing by maintaining different banks of buckets, similar to a sketch, with a different hash function.

While the basic idea of blending LDA with sketches makes intuitive sense, several problems must be overcome to design such a data structure. For instance, flows may differ in their delay properties as well as their sizes significantly. It is important to ensure the interference due to collisions does not impact the accuracy of the flow's latency estimates. We present theoretical analysis on determining the size of LDS in order to reduce this interference.

Thus, the main contributions of this paper are as follows.

- We propose a new data structure called LDS that *obtains per-flow delay estimates* and that does not rely on delay models (Section 2). It blends LDAs that are model independent with sketching techniques that provide per-flow measurements without per-flow state.

- We present a comprehensive theoretical analysis of the data structure and show how to size it to achieve the desired accuracy (Section 3).

- We introduce a series of practical enhancements to LDS that allow network operators to fine-tune the data structure for the specifics of an actual deployment scenario (Section 4).
- We evaluate LDS with real traffic collected at a large academic network (Section 5). Our results indicate that sketching is superior to existing techniques when temporal correlation is not present. Sketching is particularly accurate for large flows, even in the presence of loss. Additionally, the accuracy of a selected subset of flows can be easily incremented.

Finally, Section 6 covers the related work in the literature, while Section 7 concludes the paper.

2. DELAY SKETCHING

Our main goal is to measure the one-way delay introduced by a network between two measurement points on a per-flow basis. While we can typically support any definition of flow, usually, this will consist of the 5-tuple formed by source and destination IP addresses, originating and destination ports, and protocol. We mainly focus on obtaining per-flow average latency, but we also outline in Sec. 4 how we can obtain second moments as well.

Our architecture is oblivious to what locations exactly constitute the measurement points. Thus, we can imagine obtaining per-flow measurements within a switch or a router across an ingress and egress interface. Alternately, we can obtain measurements across two different routers. Note that both measurement locations are merely viewpoints along the path that packets follow, and do not need to be (although they could be) the emitter or final destination of the traffic. We call the first measurement point *sender*, and the second, *receiver*. We consider the reverse path measurements separately with the *receiver* becoming the *sender* and vice versa.

2.1 Assumptions

Single stream. We assume that the sender and receiver observe the same stream of packets. In general, this is highly dependent on the particular scenario. For instance, suppose we consider an ingress (egress) switch interface as the sender (receiver). The receiver (sender) may obtain (transmit) packets from (to) many different ingress interfaces. Thus, we assume there is a simple way to filter out the packets that travel from the sender to the receiver. Note that we do not assume packets flow through a single queue, or even in a FIFO order—just that we have a way to separate out packets that appear at both the sender and the receiver. Within switches, there are often internal headers that contain the port at which they originated and the port to which they are headed to, that we can use for this purpose. Across routers, we can leverage prefix-based filtering to identify the set of packets that travel through one given path (forwarding is prefix-based). Such routers do not need to be co-located or close in terms of network hops.

Packet loss. We assume packets can be lost between the sender and receiver. Depending on the scenario, the packet loss rates may differ significantly. For example, in a financial trading network, we may imagine the network to suffer from minimal packet loss. However, in a real backbone network, packet loss may be slightly more common. Typically, while some amount of loss resilience is required in our data structures, we assume the loss rates are still quite low (say $<1\%$) as TCP may not work well under higher loss rates.

Time synchronization. We also assume the clocks of the sender and receiver are synchronized. This is a common requirement of one-way packet delay measurement techniques [17, 19, 18]. Although techniques have been proposed that do not require clock synchronization (e.g., [25, 22, 29]), removing this assumption was out of the scope of this paper. To achieve fine grained precision, packet timestamping clocks can be synchronized to the GPS signal (i.e., using Endace DAG cards), or using the IEEE 1588 protocol [16]. Both these methods are capable of sub-microsecond precision and thus suitable for our needs [10]. (In our evaluation, we obtained traces from a production network that already uses IEEE 1588 protocol to synchronize measurements.)

Embedding timestamps in packets. Similar to prior work [17, 18], we assume that it is *not* possible to embed timestamps within IP packets because existing IP headers do not have a placeholder for timestamps, and it would require significant changes to router forwarding paths and other third-party components making it difficult. We note however that our solutions are important even in the context where router vendors can put a timestamp in a packet, as the number of flows may be still large.

In fact, for ease of exposition, we present a simple data structure called SDS using the *timestamp assumption*, i.e., assuming packets can be embedded with timestamps. We will, however, get rid of this assumption in Sec. 2.3 when we describe our main data structure LDS.

2.2 Simple Delay Sketch (SDS)

As mentioned before, we initially assume the sender can embed a timestamp into the packets to be measured for easy exposition of the delay sketching idea. (We will relax this in the next section.) Thus, the receiver can obtain delays for each packet, but still need a scalable mechanism to store per-flow latencies, which is obtained by the data structure we describe in this section. The main idea of our technique is to explore sketching techniques that have been studied before in the literature to obtain measurements without maintaining per-flow state, and requiring very few memory accesses per packet. Such techniques will allow us to compute a comparatively smaller compressed summary of the traffic that allows recovering approximate measurements. We assume measurements are performed in fixed time intervals, which we refer to as *measurement intervals*.

A canonical sketch data structure that we can exploit in our setting is the multi-stage filter [13]. In this data structure, each stage has C associated counters, which are initialized to zero. Then, for each incoming packet, a hash of its flow identifier is used to determine which counter will be updated in the 1st stage. If, for example, one wishes to measure flow sizes, then the packet size is added to that counter. Since every flow always hashes to a particular position, its associated counter can be queried to obtain an upper bound on its size (only an upper bound, since other flows can hash to the same position, i.e., can collide). Additional stages can then be added that are independent replicas of this scheme, thus randomizing collisions. Then, the estimated size of a given flow is the minimum of all of its associated counters in each stage. The Count-Min Sketch [9] is also based on a similar approach.

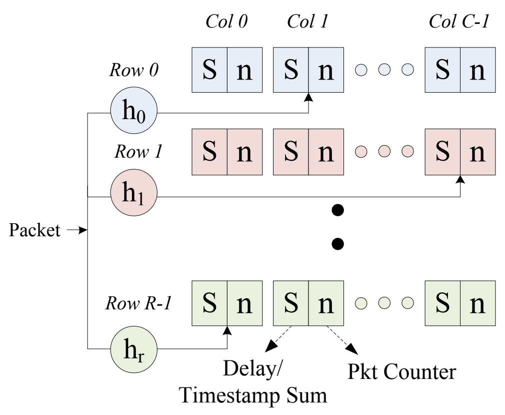

Figure 1: Basic data structure. In each cell, s stores the sum and n the number of packets that hash to that cell.

```
1: procedure Update state($flow$, $\tau$)
2:     for i=1, R do
3:         $j \leftarrow (hash(i, flow)\%C)$        ▷ Compute $i$th hash
4:         $SDS[i][j].S \leftarrow SDS[i][j].S + \tau$
5:         $SDS[i][j].N \leftarrow SDS[i][j].N + 1$
6:     end for
7: end procedure
```

Figure 2: SDS – Per-packet operations

Our initial idea is to use this sketching technique for per-flow delay measurement. The data structure we propose called Simple Delay Sketch (SDS) contains a series of cells organized in a matrix of R rows and C columns. Each row r has an associated pseudo-random hash function h_r that returns a value in the range [0, $C-1$]. Each cell of the matrix contains a tuple of values $<s,n>$, with s storing the sum and n the number of packets that hash to that cell. The data structure is graphically depicted in Figure 1.

Update. When a packet that belongs to a flow with identifier f arrives, for each row r, a position in the matrix is determined using its hash function h_r, which yields position $(r, h_r(f))$. Then, the cells in these positions are updated as follows. The s values of each cell are increased by the delay of the packet (i.e., the current time at *receiver* minus the timestamp embedded in the packet at *sender*), while n values are increased by one (i.e., maintains a count of the packets that hashed to that cell). In other words, s values contain the sum of all packet delays that hit that cell, while n values represent packet counts. Note also that the per-packet cost of this measurement scheme is, for each row, a hashing operation and two counter updates. The full algorithm is described in Figure 2.

Delay Estimation. If the data structure were single-row and infinitely large, and the hash functions were perfectly random, each non-empty cell would measure the average delay of the packets of a particular flow. That is, the data structure would be collision-free, since two flows would not hash to the same position. Therefore, to obtain the (exact) average delay of flow with identifier f, one would simply divide the s and n values of cell $(0, h_0(f))$.

```
 1: procedure Estimate delay($flow$, $SDS$)
 2:     $N_{min} = \infty$
 3:     for i=1, R do
 4:         $j \leftarrow hash(i, flow)$        ▷ Compute $i$th hash
 5:         if $SDS[i][j].N < N_{min}$ then
 6:             $N_{min} = SDS[i][j].N$
 7:             $S_{min} = SDS[i][j].S$
 8:         end if
 9:     end for
10:     return $S_{min}/N_{min}$
11: end procedure
```

Figure 3: SDS – Delay estimation algorithm

However, in practice, rows cannot be large enough, and there is a non-zero probability of several flows colliding (i.e., hashing to the same cell in one or more rows). When several flows collide, the value obtained by dividing the s and n values is a weighted average of the delays experienced by said flows, where weights correspond to the number of packets of each flow. Therefore, in practice, flow delays can not be exactly obtained, but only estimated from the data structure. To minimize the impact of collisions over measurement accuracy, it is desirable to have a large number of rows. For a given flow, we can obtain one estimate of its average delay for each of the rows, to then choose the one that is least contaminated by other flows.

Several strategies can be devised to obtain the estimates. For example, one could produce a final estimate by combining several cells, such as taking the mean or the median of the available estimates, like other sketching techniques do (e.g., see [9]). In our case, using the average would be problematic for flows that collided with larger ones in *any* of the cells, because the weight they would carry in their estimates would be small. This would invalidate the advantages of provisioning multiple rows to randomize collisions. Likewise, using the median would tend to drag each estimate towards the median delay of all flows.

We find it best to choose the cell that has the lowest n value to produce an estimate. This has two interesting properties. First, if one of the cells is collision free, the algorithm will choose it and, thus, produce an error-free result. Otherwise, it will pick the cell where the fewest amount of packets have collided, i.e., the one where the measured flow carries the highest possible weight. The full algorithm is described in Figure 3.

Of course, the cell with the smallest n will not necessarily produce the best estimate among all cells. For example, it may have collided, in one cell, with a large flow that has an extremely similar average delay. However, this strategy always chooses, among all the cells, the one where the measured flow carries the highest possible weight. In Sec. 3, we analyze the accuracy that this data structure offers, and provide guidelines to dimension and parametrize it.

The SDS data structure we presented in this section is a first-cut approach to blending the ideas of LDA with sketching techniques. However, SDS is only applicable if we assume embedding timestamps within packets—an assumption which is hard to achieve in practice, at least in the short term if not in the longer term. We now study a new data structure LDS that relaxes this assumption.

2.3 Lossy Difference Sketch (LDS)

In this section, we discuss our main data structure called lossy difference sketch (LDS) to obtain per-flow latency measurements *without* requiring the timestamp assumption. The LDS data structure starts with the basic SDS data structure, and uses the following ideas to make it practical:

1. To get rid of the timestamp assumption, the sender and receiver maintain separate copies of the data structure (described in the previous section), and the sender periodically transmits its copy to the receiver. The receiver then post-processes both sketches to obtain the delays of all packets that hash to each cell (Sec. 2.3.1). We need the sender and receiver to use consistent hashing (same hash function) to ensure packets hash to the same cell in both cases.

2. Since packet losses and reordering can occur between the sender and receiver, this may make the cells inconsistent across the sender and receiver. We detect losses easily since the number of packets does not match across the sender and receiver cells. We detect reordering using a separate field in each cell that stores a stream digest for each cell, similar to prior work [27, 20] (Sec. 2.3.2).

3. To handle packet losses, we map packets that belong to the given flow across several contiguous cells in essence forming a virtual LDA for each flow. We also use a stage of sampling to reduce the probability of a packet loss completely corrupting all cells for a given flow. We randomize the set of flows that collide in each row so as to randomize the fate-sharing. This randomization allows us to minimize the interference of other colliding flows on the estimates for any particular flow (Sec. 2.3.3).

The following subsections will discuss these ideas in detail.

2.3.1 Removing the Timestamp Assumption

The main problem when measurement points cannot embed a timestamp in the packets is that we cannot compute the packet delay at *receiver*. Thus, neither can we directly aggregate delays in the data structure as described in Sec. 2.2. However, we can achieve the same effect by using a simple, yet extremely powerful technique introduced in [17]. Ref. [17] states that to measure average delay of a set of packets, one does not need to embed a timestamp in the packet or transmit individual timestamps between *sender* and *receiver*. Instead, we can proceed as follows. In both measurement points, compute the sum of all packet timestamps, and maintain a packet count. Then, to compute average delay, compare the aggregate timestamps and divide over the total number of packets.

We leverage this idea in designing LDS as follows: In both measurement nodes, we maintain a sketch as the one described in Sec. 2.2, where each position aggregates the packet timestamps observed at each point, instead of packet delays. Both measurement nodes use the same hashing functions in order to map flows to the same counter positions. Then, at the end of the measurement interval, one of the resulting sketches is sent to the other measurement point. The aggregate timestamps at *sender* are subtracted from those at *receiver*. The result is exactly equivalent to a sketch that aggregates packet delays.

After this step, the average delay of the packets that hit a cell can be simply obtained by dividing its s and n, where s is now the sum of timestamps kept in each cell, instead of the aggregate packet delays. Assume a series of packets $p_1, p_2 \dots p_k$, with timestamps $t_1, t_2, \dots t_k$ at *sender* and $t'_1, t'_2 \dots t'_k$ at *receiver*. The delay of the ith packet is then $t'_i - t_i$. Our data structure calculates $\frac{s}{n} = \frac{\sum_{i=0}^{k} t'_i - \sum_{i=0}^{k} t_i}{k} = \frac{\sum_{i=0}^{k} t'_i - t_i}{k}$, i.e., the average packet delay.

Note that, while we focus on average delay estimation, other useful estimates, including per-flow delay standard deviation, can also be mined from the data structure as described later in Sec. 4.

2.3.2 Detecting Losses and Reordering

When packets are lost some of the n fields in *receiver* would be smaller than those obtained by *sender*. This is an important problem, because our data structure relies on the difference of s values at *sender* and *receiver* to calculate average packet delays, as explained in Sec. 2.2; using cells where the set of packet delays aggregated in each measurement point differs introduces severe error [17]. In general, when packet counts n do not match, the set of timestamps aggregated in the corresponding s fields will not be consistent. Thus, in LDS, we do not use such cells for delay measurement.

Packet reordering poses an additional challenge: n fields can match, while the set of aggregated packet timestamps might differ. This is a problem that is analyzed in more detail in [20]; in summary, at the boundaries of measurement intervals, packets might jump to the next (or previous) interval. This, coupled with loss, can easily cause matching packet counts, and mismatching sets of packet timestamps. As introduced in [27, 20], this problem can be solved by attaching, to each sketch position, a small digest d of the packets that hit such cell, which can be achieved simply with an extra hashing operation as follows.[1]

In LDS, thus, each cell will consist of the additional d field along with s and n for each cell. This value contains a digest of all packets that hashed to a cell. We require digests to be computationally lightweight, and to provide a means to detect loss and packet reorders with high probability. As discussed in [27, 20], an easy way to achieve this is to cumulatively XOR the hashes of the packet contents. This scheme guarantees that, using b bit hashes, mismatches will be detected with probability $1 - 2^{-b}$.

It is easy to see that, with this basic data structure, no accurate delay estimates can be produced for (i) those flows that experience even a single loss or reordering, and (ii) for those that collide with flows that have experienced such conditions. For such flows, packet digests (and often, n values) in *sender* will not match those at *receiver*, and thus will always be invalidated. We next discuss the mechanism used in LDS to make the flow estimates more robust to packet losses or reordering.

2.3.3 Robustness against Loss and Reordering

To make LDS more robust to losses and reordering, we leverage the basic idea used in LDA, which consists of partitioning each flow's packets into k sub-streams, and mapping

[1]In the final data structure described in Sec. 2.3.4, the hash value provided by h' is used, thus saving this extra hashing operation.

each packet to one of k different cells *in every row*. To coordinate both measurement nodes, the cell that a given packet will hit is also determined by a hash function, although this time of the full packet instead of only headers, so that successive packets of the same flow are scattered across the k cells.

This way, if one cell is hit by lost or reordered packets, only a subset of packets that belong to the flow are thrown away. The remaining cells to which the flows' packets are mapped will possibly remain intact, thus providing with reasonable estimates for the flow. To reduce the chances of losing cells to losses and reordering, we additionally place a packet sampling stage that will reduce the absolute number of packet losses, but also reduce the number of good estimates (similar to LDA). As analyzed in detail in [14], the LDA operates optimally when the sampling rate p is set to L/k, where L corresponds to the absolute number of losses in the packet stream. In Sec. 4.2 explores how to configure sampling rates in real deployments.

Now, we have two choices for the k cells. First, we can essentially replace each counter in the SDS data structure with k different cells. Second, we can allocate k (contiguous or random) cells in the counter matrix independently to each flow. If the k cells are contiguous, we can think of them as overlapping virtual LDAs (vLDA) per-flow, but they are not dedicated per-flow. In LDS we use this second approach, since it has the advantage of allowing a larger k without reducing the number of cells, thus increasing the robustness against loss at the cost of introducing extra collisions in the data structure.

2.3.4 Final Data Structure

The final data structure LDS is formally described as follows. LDS contains a matrix of R rows and C columns. Each cell of the matrix contains a tuple of values <s,n,d>, which keep aggregate timestamps, packet counts, and packet digests respectively. Both the sender and receiver maintain separate LDS copies, that is transmitted by the sender at the end of the measurement interval. Each row r has now *two* associated pseudo-random hash functions h_r and h'_r. While h_r returns a value in the range $[0, C-1]$, h'_r returns a value in the range $[0, k-1]$, where k is a configuration parameter of our algorithm that, as we shall see, represents the length of the virtual LDAs in our data structure. Once a packet that arrives at time t hits a cell, its s is increased by t, n is incremented by 1, and d is XORed with the digest of the new packet.

Update. When a packet with payload x and flow identifier f arrives at time t, for each row r, a position in the matrix given by $(r, (h_r(f) + h'_r(x)) \, mod \, C)$ is determined. That is, the flow hash is used to obtain a base position in each row, while the packet payload's hash determines an offset to that position in the range $[0, k-1]$. Thus, the packets of a given flow are randomly distributed among the neighboring cells. To coordinate this randomization between sender and receiver, they both use, again, the same pre-arranged (consistent) hash functions. Figure 4 presents a diagram of this scheme, while Fig. 5 formally describes this algorithm.

This scheme has the advantage of, while not introducing the full overhead of embedding a LDA in each cell, still obtains its advantages, by spreading the packets of each flow across several cells thus gaining protection against loss or reordering. If a flow experiences losses, they will invalidate some, but not necessarily all of the counters, which gives the algorithm a chance to recover its delay. As will be discussed in Sec. 3, this feature increases the amount of collisions. Therefore, to support the same number of flows, it still has to be larger than the basic data structure presented in Sec. 2.

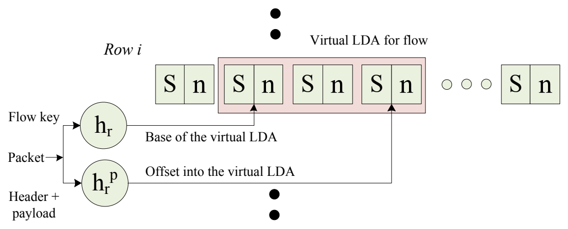

Figure 4: Virtual LDA extension to the data structure (d fields are not depicted).

```
 1: procedure UPDATE STATE($pkt$, $f$, $\tau$)
 2:     $ph \leftarrow hash\_packet(pkt)\%k$          ▷ $k$ is vLDA size
 3:     for i=1, R do
 4:         $fh \leftarrow (hash(i, f))$
 5:         $j \leftarrow ((fh + ph)\%C)$
 6:         $LDS[i][j].S \leftarrow LDS[i][j].S + \tau$
 7:         $LDS[i][j].N \leftarrow LDS[i][j].N + 1$
 8:         $LDS[i][j].D \leftarrow LDS[i][j].D \oplus pkthash$
 9:     end for
10: end procedure
```

Figure 5: LDS – Per-packet operations

Delay Estimation. When producing an estimate of a given flow, all associated usable vLDA cells are initially selected. After this step, the question of how to estimate flow delays arises. The algorithm now has to choose among the usable cells to produce an estimate. In the event that, for a flow, none of its cells are invalidated, it has $R\,k$ cells that can produce delay estimates (for each row, all cells of the flow's vLDA).

Again, several strategies could be used to select which cells are going to be used for estimation. For example, one could aggregate all usable cells of each row into a single one, thus obtaining one candidate delay estimation per row and, like in the previous data structure, choose the one with the least amount of packets.

Such a strategy is impractical, because it unnecessarily gives up the advantages of having the packets spread across several positions in the data structure. Instead, it is beneficial to selectively discard specific positions with high measurement interference.

We thus adopt the following strategy. First, among all cells, we choose the one with the smallest number of packets. Assuming that each vLDA cell contains $1/k$th of the packets of the measured flow, this is the cell that has experienced least colliding packets. Let the number of packets aggregated in this cell be n. Then, from the rest of the cells, we select those that contain, at most, $n\,(1 + \alpha)$ packets, where α is a configuration parameter that reflects a maximum percentage of tolerable interference. Too large an α leads to the inclusion of interfering packets, while setting it too small discards valid samples. We empirically found

```
1: procedure Delay Estimation($f$, $L1$, $L2$)
2:     $N_{min} = \infty$
3:     for i=1, R do
4:         $fh \leftarrow hash(i, f)$
5:         for $j = fh, (fh + k)\%C$ do        ▷ $k$ is vLDA size
6:             if L1[i][j].N == L2[i][j].N &&
7:                     L1[i][j].D == L2[i][j].D then
8:                 $cell = \{L2[i][j].S - L1[i][j].S, L1[i][j].N\}$
9:                 $S = S \bigcup cell$        ▷ Stores all valid cells
10:                $N_{min} = min\{N_{min}, cell.N\}$
11:            end if
12:        end for
13:    end for
14:    for $cell \in S$ do
15:        if $cell.N < (1 + \alpha)N_{min}$ then
16:            $S_{sum} += cell.S$
17:            $N_{sum} += cell.N$
18:        end if
19:    end for
20:    return $S_{sum}/N_{sum}$
21: end procedure
```

Figure 6: LDS – Delay estimation algorithm

$\alpha = 0.1$ to represent a good trade-off between these two factors in our setting.

Then, the resulting set of cells are aggregated to produce a final estimate. Since, within each vLDA, packets are randomly distributed across cells, the possibility of double counting packets exists. Note that this is not problematic for the measurement of average delay.

The pseudocode in Fig. 6 captures our delay estimation mechanism more formally. Here $L1$ and $L2$ are the sender- and receiver-side LDSes. Note that k refers to the configured vLDA size for each flow. Also, we assume both $L1$ and $L2$ have already been updated with the same hash functions $hash(i)$, $i = 1 \ldots R$.

3. ANALYSIS

In this section, we analyze our data structure and provide guidelines on how to dimension it in order to obtain the desired level of accuracy. We start with the analysis of SDS and then extend it to the more general case of LDS.

3.1 Simple Delay Sketch

One could hope to dimension the data structure so that measurements are error-free with high probability, i.e., flows are highly likely to be free from collision in, at least, one of the cells. Unfortunately, this would require a great deal of memory since, for example, when using a single row with as many counters as flows, the probability that a given flow is collision free is only e^{-1} (as in a standard Bloom filter with a single hash function). In order for this probability to grow beyond 95%, we require a number of counters that is well above of the number of flows we want to measure.

As explained, in our data structure, colliding flows cause interfering measurements, and estimates produced by each row are an average delay weighted by the number of packets of the colliding flows. In other words, in practice, larger flows tend to have less error, since they will most often collide with small flows, rather than larger ones.

In order for a flow to be accurately measured, the total number of packet in flows that collide with it must be sufficiently small so as not to significantly impact its delay estimate. Specifically, we say that a flow suffers *only small collisions* if the number of colliding packets is no more than some threshold x. To compute the probability Q of small collisions we consider first the number of flows colliding with a given flow, and then the number of packets that they bring. Letting K_i denote the probability of i colliding flows, and S_i the probability that these i flows bring no more that x packets in total, then $Q = \sum_{i=0}^{n} K_i\, S_i$, assuming n background flows. By definition, $S_0 = 1$, since a flow is always correctly measured when it is free from collision.

We aim to dimension the data structure in a way that the probability Q of only small collisions stays high. Assuming a uniform distribution of hash values, K_i is probability of obtaining i items under the Binomial distribution $B(n, 1/C)$, yielding a mean number n/C of colliding flows. To compute S_i we need to assume some distribution of flow sizes. We will assume Pareto distributed flow sizes. This distribution is often used to model flow sizes (e.g., [15, 23]) and can be fitted to the characteristics of the network data we use in the evaluation in Sec. 5. The sub-exponential property of the Pareto distribution implies that, if T_k is the sum of the sizes of k flows, then $\Pr[T_k > x] \approx k \Pr[T_1 > x]$ for large x at fixed k. In other words, when T_k is very large, this tends to be because one entry in the sum is very large, not because several are moderately large. Using this property, we obtain $Q \approx \sum_{i=0}^{n} K_i(1 - iP) = 1 - \frac{n}{C} * P$, where $P = \Pr[T_1 > x]$ under the (fitted) Pareto distribution. The accuracy of approximation increases for small n/C and large x.

Using this analysis, we can adapt the size of our data structure to obtain the desired probabilistic accuracy bound. For example, using Pareto parameters that match our traffic (see Sec. 5), we obtain a probability $Q \approx 91\%$ of small collisions comprising at most 50 packets, with half as many counters as flows ($n/C = 2$), structured in 1 row. This means that, for example, flows with 1000 or more packets have a 91% probability of small collisions comprising no more than 5% of their packets. Incidentally, a numerical computation of Q without the subexponential approximation differed by a few tenths of absolute percent in this example.

The formulation of this example illustrates a key requirement to measure a flow accurately: not only must it suffer only small collisions but the flow itself must be large. (How large depends on the delay distribution.) To simplify the analysis, we stipulate a large flow to be one with at least x packets, where x is the threshold total packets for small collisions. With this formulation, we call a flow *survivable* in storage if it is both large, and suffers only small collisions. What then, is the maximal storage capacity of survivable flows? Suppose n flows are stored. The average number of large flows is nP and so the average number of survivable flows is $nPQ = nP(1 - nP/C)$. This expression is maximized at $n = C/(2P)$, yielding $C/4$. This is reminiscent of the collision free capacity C/e of the standard Bloom filter. The difference is that the proposed structure can store up to $C/4$ *survivable* flows out of a potentially far larger $C/(4P)$ that are presented for storage.

In fact we do not expect the operating regime to accommodate the maximal number of flow because the probability of *large* (i.e., not small) collisions is $1 - Q = nP/C = 1/2$. We now investigate operating regimes with rare large collisions in the generality of multiple $R \geq 1$ rows. We assume the primary design aim is to limit the probability of large collisions, with a secondary aim of maximizing the number of survivable flows under that constraint. In the case of multiple rows, a large flow is survivable if it has small collisions in at least one row. With R rows, the total resources C are divided up evenly between rows, and so substituting C/R for C in Q, the relevant survival probability is $Q(R) = 1-(1-Q)^R = 1-(nP/CR)^R$. For a cleaner analysis it is convenient to change variables from n to $z = nP/C$, which can be thought of as the offered load of large flows per unit storage. Then $Q(R) = q(z,R) := 1-(zR)^R$.

As a function of R for fixed z, $q(z,R)$ is maximized at $R = 1/(ez)$. But only $R \geq 1$ are physical. (In this analysis we omit consideration of integrality; in practice we round to an integer at the end). Thus $\max_{R\geq 1} q(z,R) = q(z) := q(z, \max\{1, 1/(ez)\})$. $q(z)$ is a decreasing function of z which takes the value $1-z$ for $z > 1/e$ (corresponding to $R = 1$) and $1-e^{-1/(ez)}$ for $z \leq 1/e$ (corresponding to the case $R > 1$). Assuming we wish a small probability $\varepsilon < 1/e \approx 0.63$ of large collisions, then we should be in the small $z < 1/e$ regime, so we would want to chose z such that $\varepsilon > e^{-1/(ez)}$, which corresponds to the choice $R = -\log(\varepsilon)$, modulo discretization, then making sure the offered load z is less than $z_{max} = -1/(e\log(\varepsilon))$. The relative gain of allowing multiple rows can be seen as follows: under the constraint $R = 1$, achieving the same bound on the probability of large collisions would require $z = 1-q(z,1) \leq \varepsilon$. Hence allowing $R > 1$ allows us to increase the offered load by a ratio $-1/(e\varepsilon\log(\varepsilon)) > 1$ for target $\varepsilon < 1/e$. Conversely, maintaining the same load achieves a dramatic reduction in the frequency of large collisions. In the previous example $z = nP/C = 0.0875$, so we are in the regime $z \leq 1/e$, leading to optimal $R = 4.20$. Rounding to the nearest integer $R = 4$, we obtain $q(4,z) = 0.9850$, as compared with the previous $q(1,z) = 0.91$.

3.2 Lossy Difference Sketch

The introduction of the Virtual LDAs in the LDS has several side effects. The principal consequence of further spreading flow packets across the data structure is that fewer positions remain unused and, more importantly, more collisions occur. However, this is to some extent compensated by the fact that every flow is spread across k positions, and, thus, collision randomization is higher. In this section, we will investigate how these factors change the previous analysis.

The Virtual LDA divides up the packets of a flow amongst k locations in each of R rows. Accurate estimation of a given flow depends on having only small collisions in at least one of these locations. Thus we adapt our notion of survivability as follows for general k: A given flow is *survivable* if it is large (the number of packets exceeds some value x) while at the same time suffers only small collisions (of size no more than x/k) at at least one of the Rk locations it occupies in the Virtual LDA.

In this section we examine a simplified model of the Virtual LDA that admits an extension to the analysis of Section 3.1 to approximate the probability of survivability. This shows that, from the collision survivability point of view, the Virtual LDA is no worse than the multirow data structure described in Section 3.1, and is actually expected to be better. This property, coupled with the superior loss resilience of the Virtual LDA, recommends it as the better choice.

Our model and analysis are as follows. For a specific flow, let U_ℓ be the number of its packets hashed to a location ℓ, and V_ℓ the number of packets from all other flows that are mapped to that location. The estimation algorithm first determines the location ℓ of minimal $U_\ell + V_\ell$. Since we are concerned principally with the case that the specific flow has some large number u of packets, our first simplification is to ignore the sampling variability amongst the U_ℓ and approximate the U_ℓ as taking the same value (i.e., the average u/k). Thus the problem of minimizing $U_\ell + V_\ell$ is thus reduced to that of minimizing the V_ℓ.

Because the locations allocated to a given flow in a row are contiguous, the V_ℓ are in general dependent, because if packets from a background flow hash to location ℓ, the other packets from the same flow are more likely to collide at a neighboring location ℓ'. This dependence leads to positive correlations amongst the V_ℓ, meaning that the joint probability of collisions occurring at all locations of a flow is greater than the product of the marginal probability of collisions occurring at each site. Conversely, the corresponding survival probability is bounded below by that of a model where collisions are independent: it is conservative to use this as our second simplification. Thus we model the distribution K_i of the number of colliding flows as a Bernoulli $B(nk, R/C)$ random variables.

For our final simplification, we note that under our Pareto model, the probability that the number $T(k)$ of packets sampled from a background flow to each of the k locations in a row exceeds a level x obeys $\Pr[T(k) \geq x] \approx \Pr[T_1 \geq kx]$ for large x, where T_1 the length of the background flow; see [26]. Coupled with the subexponential approximation for sums of flow lengths, we approximate the probability of a small number of packets (at most x/k) due to i colliding flows at some site ℓ as $S_i = \Pr[V_\ell < x/k|\text{i colliding flows}] \approx 1 - iP$ where $P = \Pr[T_1 > x]$ under the (fitted) Pareto distribution.

Thus, under our simplifications, the probability of small collisions is $Q(Rk) = 1-(nkRP/C)^{Rk}$. The optimization and dimensioning strategy is then immediate by comparison with Section 3.1: (i) choose a value k based on targets for loss resilience; (ii) for a given small target survivability rate $1-\varepsilon$, calculate the number of rows by rounding $\max\{1, -\log(\varepsilon)/k\}$ to the nearest integer.

4. PRACTICAL ENHANCEMENTS

In this section, we introduce a series of enhancements to the LDS data structure that make it more practical for real deployment. We start by defining a mechanism to boost the accuracy for a selected subset of flows of particular interest to network operators. We then investigate how to parametrize the sampling rates to support a wide range of loss ratios. Finally, we note that the data structure contains additional information that can be used to mine other interesting metrics, including per-flow packet loss and heavy hitter detection.

4.1 Weighting of Flows

As formally analyzed in Sec. 3, the LDS intrinsically produces better estimates for large flows. This is due to the fact that, when flows collide, estimates are average delays of said flows, weighted by the amount of packets.

However, often times, small flows are of interest (a notable example are DNS flows, which usually consist of only one packet per direction). On the other hand, operators might be particularly interested in measuring a particular set of flows with higher accuracy. For example, one could increase the accuracy for certain subnetworks where critical services are hosted, or where troubleshooting activities call for precise examination of network delays (a practical use case for flow weighting is presented in Sec. 5).

We provide a mechanism to raise the accuracy of flows at will. This can be very simply accomplished by weighting flows according to some pre-defined policies driven by the operator's desires. Such policies can define a flow's weight according to any information present on packet headers. The default weight for non-policed flows is defined as 1 for simplicity.

These weights are taken into consideration straightforwardly by slightly modifying the update procedure. When a packet of a flow f arrives, its headers are examined and a weight w is determined according the existing weighting policies. Then, a cell of each row is selected as explained in Sec. 2.3.4. For each of the cells, their s value is increased by w times the packet timestamp, while n values are increased by w (recall that, previously, s values were increased by the timestamp and n values by 1).

No modification is required to the estimation procedure. When all flow weights are equal, the estimates are identical to those of Sec. 2.3.4. Otherwise, flows are weighted by their number of packets times their weight. It shall be noted, however, that this extra accuracy will always come at expense of the accuracy of the estimates for flows with lesser weight. Thus, it is not advisable to heavily increase the weight of a large percentage of flows, as it will dramatically reduce the accuracy for all the others. We will analyze the effect of weighting from a practical standpoint in Sec. 5.

4.2 Multi-Bank LDS

As explained in Sec. 2.3.3, to maximize the collection of delay samples in front of packet loss, each vLDA should sample the incoming packet stream at rate L/k, where L corresponds to its associated number of losses and k to its length. However, in a real scenario, the absolute number of losses that each flow will experience is unpredictable, which raises the question of how to set the sampling rate. On the one hand, a reasonable amount of loss has to be supported, which calls for low sampling rates. On the other, aggressive sampling will miss small flows and fail to collect enough packets for those that do not experience loss.

Inspired by [17], we propose dividing the counters of the LDS in several banks, and have each bank sample the incoming packets at a different rate. This way, at least one of the banks will suit the actual loss rate of each flow.

A natural way to divide counters in banks is to set a different sampling rate on a per-row basis. In the evaluation provided in Sec. 5, we show that configuring one row for worst-case loss scenarios, and maintaining increasingly higher sampling rates in the other rows, does not sacrifice accuracy in normal scenarios with low loss, while still providing protection against high loss. When a flow surpasses the target worst-case threshold, the LDS will be unable to provide delay estimates for that flow. In such a case, however, the data structure can provide an estimate for its number of lost packets, as will be explained in Sec. 4.3.

This variant of the LDS requires very few changes to the algorithms detailed in Sec. 2.3. Now each row has an associated sampling stage, which is also implemented using pseudo-random hashing to coordinate measurement nodes, while the estimation procedure only needs to be modified to be aware of the sampling rate that each cell has applied. In particular, packet counts need to be inverted before deciding which LDS cell will be selected to produce a final estimate. After the cell selection procedure, delay estimates are produced normally.

4.3 Mining Other Estimates

The LDS data structure can be mined to extract additional information of practical interest to network operators. Firstly, the data structure can provide per-flow delay variance estimates by examining the differences across the delays recorded in buckets dedicated to a given flow. The procedure to obtain this estimate was originally proposed and is thoroughly described in [17]. Our data structure has a comparatively smaller number of buckets per flow, but the same method could be applied to obtain rough delay variance estimates. This additional estimate can be extremely useful in practice to detect unexpected delay variations, such as jitter or delay peaks.

If we ignore the s fields and focus only on the n fields of each cell, the data structure behaves similarly to a Count-Min Sketch [9]. Consequently, an estimate of the length of a given flow can be obtained as follows. For each row, aggregate all vLDA cells to obtain a packet count. Then, take the minimum of such values. This is the final estimate. It can be easily shown that this estimate is, at best, error free, and, in the presence of collisions, it can only be greater than the actual value. This is an interesting property for certain problems and, especially, for heavy hitter detection. The accuracy of this technique is analyzed in greater detail in [9].

Simply by attaching another counter to each cell, where packet sizes are aggregated, we can also estimate flow sizes in bytes. Both these new counters and the existing could be used for heavy hitter detection in terms of bytes or packets respectively. Additionally, one could obtain crude estimates for the average packet sizes. In this paper we divide the aggregate delay over the number of packets to estimate average flow delays. Likewise, total flow sizes could be used to obtain a per-flow average packet size. Finally, we note that per-flow packet loss can be obtained by comparing the n fields of our data structure as collected in *sender* and *receiver*.

5. EVALUATION

With the objective of evaluating the LDS data structure in a realistic scenario, we deployed two network monitors in an operational network. For the sake of reproducibility, we collected a packet delay trace, rather than directly processing live traffic. We then ran a series of experiments using LDS and two state of the art techniques that will be described in Sec. 5.1. We note, however, that all the traffic measurement procedures, including LDS and both reference techniques, were fast enough to run on-line and, therefore, the results we present are completely equivalent to live traffic analysis.

The first monitor was deployed in a 10 Gb/s link that connects a large research and educational networking consortium to the rest of the Internet. The second was located in the 1 Gb/s access link of one University part of this consortium. We obtained a copy of the traffic that traversed both links in both directions and used Endace DAG

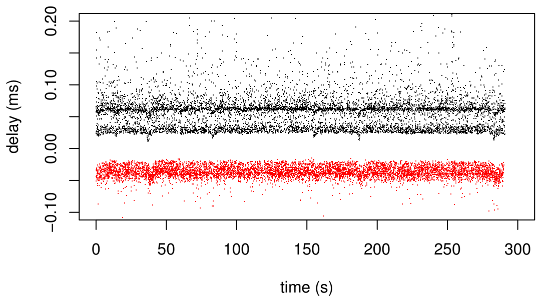

Figure 7: Timeseries of a sample of the packet delays, with outbound delays portrayed as negative values.

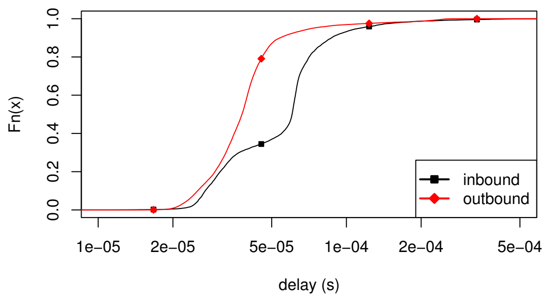

Figure 8: CDF of the delays for inbound and outbound packets.

cards [12] to simultaneously capture packets in both measurement points. We synchronized DAG clocks using the PTP protocol, which reportedly provides sub-microsecond accuracies [10] and thus is accurate enough for fine grained delay measurement.

We then wrote a CoMo module [4] to analyze the trace and extract, for each packet, its flow identifier, packet identifier, and exact one-way delay. The trace averages 27Kpkts/s and contains around 7.76 million packets that belong to approximately 146000 flows.

Figure 7 shows a time series of a sample of the packet delays for each traffic direction. Two features are apparent from this figure that make inbound traffic (i.e., destined towards the University network) more interesting. First, inbound packet delays present higher variability. Second, two delay modes are clearly appreciable in the inbound traffic, as can be also confirmed in the CDF of the packet delays presented in Figure 8. Therefore, unless otherwise noted, the experiments presented next in the evaluation are performed on the inbound traffic.

5.1 Comparison with Existing Methods

The objective of this section is to compare the accuracy of LDS with the state-of-the-art on per-flow delay measurement. We choose the NetFlow Multi-Point Estimator (MPE) [19] and the Reference Latency Interpolation (RLI) [18] as representatives of a recently introduced class of techniques that exploit temporal delay correlation to refine measurements from a few samples.

The Multi-Point Estimator is conceived as an extension to NetFlow, and requires routers to use coordinated sampling. Under this assumption, for each sampled flow, there exist two delay samples from which to estimate the flow delay (NetFlow records include a timestamp of the first and last packet). Additionally, based on the empirical observation that packets that travel close in time experience similar delays, the method can interpolate the delay between these two samples using the NetFlow records of other flows that start or end within the duration of the measured flow. The main difference between MPE and RLI is that, while MPE relies on a modified version of NetFlow, RLI injects active probes to obtain the necessary delay samples and assumes that packets between two probes experience the same delay.

We evaluate LDS with three different configurations: one that provisions half as many counters as flows ($n/C \approx 2$) (to obtain a configuration that, as will be discussed, is comparable with MPE), while the other two are 10 times larger and smaller than this reference LDS. Consistently with the example in Sec. 3, we structure the sketch in 4 rows, which yields a sketch of 17500×4 counters for the first configuration. Given that loss is negligible in our scenario, we set the vLDA length $k = 1$, and the sampling rate $p = 1$. We analyze the impact of loss in detail in Sec. 5.2.

Figure 9 plots the CDF of the relative error obtained by each flow in the traffic, for different flow sizes. The figure includes the accuracy of MPE with a sampling rate of 1% and 10%,[2] RLI with 1KHz probing, and a simple method that estimates the delay of each flow to be the average delay of all packets. The figure shows that LDS greatly outperforms both MPE and RLI. The increase in accuracy compared to MPE can be explained by two primary causes. First, MPE completely misses a large number of small flows (e.g., more than 50% with 10% sampling). For these flows, we estimate their delay as the average delay of all packets, instead of simply assigning an error of 1. In contrast, LDS can always obtain an estimate for all flows. Second, for the flows it does collect, it interpolates the delays using other flows, but in our case these are not necessarily correlated, as can be observed in Figure 8. While RLI outperforms MPE, its accuracy is also far from LDS, especially for large flows, and requires significantly more memory and state maintenance.

Figure 9 (left) shows that, as predicted by the analysis, large flows are very accurately measured. A still notable accuracy for flows of 100 or more packets is also observed in the middle plot. Figure 9 (right) shows the per-flow accuracy for all flows, including also those with less than 100 packets. According to the analysis in Sec. 3, these flows are not considered to be survivable, since they tend to experience large collisions. These flows however only account for 20% of the packets in our trace. Even in this case, the accuracy of LDS is consistently above the state of the art. This result shows that the estimate of LDS for unsurvivable flows is in practice more accurate than just using the average delay of all packets.

LDS also features better memory usage. For example, with 10% sampling, MPE captures around 70000 flows, so (generously disregarding the fact that NetFlow stores flow keys) it consumes roughly as much memory as the 17500×4 LDS. Thus, with the same memory budget, LDS clearly outperforms MPE in terms of measurement accuracy. Note also that, even when LDS uses 10 times less memory than MPE ($C = 1750 \times 4$), it obtains significantly higher accuracy, especially for medium sized to large flows. LDS also outperforms

[2]Note that MPE uses sampling to control the memory usage, while for LDS sampling is only a measure against packet loss.

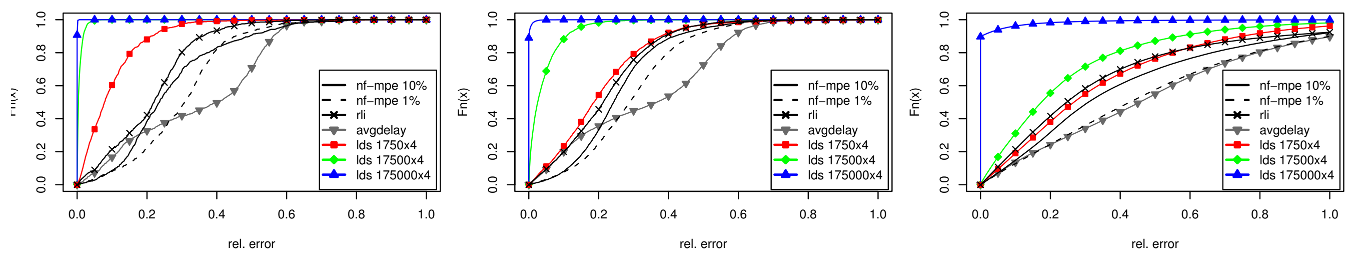

Figure 9: CDF of the relative error of various measurement methods for flows with > 1000 pkts. (left), with > 100 pkts (center) and all flows (right).

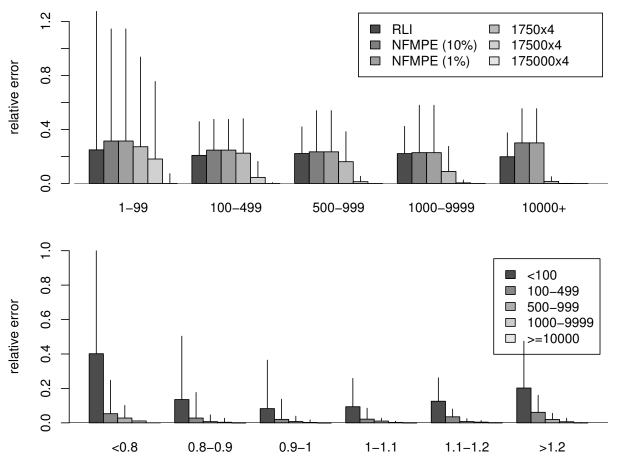

Figure 10: Median and 95-pct of relative error, binning flows by number of packets (top) and number of packets and average delay (relative to avg. flow delay; bottom).

RLI, which requires even more memory than MPE, since it maintains per-flow state.

Figure 10 (top) plots the median and 95-percentile of the relative error of flows binned by size. Consistently with the analysis of Sec. 3, the figure shows that larger flows are more accurately measured. Our method compares extremely favorably to both MPE and RLI. Only with 7,000 counters we obtain significant improvements for flows larger than 1,000 packets. When using 70,000 counters, which take about 1MB, flows with 500 to 999 packets obtain 1.2% median relative error, which falls to 0.4% and 0.06% for the larger size bins.

Figure 10 (bottom) bins flows both by size and average delay with a fixed sketch size of 70,000 counters. The figure shows how the errors are slightly larger as delay deviates in extreme values for the mean. This happens because smaller flows show more extreme values, but interferences tend to drag measurements toward the mean.

5.2 Measurement under Packet Loss

We now analyze the effect of packet loss to our data structure. Under small loss rates, it is desirable to keep the vLDA length parameter k small, since increasing it increments the number of collisions in the sketch. However, increasing k provides higher protection against loss. Additionally, sampling helps contain loss, since a large number of losses in a single flow can potentially invalidate the k counters.

We wish to dimension our data structure to support a given maximum number of losses per flow. The main intuition behind this approach is that, when flows experience a large amount of losses, performance degradation is more a consequence of loss than delay; thus, delay measurements cease to be meaningful (note that LDS can be mined to estimate per-flow loss, as explained in Sec. 4.3).

In this experiment, we arbitrarily set a target number of losses of 500 packets per flow. However, we also wish the LDS to be able to capture a large sample size if losses are much lower. Thus, we use a multi-bank configuration of LDS, as described in Sec. 4.2. We provide 4 rows with $k = 5$ vLDA buckets, like in the previous scenario, increase the size of each row of the sketch by a factor of k, and set $\alpha = 0.1$. We then pick suitable packet sampling rates for each row. With $k = 5$, each vLDA cell needs to support 100 losses, according to our target number of losses. This means that the sampling rate should be set to 0.01 to support this worst-case loss. Then, we wish the rest of the banks to tolerate lower loss in order not to sacrifice the accuracy of LDS in the normal case. We set the rest of the banks with increasing sampling rates of 0.1, 0.5 and 1 to tolerate up to 50, 10 and 5 losses respectively. For comparative purposes, we also set two LDS with a fixed sampling rate in all rows of 0.05 and 1.

Since, in our scenario, losses are negligible, we introduce random, uniform loss to test such configurations. Consistently with the assumptions made in Sec. 2.1, we perform 3 series of experiments with loss rates 0.1%, 0.5% and 1%. It should be noted that uniform loss is one of the most harmful loss models to LDS, for three main causes. First, losses are spread among a large number of flows, instead of being contained within a few. Second, the absolute number of losses that hit each bucket heavily varies according to the lengths of the involved flows. Recall from Sec. 2.3 that the optimal sampling rate for each bucket depends on its absolute number of losses. Third, this loss model penalizes large flows, which are precisely those that our method can measure most accurately.

Figure 11 shows the results we obtained. We start by noting that neither 5% nor 100% sampling single-bank LDSs perform satisfactorily. The former maintains its accuracy under higher loss, but is too conservative and underperforms on low loss. Conversely, 100% sampling is too optimistic and does not offer protection against loss. Thus, its cells become

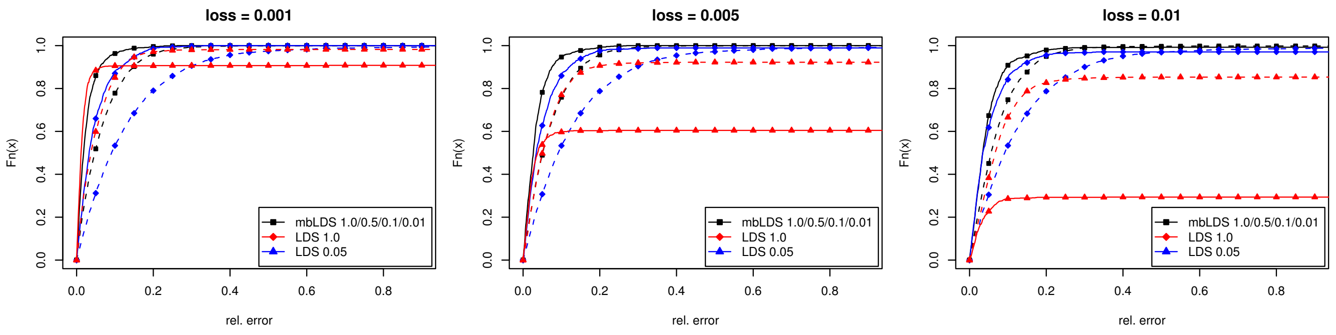

Figure 11: CDF of the relative of several LDS parametrizations under varying uniform loss rates. Solid lines represent flows with at least 10^3 packets, while dashed lines, flows with more than 100 packets.

too quickly invalidated under increasing loss, causing measurements to be lost.

In contrast, the multi-bank LDS performs consistently well under all loss rates. This desirable behavior is a consequence that, in all three scenarios, most flows experience losses that are well tolerated by at least one of the banks. Therefore, very seldom a flow invalidates all of its buckets, and accurate measurements are always produced.

The accuracy of LDS in Fig. 11 is still above that of RLI and MPE without loss (as presented in Sec. 5.1). However, MPE and RLI are more robust to loss. Hence, in scenarios with high loss and temporal correlation, we expect MPE and RLI to be a better choice.

5.3 Flow Weighting

We now test a more realistic use case of our technique. We envision a data center that hosts network services for a series of customers. Not all customers, though, are equally sensitive to network delay. We group the hosted services in three classes. First, bronze customers are not overly concerned about packet delays in the data center. For example, those could include bulk data transfer applications, such as backup, static web content serving, e-mail relaying, or computing intensive tasks.

Second, silver customers are somewhat dependent on network delay, but they do not require strict compliance of low-delay QoS requirements. A class that would fit these well are interactive services, such as remote shells, highly interactive web applications (e.g., Google is known to seek low delay to enhance the user's browsing experiences of AJAX-powered web applications), or web services for third party applications.

Finally, gold customers host applications that are extremely sensitive to delay, and wish to closely track the QoS of the services they are offering. Perfect examples for this class of applications involve multi-media streaming, audio/video conferencing, or remote gaming. Financial services such as automated trading could also fit this category, although, given that, in their case, low-delay data transmission is critical, they are unlikely to be hosted in shared infrastructure.

Since we do not have access to network traffic from a data center that hosts such applications, we adapt our scenario as follows. We randomly assign each flow to one of the categories. Bronze customers take 90% of the flows; silver customers, 9%, and gold customers, 1%. This approach ensures that the results are not an artifact of flow sizes, since large flows tend to be more accurately measured. In a real setting, these weights can be adjusted to the specific characteristics of the traffic under measurement.

We experiment with different sizings of the data structure, and various weights for each of the customer classes. Figure 12 shows the result of a series of experiments. We have tested two reference configurations. One that uses 40,000 counters (first row), an another that uses 100,000 counters and provides greater accuracy (second row). The first configuration fits in 625KB, and the second, in around 1.5MB. As for flow weights, we have tested three different configurations, which can be observed in each column: assigning weights of 1, 25 and 100 (first); 1, 10 and 100 (second), and, 1, 50 and 2500 (third).

The figure shows the relative error for the full set of flows (solid lines) and only for flows with more than 100 packets (dashed lines). Besides the error of each customer class, the figure also shows, as a reference, the result of applying no weighting. An important observation to be made is that flows from classes that have higher weights obtain significantly greater accuracy. The accuracy boost greatly depends on the actual weights; for example, when gold customers carry weight 2500, they obtain extreme accuracy. However, this penalizes the accuracy of bronze customers. In this case, we have ensured that the accuracy of bronze customers is not highly penalized, because few flows belong to higher priority classes. If, otherwise, the number of flows in each class was more balanced, the only option to increase accuracy for the flows of a given class without significantly diminishing that of a lower priority class would be to increase the sketch size. In other words, this method is only applicable to increase the accuracy of a small subset of flows.

6. RELATED WORK

One-way packet delay has been measured both using passive and active schemes. Active monitoring methods (e.g., [5, 24, 6, 28]) are based on injecting probe traffic in the network under study, and inferring one-way delay from the delays incurred by such probes. In contrast, passively monitoring network delays has been traditionally accomplished by recording packet timestamps in two measurement points, and exchanging such timestamps for comparison. Because these techniques generate huge data volumes, they require aggressive sampling to reduce the overhead. Further, packet sampling has to be coordinated across nodes, since the timestamps recorded at both measurement points must corre-

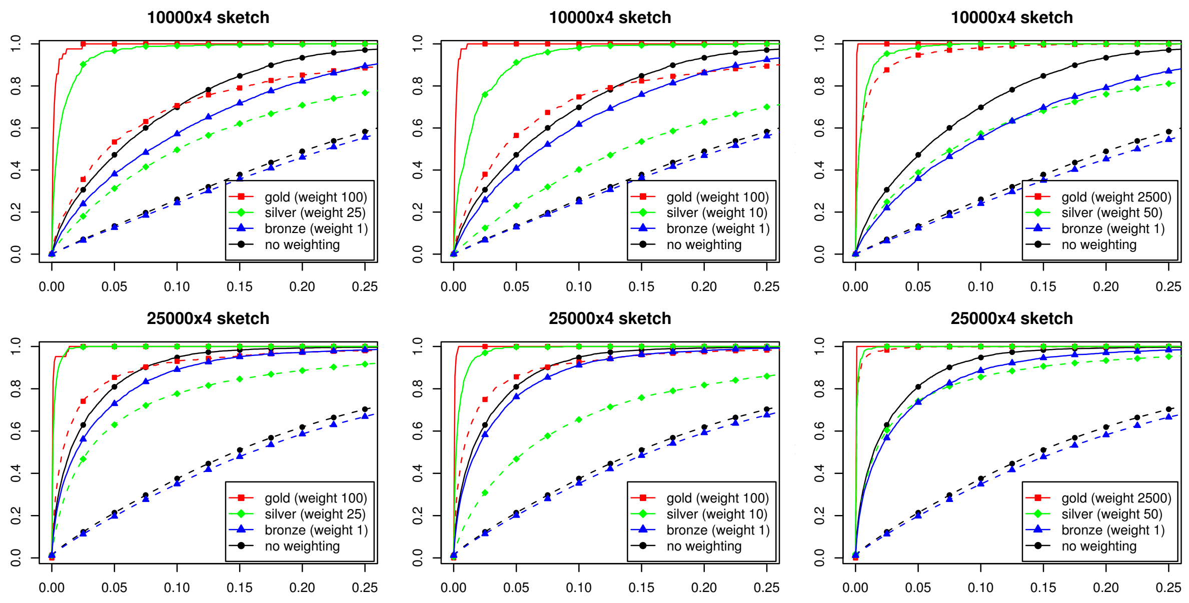

Figure 12: CDFs of the relative error with various flow weights and sketch sizes. Solid lines correspond to flows with more than 100 packets, while dashed lines include the full set of flows.

spond to an equal subset of all packets. This effect can be achieved using consistent hashing, i.e., using same pre-arranged hash function—an idea used before in trajectory sampling [11]).

More recently, LDA [17] has been proposed as a mechanism to overcome the linear relationship between sample size and overhead. LDA has been further analyzed in [27, 14] as well. Since our paper borrows some of the ideas of LDA, we have discussed this idea in great length. The problem of obtaining per-flow latency estimates in a scalable fashion, which is exactly the problem we attempted to solve in this paper, has received recent attention [18, 19]. [19] proposes modifying NetFlow [7] to allow measurement of one-way delay. If NetFlow samples packets using consistent hashing, the first and last timestamp fields of NetFlow records can be used to obtain two delay samples of a given flow that can be refined from using samples from other flows in between these two timestamps. The core idea of temporal delay correlation forms the basis for Reference Latency Interpolation [18], that we also discussed in detail in the paper. The biggest difference between RLI and our work is that we do not assume temporal correlation of packet delays. Removing the dependence on this assumption is beneficial in many ways, as explained in Sec. 1.

Besides packet delay measurement techniques, sketching is also very relevant to this work. Most relevant to us are two similar data structures: Multi-Stage Filters [13], which are designed for elephant flow detection, and the Count-Min Sketch [9], which can provide per-flow estimates with probabilistic accuracy guarantees. Other sketching techniques are reviewed and compared in [8].

7. CONCLUSIONS

We have presented a sketch-based data structure capable of producing per-flow one-way delay estimates. Although sketching naturally produces the best estimates for larger flows, this data structure can enhance the accuracy of arbitrary flows. For measurement in networks with packet loss, we have combined our sketching technique with a recently appeared data structure called Lossy Difference Aggregator.

State-of-art techniques rely on temporal correlation of delays to produce their estimates. However, in practice, routers can use various queueing policies for different kind of traffic, which greatly reduces the effective of said techniques. In our evaluation, we show how our technique achieves higher accuracy than such techniques when using a similar amount of memory, even in the presence of packet loss.

We have also presented a practical deployment scenario where our technique and its ability to improve measurement for arbitrary flows could be very useful. In particular, our technique could very well cater a data center with shared resources, where various applications present diverse degrees of dependency on network delay. In such a scenario, our technique can be used to obtain extremely precise measurements for the most critical applications, while still providing an acceptable degree of accuracy for other applications.

8. ACKNOWLEDGMENTS

This research was funded by the Spanish Ministry of Science and Innovation under contract TEC2011-27474 (NOMADS project), and by the *Comissionat per a Universitats i Recerca del DIUE de la Generalitat de Catalunya* (ref. 2009SGR-1140).

9. REFERENCES

[1] Corvil. `http://www.corvil.com/`.
[2] Juniper Networks T series Core Routers Architecture Overview. `www.juniper.net/us/en/local/pdf/whitepapers/2000302-en.pdf`.
[3] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and

M. Sridharan. Data center TCP (DCTCP). In *Proc. of ACM SIGCOMM*, 2010.

[4] P. Barlet-Ros, G. Iannaccone, J. Sanjuàs-Cuxart, D. Amores-López, and J. Solé-Pareta. Load shedding in network monitoring applications. In *Proc. of USENIX Annual Technical Conf.*, 2007.

[5] J. Bolot. Characterizing end-to-end packet delay and loss in the internet. *Journal of High Speed Networks*, 2(3):289–298, 1993.

[6] B. Choi, S. Moon, R. Cruz, Z. Zhang, and C. Diot. Practical delay monitoring for ISPs. In *Proc. of ACM CoNEXT*, 2005.

[7] Cisco. NetFlow. http://www.cisco.com/web/go/netflow.

[8] G. Cormode and M. Hadjieleftheriou. Methods for finding frequent items in data streams. *The VLDB Journal*, 19(1):3–20, 2010.

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

[10] L. De Vito, S. Rapuano, and L. Tomaciello. One-way delay measurement: State of the art. *IEEE Transactions on Instrumentation and Measurement*, 57(12):2742–2750, 2008.

[11] N. Duffield and M. Grossglauser. Trajectory sampling for direct traffic observation. *IEEE/ACM Transactions on Networking*, 9(3):280–292, 2001.

[12] Endace. DAG network monitoring cards. http://www.endace.com.

[13] C. Estan and G. Varghese. New Directions in Traffic Measurement and Accounting: Focusing on the Elephants , Ignoring the Mice. *ACM Transactions on Computer Systems*, 21(3):270–313, 2003.

[14] H. Finucane and M. Mitzenmacher. An improved analysis of the lossy difference aggregator. *ACM SIGCOMM Computer Communication Review*, 40(2):4–11, 2010.

[15] S. Fred, T. Bonald, A. Proutiere, G. Regnie, and J. Roberts. Statistical bandwidth sharing: a study of congestion at flow level. In *Proc. of ACM SIGCOMM*, 2001.

[16] IEEE. IEEE/ANSI 1588 standard for a precision clock synchronization protocol for networked measurement and control systems, 2002.

[17] R. Kompella, K. Levchenko, A. Snoeren, and G. Varghese. Every microsecond counts: tracking fine-grain latencies with a lossy difference aggregator. In *Proc. of ACM SIGCOMM*, 2009.

[18] M. Lee, N. Duffield, and R. Kompella. Not all microseconds are equal: fine-grained per-flow measurements with reference latency interpolation. In *Proc. of ACM SIGCOMM*, 2010.

[19] M. Lee, N. Duffield, and R. Kompella. Two samples are enough: opportunistic flow-level latency estimation using netflow. In *Proc. of IEEE INFOCOM*, 2010.

[20] M. Lee, S. Goldberg, R. Kompella, and G. Varghese. Fine-grained latency and loss measurements in the presence of reordering. In *Proc. of ACM SIGMETRICS*, 2011.

[21] R. Martin. Wall street's quest to process data at the speed of light. `www.informationweek.com/news/infrastructure/showArticle.jhtml?articleID=199200297`.

[22] S. Moon, P. Skelly, and D. Towsley. Estimation and removal of clock skew from network delay measurements. In *Proc. of IEEE INFOCOM*, 1999.

[23] K. Park, G. Kim, and M. Crovella. On the relationship between file sizes, transport protocols, and self-similar network traffic. In *Proc. of International Conference on Network Protocols*, 2002.

[24] V. Paxson. Measurements and analysis of end-to-end Internet dynamics. Technical Report CSD-97-945, University of California at Berkeley, 1998.

[25] V. Paxson. On calibrating measurements of packet transit times. In *ACM SIGMETRICS Performance Evaluation Review*, volume 26, pages 11–21. ACM, 1998.

[26] C. Y. Robert and J. Segers. Tails of random sums of a heavy-tailed number of light-tailed terms. *Insurance: Mathematics and Economics*, 43(1):85 – 92, 2008.

[27] J. Sanjuas-Cuxart, P. Barlet-Ros, and J. Solé-Pareta. Validation and Improvement of the Lossy Difference Aggregator to Measure Packet Delays. *Traffic Monitoring and Analysis Workshop*, 2010.

[28] J. Sommers, P. Barford, N. Duffield, and A. Ron. Accurate and efficient SLA compliance monitoring. In *Proc. of ACM SIGCOMM*, 2007.

[29] L. Zhang, Z. Liu, and C. Honghui Xia. Clock synchronization algorithms for network measurements. In *Proc. of IEEE INFOCOM*, 2002.