Every Timestamp Counts: Accurate Tracking of Network Latencies using Reconcilable Difference Aggregator

Yongquan Fu, Pere Barlet-Ros and Dongsheng Li

User-facing services deployed in data centers must respond quickly to user actions. The measurement of network latencies is of paramount importance. Recently, a new family of compact data structures has been proposed to estimate one-way latencies. In order to achieve scalability, these new methods rely on timestamp aggregation. Unfortunately, this approach suffers from serious accuracy problems in the presence of packet loss and reordering, given that a single lost or out-of-order packet may invalidate a huge number of aggregated samples.

In this paper, we unify the problem to detect lost and reordered packets within the set reconciliation framework. Although the set reconciliation approach and the data structures for aggregating packet timestamps are previously known, the combination of these two principles is novel. We present a space-efficient synopsis called Reconcilable Difference Aggregator (RDA). RDA maximizes the percentage of useful packets for latency measurement by mapping packets to multiple banks and repairing aggregated samples that have been damaged by lost and reordered packets. RDA simultaneously obtains the average and the standard deviation of the latency. We provide a formal guarantee of the performance and derive optimized parameters. We further design and implement a user-space passive latency measurement system that addresses practical issues of integrating RDA into the network stack. Our extensive evaluation shows that compared to existing methods, our approach improves the relative error of the average latency estimation in 10-15 orders of magnitude, and the relative error of the standard deviation in 0.5-6 orders of magnitude.

***Index Terms*—Passive measurement, loss, reorder, reconciliation, latency.**

I. Introduction

Most user-facing services are deployed in data centers. These services must respond quickly to user actions in order to provide a fluid experience to end users. Even delays of a few milliseconds may have a severe impact on the quality of experience [24], [25], [13], [14], [12], [11]. Although significant progress has been made in the design of new network architectures and transport protocols for data centers [5], [6], [11], [29], meeting the tail of the latency distribution still remains a challenge, given the complexity of scale-out services and the varying queueing latencies typical from bursty data center workloads. Therefore, the measurement of network latencies is of paramount importance to assess the compliance of application deadlines and to diagnose problematic tail response times.

Unfortunately, measuring network latencies in data centers is extremely challenging. Traditional latency measurement methods based on active probing are not accurate enough. Active probing collects measurement samples of the injected probing packets, but not those from the packet streams between measurement points. Consequently, the sampled end to end latencies provide coarse-grained metrics, but have low fidelity unless the sampling rates are very high. For example, software bugs or faulty interfaces in switches may randomly produce failures on some packets according to the route in the network or packet header information [31]. Unfortunately, high sampling frequencies have high bandwidth cost, computing cost, and storage overhead. Due to the large number of servers and the high probing frequencies required, active measurements do not scale well with data center speeds and size, which can even interfere with regular data center traffic [15].

Passive methods are generally preferable for measuring latencies in data centers, assuming that measurement points can synchronize their clocks. In passive methods, one measurement point (called sender) records the timestamps of outgoing packets to the other measurement point (called receiver), while the receiver records the timestamps of packets coming from the sender, and vice versa[1] At the end of a measurement interval, the sender and the receiver exchange the timestamps and subtract them to obtain the one-way packet latency. Although this approach is very accurate, it does not scale well with increasing traffic volumes.

To address this scalability problem, some studies have recently proposed a new set of efficient data structures, such as LDA (Lossy Difference Aggregator) [19], FineComb [22], LDS (Lossy Difference Sketch) [27] and COLATE [28]. The main intuition behind these proposals is that to measure packet latencies it is not necessary to exchange individual timestamps, but instead they can be aggregated to an array of buckets, where each packet is mapped to a random bucket and each bucket accumulates the timestamps and the total number of packets. Nevertheless, the accuracy of these methods still degrades drastically in the presence of packet loss and reordering. As modern data centers may use multiple alternative paths to increase the aggregate bandwidth or to provide fault tolerance [5], while multi-path routing protocols (e.g., equal cost multipath) may balance the load among different paths. Data centers may also drop packets because of congestion

Yongquan Fu and Dongsheng Li are with the Science and Technology Laboratory of Parallel and Distributed Processing; College of Computer Science, National University of Defense Technology.

Pere Barlet-Ros is with UPC BarcelonaTech.

(*corresponding author: Dongsheng Li*)

E-mail: yongquanf@nudt.edu.cn, dsli@nudt.edu.cn, pbarlet@ac.upc.edu.

[1] The term “timestamp” refers to the time a packet was sent or the time a packet was received. The sender and the receiver do not need to be the origin and destination of the traffic, but the two network points from where we want to estimate the latency, e.g., two switches in a data center.

DOI: 10.1109/TNET.2017.2762328

resulting from bursty traffic [11] or even due to packet black holes [15].

In order to achieve robustness against lost packets and reordered packets, these problematic packets should be detected and removed from the buckets. Unfortunately, detecting lost or reordered packets in a space-efficient way is challenging. FineComb [22] and LDS [27] detect and discard a pair of buckets that have some different packets. However, discarded buckets may contain many useful packets for computing the latency. Further, FineComb repairs buckets that are only damaged by reordered packets, however, FineComb cannot identify lost packets at the sender. Moreover, even with a small number of lost packets, the number of useful buckets decreases fast, as shown in Subsection III-C, which may result in too many samples discarded. Finally, FineComb adopts a brute-force approach to find buckets that contain reordered packets, which incurs a high computational cost.

In this paper, we unify the problem to detect lost packets and reordered packets within the set reconciliation framework. Although the set reconciliation approach [10] and the data structures for aggregating packet timestamps are previously known, the combination of these two principles is novel. Based on the observation, we proposed a naive synopsis that simultaneously aggregates the timestamps and detects lost packets and reordered packets using a unifying hashing based data structure. Unfortunately, the naive synopsis is space-redundant and fails to compute the standard deviation of the latency.

Next, we presented a space-efficient data structure (RDA) that obtains the average and the standard deviation of the latency. RDA maximizes the percentage of useful packet samples for latency measurements by mapping packets to multiple banks and repairing aggregated samples that have been damaged by lost and reordered packets. We provide a formal guarantee of the performance and derive optimized parameters.

Further, we designed and implemented a user-space end-to-end passive latency measurement system. Different from existing studies, we are able to measure the packet stream in a pipelined approach by delimiting the measurement interval with the already synchronized clock between measurement points instead of controlling packets.

Finally, our experimental results show that compared to existing methods, our proposal improves the relative error of estimating the average latency in 10-15 orders of magnitude, and the relative error of estimating the standard deviation in 0.5-6 orders of magnitude.

Going forward, Section II introduces background of passive synopsis based latency aggregation. Section III states the problem of problematic packets. Next, Section IV proposes a unified framework to detect these packets and presents a naive synopsis that reconciles lost and reordered packets and estimates the average latency. Then, Section V introduces RDA that is space-efficient and accurately computes the average and the standard deviation. Section VI presents the performance bounds for RDA. Section VII presents extensive evaluation results. Section VIII presents the implementation of RDA based passive latency measurement in user space. Section IX reports a prototype deployment on a small data center. Finally, we conclude in Section X. We summarize related work in the Appendix, which is available in the online supplemental material.

II. Background

We introduce the background of synopsis based passive latency measurement in this section.

A. Requirements

Many applications deployed in data center have tight latency requirements. For example, high frequency algorithmic trading applications have very short holding period, even delays greater than 100 μs can cause financial losses [19], storage-area networks (SAN) use Fiber Channel over Ethernet to deliver similar latencies as the traditional IO bus between CPUs and remote disks [20], Spark provides millisecond large-scale data processing [26]. Unfortunately, the latency distributions of these requests are usually long-tail [11], where the average and the tail latency may differ by several orders of magnitude, which significantly increases the completion time of the service, since the number of users is usually on the orders of millions to billions. Understanding and troubleshooting fine-grained latency issues needs packet-level information.

In this paper, we measure fine-grained, packet-level latency without sampling. The measurement is divided into *intervals*. A measurement interval seeks to capture the latency of a maximum number n of packets, where n is a constant. Our passive latency measurement is based on the coordinated measurement scheme proposed by Kompella et al. [19]: (i) **Average**, captures the central tendency of latency, which characterizes the long-term latency trend [11]; (ii) **Variance**, measures how far the latencies are spread out, which correlates with the latency tail: the higher the variance, the worse the long-tail problem [17], [30]. Estimating other metrics like the order statistics such as the maximum delay or the quantiles requires knowledge of the latency value of each packet, unfortunately, the coordinated measurement scheme does not fulfill this requirement as it mixes the latency values of different packets.

B. Assumptions

We follow the same assumptions for passive latency measurement problem [19]:

(i) Timestamps are not embedded in packet headers, as in LDA, FineComb and LDS. UDP packets do not carry the time stamps. TCP protocol contains a timestamp field [8], which unfortunately incurs an additional overhead for every TCP packet. For a 32-bit timestamp, the bandwidth consumption could increase by 10% [19].

(ii) Two measurement points should have synchronized their clocks to the microsecond-level precision before the measurement starts [19], [22], [27]. End points are synchronized using the Network Time Protocol (NTP) [1] or the IEEE 1588 Precise Time Protocol (PTP) [2]. Both NTP and

PTP provide microsecond-level (10^{-6} second) precision in a local area network (LAN). Moreover, the Global Positioning System (GPS) provides up to nanosecond-level (10^{-9} second) precision [23] at the expense of the dedicated GPS hardware. Recently, the Datacenter Time Protocol (DTP) [21] provides nanosecond-level precision time synchronization across the whole data center via the physical layer (PHY) protocols.

C. *Synopsis based Latency Aggregation*

We next briefly introduce the coordinated measurement scheme [19]. A bucket consists of a timestamp accumulator and a counter. For two LDAs with the same number of buckets and the same hash function, a packet is always mapped to the same bucket. The accumulator accumulates the timestamps of all packets inserted to the bucket, while the counter maintains the number of packets inserted to that bucket. The sender and the receiver uses the same hash function to ensure that a packet is always mapped to the same position in the array.

Assuming that two measurement points record the same set of packets, for each pair of buckets in the same location at the sender and receiver, the difference between the sum of accumulated timestamps, divided by the sum of accumulated numbers of packets, amounts to the average latency of packets that are inserted to this bucket. Using an array of buckets ensures that a single lost or reordered packet may not invalidate all aggregated timestamps, since if a lost or reordered packet exists in a bucket, then the set of packets aggregated at this bucket will be different from those at the other measurement point, and the subtraction of the aggregated timestamps of two buckets will no longer amount to the sum of latencies.

In order to estimate the average latency, the sender sends its LDA to the receiver, and then the receiver computes the difference between both LDAs for each pair of useful buckets. A pair of buckets is useful if the value of their counters is the same in both LDAs. The remaining buckets are discarded as not useful. Then, the average latency is computed as the sum of the differences in the timestamp aggregators divided by the sum of the counters.

For brevity, given two LDAs D_A and D_B with $m \times 1$ timestamp accumulator arrays maintained at the sender A and the receiver B, respectively. Let $D\left[i\right].T$ denote the timestamp accumulator and $D\left[i\right].C$ the packet counter for the i-th bucket, where $i \leq m$. In LDA, a pair of buckets are called **useful** if their packet counters match with each other.

The average latency is computed using all pairs of useful buckets, while the standard deviation is calculated efficiently without additional storage overhead. Let

$$\begin{array}{c}\widetilde{D}_A\left[j\right].T = D_A\left[2j\right].T - D_A\left[2j-1\right].T\\ \widetilde{D}_B\left[j\right].T = D_B\left[2j\right].T - D_B\left[2j-1\right].T\\ \widetilde{D}_A\left[j\right].C = D_A\left[2j\right].C + D_A\left[2j-1\right].C\\ \widetilde{D}_B\left[j\right].C = D_B\left[2j\right].C + D_B\left[2j-1\right].C\end{array} \tag{1}$$

be the collapsed timestamp accumulator and packet counters, where $j \in [1, \lfloor m/2 \rfloor]$. Let

$$F = \frac{\sum\limits_{j\in\left\{i\middle|\widetilde{D}_A[i].C=\widetilde{D}_B[i].C\right\}}\left(\widetilde{D}_B\left[j\right].T - \widetilde{D}_A\left[j\right].T\right)^2}{\sum_{\widetilde{D}_A[i].C=\widetilde{D}_B[i].C}\widetilde{D}_A\left[i\right].C} \tag{2}$$

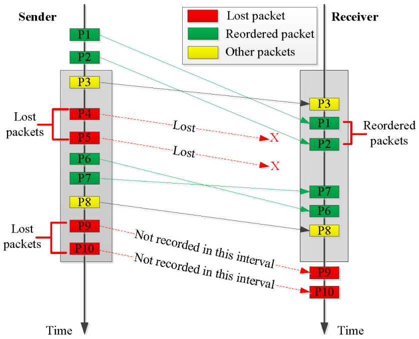


Fig. 1. The timeline of a measurement interval.

Then the standard deviation is approximated as:

$$\sigma^2 = F^2 - \mu^2 \tag{3}$$

III. Problem Statement

Having presented the background, we next present a new measurement interval that continuously monitors the end to end latency. Next, we discuss challenges to obtain accurate latency aggregation due to problematic packets.

A. *Continuous Monitoring via Pipelined Measurement Interval*

Existing approaches use **delimiting packets** to define a measurement interval. To start a measurement interval, the sender sends an interval-start message to the receiver. When the receiver records up to n packets into its synopsis, the receiver sends an interval-end message to the sender to terminate the measurement interval. Unfortunately, due to the delay of processing the interval-start and interval-end messages, we are unable to continuously monitor the end-to-end latency between a pair of measurement points.

Different from existing studies, we propose to measure the packet stream in a pipelined approach by delimiting the measurement interval with the already synchronized clock between measurement points instead of delimiting packets. Each measurement interval i is defined by a beginning timestamp t_i and an interval-length parameter δ_i (say one second). A pair of measurement points capture packets during the interval delimited by two timestamps (t_i, $t_i + \delta_i$). As a result, the problem of continuously aggregating the latency can be simply solved: we only need to set the beginning timestamp t_{i+1} of the successor measurement interval $i+1$ to the beginning timestamp of the last measurement interval plus the interval-length parameter, i.e., $t_{i+1} = t_i + \delta_i$.

During each measurement interval, each measurement point maintains a separate synopsis that aggregates the timestamps of packets within this measurement interval and calculates the average and the standard deviation of the latency of this measurement interval.

B. *Problematic Packets*

Figure 1 illustrates a measurement interval. We can see that each measurement point records some problematic packets in Figure 1:

- The receiver records packets P1 and P2, while the sender misses these two packets.
- The sender records packets P4 and P5 that are lost during the measurement interval, consequently, the receiver misses these two packets.
- The sender records P9 and P10, but the receiver misses these two packets, as packets P9 and P10 are sent before the sender terminates the measurement interval.
- Further, some packets may arrive in an out-of-order sequence, e.g., packets P6 arrives at the receiver later than P7. However, as long as both packets are stored at the sender and the receiver, we do not care about such packets during the latency aggregation process.

In order to obtain truthful latency statistics, we need to discard the lost packets and the reordered packets from the latency aggregation. If two measurement points are within the same router or directly connected, there may be few lost or reordered packets between them. While if these devices are several hops away, significant loss or reordering may arise with increasing traffic volumes, since a packet that passes one device may not traverse the other one due to multipath routing or load balancing.

Lost and reordered packets interfere with latency measurements, since only one measurement point obtains this packet, while the other measurement point is agnostic to this packet. Consequently, these packets should be eliminated from the latency estimation.

In an extreme case, a packet that is sent in the previous measurement interval from the sender may arrive at the receiver at the next measurement interval. At the previous measurement interval, this packet is only stored at the sender, i.e., a lost packet, while for the next measurement interval, this packet is only stored at the receiver, i.e., a reordered packet.

Besides the lost and reordered packets, duplicate packets may arise, e.g., timeout packets, or duplicated acknowledgement packets to trigger the fast retransmission. The duplication issue has not been discussed in the literature to the best of our knowledge. Unfortunately, it is difficult to determine the duplicated packets without storing all packets. A simple approach is to record the packets with unique identifiers into a cache and to discard all the other duplicated packets. An interesting open question is how to preserve as many duplicates as possible to maximize the number of useful packets for latency computation.

C. Challenges for Existing Aggregation Approaches

Having stated the problematic packets that may occur during the measurement process, we next analyze the useless buckets caused by problematic packets for LDA and FineComb. In the next section, we propose a unifying framework to detect the lost and reordered packets and a naive approach based on the set reconciliation and discuss its limitations.

1) LDA

LDA [19] selectively samples packets from the packet stream and maps them to a number of buckets. Unfortunately, even if the counters of two buckets match with each other, these two buckets may contain different packets due to different combinations of lost and reordered packets, as shown in Figure 2.

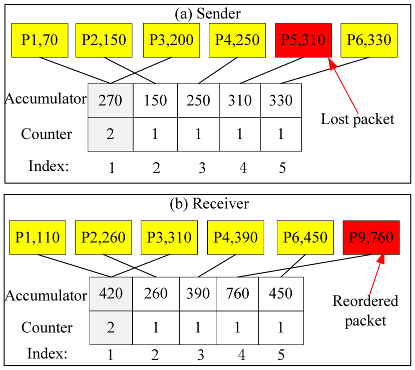


Fig. 2. An example of LDA that are damaged by problematic packets. The sender sends packets P1 to P9 to the receiver. A measurement interval begins before packet P1 and ends after packet P6. However, P5 is lost and thus not recorded at the receiver. Further, due to packet reordering, P9 arrives at the receiver earlier than the ending message, which is thus recorded at the receiver. In (a), the sender inserts the timestamps (shown in the upper-level box) of packets P1, P2, P3, P4, P5, P6 into its LDA. In (b), the receiver inserts the timestamps of packets P1, P2, P3, P4, P6 and P9 into its LDA.

In Figure 2, all pairs of buckets are useful in LDA. The average latency using all useful buckets is $((420-270)+(260-150)+(390-250)+(760-310)+(450-330))/(2+1+1+1+1) = 161$. However, the actual latency is 104 using the successfully delivered packets P1, P2, P3, P4 and P6. This is because the fourth pair of buckets have two different packets P9 and P5.

2) FineComb

FineComb detects whether each bucket is damaged by problematic packets, by appending a parity-string field to each bucket. The parity string is computed as the XOR value of entire packets that are mapped to this bucket. If two buckets have some different packets, we can see that their parity strings will differ from each other, and they will be discarded as useless.

Further, FineComb tries to remove reordered packets from buckets, by maintaining a stash of packets that are likely to be reordered at the receiver. After the measurement interval ends, the receiver obtains the sender's FineComb and subtracts each pair of buckets at the same location in two FineCombs. The result is stored in a new bucket. Next, for each of these new buckets, the receiver compares its parity string with the XOR result of each possible combination of packets in the stash: If these two XOR values match with each other, then FineComb assumes that this combination of packets are reordering packets in its bucket.

However, *if a bucket contains some lost packets, FineComb will be unable to repair this bucket, and all useful samples in this bucket become useless*. As a lost packet may be spread to any bucket, we next compute the expected number of buckets that contain at least one lost packet in Theorem 1:

Theorem 1. *Suppose that a number n_l of lost packets are inserted into a FineComb with m buckets using a perfectly random hash function. The expected number m_l of buckets* *that contain at least one lost packet is* $m \cdot \left(1 - e^{-n_l/m}\right)$.

The proof is presented in the Appendix, which is available in the online supplemental material. Theorem 1 gives the expected number of buckets that contain lost packets. For example, when m = 100, n_l = 50, the expected number m_l of buckets having lost packets is approximately 40, which means that only with 50 lost packets, 40% of the buckets used by FineComb would be useless.

IV. Naive Approach

Having stated the challenges, we propose a unifying framework to detect the problematic packets from the synopsis. Next, based on the connection with the set reconciliation problem [10], we propose a naive synopsis sRDA (simple Reconcilable Difference Aggregator). sRDA simultaneously solves two problems with a unifying hashing based data structure: (i) detecting and removing lost packets and reordered packets between a pair of measurement points, and (ii) computing aggregated latency using the useful packets that are recorded at both sides. Finally, we discuss the limitations of sRDA. In the next section, we present a novel synopsis that addresses these limitations.

A. Detecting Problematic Packets and the Set Reconciliation

Recall that each packet's **identifier** is invariant at both the sender and the receiver, while the timestamp of each packet varies between the sender and the receiver. Let S_S be the set of identifiers of packets that are intercepted by the sender. Let S_R be the set of identifiers of packets that are intercepted by the receiver. The **set intersection** $S_S \cap S_R$ of the two sets correspond to the set of identifiers of packets stored at both measurement points. The union of the identifiers of the lost and reordered packets refers to the **set difference** for S_S and S_R. Let $\overline{S_R} = \{x \,|x \notin S_R\}$ and $\overline{S_S} = \{x \,|x \notin S_S\}$ refer to the complement sets of S_S and S_R, respectively. The **set difference** $S_S \oplus S_R$ can be represented as

$$S_S \oplus S_R = \left\{ p \,\middle|\, p \in \underbrace{(S_S \cap \overline{S_R})}_{\text{Lost}} \cup \underbrace{(S_R \cap \overline{S_S})}_{\text{Reordered}} \right\}$$

The problem of finding the identifiers of the lost and the reordered packets is transformed to the problem of detecting the set difference $S_S \oplus S_R$ for the packets stored at the sender and those at the receiver. We can see that the packets in the set intersection $S_S \cap S_R$ are useful for computing the latency metric, while the packets in the set difference should be eliminated from the latency calculation.

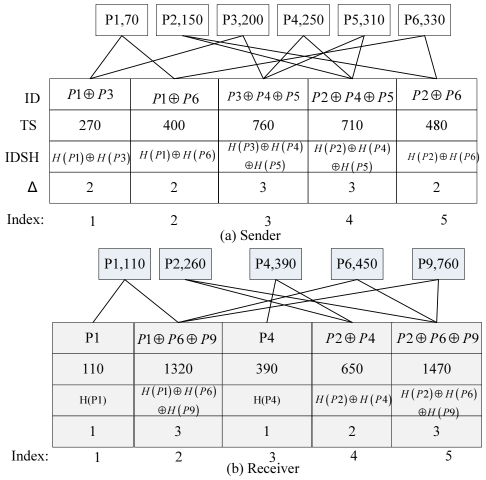
As the synopsis mixes the timestamps of individual packets, each measurement point is agnostic of the packets that will be lost or reordered *a priori*, consequently, each measurement point has to keep a local cache of packets, in order to filter out the timestamps of lost and reordered packets from the synopsis.

B. Synopsis Structure

A sRDA keeps a flat array of buckets that store the aggregated timestamps and the necessary information to find the lost and reordered packets:

- To aggregate the latency, each bucket records the sum of the timestamps of the packets (**TS** field) and counts the number of packets inserted to that bucket (Δ field).
- To detect the set reconciliation, each bucket accumulates the XOR value of the identifiers of the inserted packets (**ID** field), and accumulates the XOR value of the **hashing number** of the packet identifiers using an independent hash function $H()$ (**IDSH** field).
  Both ID and IDSH fields are used to reconcile the set difference between the packets stored at the sender and those at the receiver. First, for a pair of buckets that have some common packets, we can see that XORing the ID fields of these two buckets cancels the identifiers of common packets, but preserves the set difference in two buckets. Second, IDSH enables us to determine whether a bucket contains a unique packet.

Figure 3 shows an example of two sRDAs at the sender and the receiver. We can see that each pair of buckets in two sRDAs have some lost packets, or reordered packets or even both. As a result, all buckets are useless for latency estimation. We need to detect the lost and reordered packets and remove them from the sRDAs.



ID	$P1 \oplus P3$	$P1 \oplus P6$	$P3 \oplus P4 \oplus P5$	$P2 \oplus P4 \oplus P5$	$P2 \oplus P6$
TS	270	400	760	710	480
IDSH	$H(P1) \oplus H(P3)$	$H(P1) \oplus H(P6)$	$H(P3) \oplus H(P4)$ $\oplus H(P5)$	$H(P2) \oplus H(P4)$ $\oplus H(P5)$	$H(P2) \oplus H(P6)$
Δ	2	2	3	3	2
Index:	1	2	3	4	5

(a) Sender

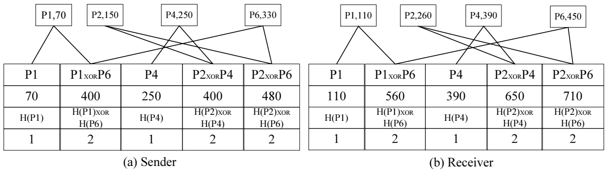
P1	$P1 \oplus P6 \oplus P9$	P4	$P2 \oplus P4$	$P2 \oplus P6 \oplus P9$
110	1320	390	650	1470
H(P1)	$H(P1) \oplus H(P6)$ $\oplus H(P9)$	H(P4)	$H(P2) \oplus H(P4)$	$H(P2) \oplus H(P6)$ $\oplus H(P9)$
1	3	1	2	3

Index: 1 2 3 4 5

(b) Receiver

Fig. 3. The sRDAs of the sender and the receiver. There are five buckets in each sRDA, and each packet is hashed to two random buckets. The timestamp of each packet is appended to the packet identifier. Solid lines are used to represent the mapping relationship between the packets and the buckets. Packets P3 and P5 are lost at the receiver, while the packet P9 is reordered and not stored at the sender.

A packet is hashed to k (2 by default) buckets using k independent hash functions. When a new packet arrives at the sender or the receiver, we hash the identifier of this packet and obtain at most k different buckets. Then, for each of these unique buckets, we update the bucket as follows: (a) ID = ID $\oplus$ packet's identifier; (b) TS = TS + packet's timestamp; (c) IDSH = IDSH $\oplus$ H(packet's identifier); (d) Δ = Δ + 1.



P1	P1XORP6	P4	P2XORP4	P2XORP6
70	400	250	400	480
H(P1)	H(P1)XOR H(P6)	H(P4)	H(P2)XOR H(P4)	H(P2)XOR H(P6)
1	2	1	2	2

(a) Sender

P1	P1XORP6	P4	P2XORP4	P2XORP6
110	560	390	650	710
H(P1)	H(P1)XOR H(P6)	H(P4)	H(P2)XOR H(P4)	H(P2)XOR H(P6)
1	2	1	2	2

(b) Receiver

Fig. 4. Repaired sRDAs in Figure 3.

C. Detecting Lost Packets and Reordered Packets

First, we construct a **subtraction sRDA** that preserves the set difference, but cancels out the identifiers of packets in the set intersection. To that end, we exploit the XOR operation of two identical identifiers cancelling out this identifier. For each pair of buckets (I_s, I_r) at the same location, we construct a new **subtraction bucket** whose: (a) ID field amounts to the XOR value of two IDs in the two buckets: I_r.ID $\oplus I_s$.ID; (b) IDSH field amounts to the XOR value of two IDSHes in the two buckets: I_r.IDSH $\oplus I_s$.IDSH; (c) Δ field amounts to the subtraction of two Δ values: $I_r.\Delta$ - $I_s.\Delta$. We do not modify the TS field, since our goal is to list the identifiers of packets in the set difference.

Second, let a bucket be **pure** if this bucket contains only one packet, we next iteratively list all pure buckets and delete the corresponding packet from the subtraction sRDA until no pure buckets exist. As the subtraction sRDA only contains the problematic packets, we decode all packets inserted to this subtraction sRDA based on the set reconciliation [10].

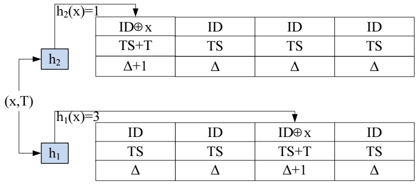
The set reconciliation is based on two key ideas: (i) **Pure condition**: Intuitively, for a non-pure bucket i, hashing its ID fields using the hash function $H(\cdot)$ will differ from its IDSH field; moreover, if $I(i).\Delta = \pm 1$, the numbers of packets stored at two original buckets differ by one. Therefore, if $H(I(i).\text{ID}) = I(i).\text{IDSH}$ and $I(i).\Delta = \pm 1$ both hold, this bucket i is claimed to be pure. (ii) **Separation**: Further, we determine whether the packet inserted into a pure bucket is a lost packet or a reordered packet:

- Δ = 1: the receiver's bucket must contain one more packet from the sender's bucket, consequently, this packet is only stored at the receiver, i.e., this packet is reordered.
- Δ = -1: the sender's bucket must contain one more packet than the receiver's bucket, therefore, this packet is stored only at the sender, i.e., this packet is lost.

The detailed decoding procedure of sRDA is presented in the Appendix, which is available in the online supplemental material.

D. Calculate Latency Aggregation

We can estimate the average latency after removing the lost packets and the reordered packets. For example, Figure 4 shows the repaired sRDAs in Figure 3. The accumulated subtraction of the timestamps in all buckets is computed as: $(110-70)$ + $(560-400)$ + $(390-250)$ + $(650-400)$ + $(710-480)$ = 820. The accumulated numbers of packets in all buckets is 1+2+1+2+2=8. Therefore, the average latency is $\frac{820}{8}$ = 102.5, which matches the ground-truth average value.



ID⊕x	ID	ID	ID
TS+T	TS	TS	TS
Δ+1	Δ	Δ	Δ

ID	ID	ID⊕x	ID
TS	TS	TS+T	TS
Δ	Δ	Δ+1	Δ

Fig. 5. RDA has multiple banks of buckets. A packet is inserted into a randomly selected bucket in each bank.

Further, we hope to compute the standard deviation. According to Eq (3), we can collapse adjacent buckets and derive the standard-deviation metric. Nevertheless, sRDA has a serious flaw. For example, in Figure 4, we collapse four adjacent buckets and obtain two collapsed buckets as follows: (i) Sender: the collapsed timestamps are 400-70=330, 400-250=150, respectively, and the collapsed counters are 3, 3, respectively; (ii) Receiver: the collapsed timestamps are 560-110=450, 650-390=260, respectively, and the collapsed counters are 3, 3, respectively. We next use collapsed buckets to compute the standard deviation according to Eq (3) as: $\frac{(450-330)^2+(260-150)^2}{3+3}$ - 102.5^2 = -6089.6, however, the ground-truth standard deviation is 43.49! This is because collapsing the first and the second bucket cancels out the timestamp of P1, while collapsing the third and the fourth bucket cancels the timestamp of P4, consequently, the first term $\frac{(450-330)^2+(260-150)^2}{3+3}$ = 4416.7 is much smaller than the squared average latency. Unfortunately, sRDA always has a probability to map a packet to adjacent buckets, and this probability increases as more packets are inserted to the sRDA.

Space-redundancy: Besides the above flaw, the IDSH field contains redundant information with respect to the ID field. Removing this redundant field saves 25% space, which is necessary to scale to large numbers of buckets.

V. RDA

Having presented the limitations of the naive approach, we next introduce a novel synopsis data structure RDA that is space-efficient and accurately calculates the average and the standard deviation.

A. Organization and Bucket Structure

In RDA, each bucket consists of the ID, TS and the Δ fields that are defined in the sRDA structure. We organize buckets into a multi-bank structure that consists of k banks of buckets, where each bank contains m buckets. We set the number of hash functions to k, so that we insert each incoming packet into a random bucket in each bank. As a result, no bank has two identical packets and RDA avoids each packet to be inserted to adjacent buckets. Figure 5 shows an RDA with two banks of four buckets.

In order to ensure constant time to access any bucket, we use one contiguous array to store the RDA in the main memory. Let a bank consist of m buckets. To logically split the array to k banks, the first bank contains the buckets from 1 to m, while the i-th ($i \geq 1$) bank consists of the buckets of $[(i-1) \cdot m + 1, i \cdot m]$.

To insert a packet into the RDA, we hash the packet's identifier and timestamp into each bank. Specifically, we select a bucket uniformly at random from each bank, by hashing the identifier of the packet with an independent hash function. For each of these buckets, we update the packet as follows: (a) ID = ID $\oplus$ packet's identifier; (b) TS = TS + packet's timestamp for insertion, and TS = TS - packet's timestamp for deletion; (c) $\Delta = \Delta + 1$ for insertion, and $\Delta = \Delta - 1$ for deletion.

B. Detect Lost Packets and Reordered Packets

For a pair of RDAs, we next present a lightweight approach to detect the lost and reordered packets using a new decoding process that does not need the IDSH field of the sRDA.

(i) **Cancel common items**: We perform a **subtraction** operation on a pair of RDAs to cancel out items that are inserted into both RDAs. The subtraction cancels the packet identifiers that are inserted to both RDAs, but preserves the identifiers of the lost packets and reordered packets.

We subtract the receiver's RDA using the sender's RDA, which yields a **subtraction RDA**. For each pair of buckets at the same location in two RDAs, we put a new bucket into the subtraction RDA that is constructed: (a) ID = ID $\oplus$ I.ID; (b) Δ = Δ - 1.

(ii) **Decode**: We next define a new decoding process on the subtraction RDA. As we have removed the IDSH field in the sRDA, we need to define a new pure condition in order to find buckets that contain only one identifier. Our key insight is that, *the ID field of a bucket amounts to the XOR of the packets that are mapped to this bucket, consequently, if an RDA bucket contains only one packet, then hashing the ID field of this bucket via this bank's hash function will obtain the index of this bucket in the bank; while if a bucket contains multiple packets, hashing the XOR of these packets will obtain a different index in this bank*. Therefore, *a bucket is claimed to be pure if hashing the ID field of this bucket via this bank's hash function amounts to the index of this bucket in this bucket, and its Δ field simultaneously amounts to 1 or -1.*

Algorithm 1 summarizes the process to list problematic packets in a subtraction RDA. Lines 3 to 6 record pure buckets into a set Υ by traversing each bucket in each bank. Next, lines 7 to 23 iteratively decode lost packets and reordered packets. First, line 8 removes a bucket record from the set of recorded pure buckets. This bucket may become empty due to the update of the last iteration. If the bucket becomes empty, no problematic packets exist, so we move to the next iteration. Otherwise, we continue this iteration. Lines 12 to 16 extract the packet identifier using the ID field of this bucket and classify this packet to lost or reordered based on the Δ field of this bucket. Next, lines 17 to 21 delete this packet from each bank of the subtraction RDA; meanwhile, if an updated bucket becomes pure, we save this bucket index into the set of pure buckets. Then we turn to the next iteration until no pure buckets exist.

The time to scan all buckets takes $O(k \cdot m)$, while the time to delete all pure buckets amounts to $O(k \cdot d)$, where d denotes the total number of lost and reordered packets. Thus, we need an overall $O(k \cdot (m+d))$ time. We can see that RDA's decoding complexity is independent of the size of the cache. In contrast, FineComb finds the reordered packets in each bucket using a brute-force approach whose time complexity depends on the size of the stored packets. Let m be the number of buckets, and $\left|S_R^{Stash}\right|$ the size of the receiver's cache, then FineComb's expected complexity amounts to $O(m2^{\left|S_R^{Stash}\right|})$.

Algorithm 1: Decode problematic packets for a subtraction RDA.

```
1  Decode($I_{AB}$)
   input : $I_{AB}$: subtraction RDA: $I_B - I_A$.
   output: $ID_{A\overline{B}}$: lost packets. $ID_{\overline{A}B}$: reordered packets.
2  $\Upsilon$ = {}, $ID_{A\overline{B}}$={}, $ID_{\overline{A}B}$={};
3  for each bank $l \in [1, k]$ do
4      for each $i \in [1, m]$ do
5          if $h_l(I_{AB}(i,l).ID) == i \wedge |I_{AB}(i,l).\Delta| == 1$ then
6              $\Upsilon$= $\Upsilon \cup \{(i,l)\}$ ;
7  while $\Upsilon \neq \emptyset$ do
8      Remove a record $(i,l)$ from $\Upsilon$, where $i$ denotes the bucket
       index, and $l$ denotes the index of the bank;
9      if bucket $i$ becomes empty, i.e., $I_{AB}(i,l).\Delta$ = 0,
       $I_{AB}(i,l).ID$ = 0 then
10         continue;
11     else
12         id = $I_{AB}(i,l).ID$;
13         if $I_{AB}(i,l).\Delta$ ==1 then
14             $ID_{A\overline{B}}$ = $ID_{A\overline{B}} \cup$ {id} ;
15         else
16             $ID_{\overline{A}B}$ = $ID_{\overline{A}B} \cup$ {id};
17         delta = $I_{AB}(i,l).\Delta$;
18         for each bank $l \in [1, k]$ do
19             q = $h_l$(id);
20             $I_{AB}(q,l).ID = I_{AB}(q,l).ID \bigoplus$ id;
21             $I_{AB}(q,l).\Delta = I_{AB}(q,l).\Delta -$ delta;
22             if $|I_{AB}(q,l).\Delta| == 1 \wedge h_l(I_{AB}(q,l).ID) == q$
               then
23                 $\Upsilon$ = $\Upsilon \cup \{(q,l)\}$ ;
24 return $ID_{A\overline{B}}, ID_{\overline{A}B}$;
```

C. Latency Aggregation

Having presented the process to detect the lost and reordered packets, we next compute the average latency and the standard deviation. We propose a new algorithm to compute the standard deviation.

Average Latency: We compute the average latency using all buckets that do not have lost and reordered packets. As each packet is mapped to each bank, we can see that the timestamp of a packet may be aggregated multiple times in different buckets, since each packet is mapped to multiple banks. The redundancy usually increases the number of useful latency samples when some buckets are useless due to the lost or reordered packets.

Standard deviation: To compute the standard deviation, we need to collapse adjacent buckets in each bank. In LDA, every two adjacent buckets are directly collapsed. Unfortunately, some buckets may be empty; meanwhile, when the decoding

Algorithm 2: Collapse two RDAs.

```
1  Collapse($I_1$, $I_2$)
   input : $I_1$, $I_2$: a pair of RDAs.
   output: $I_1'$, $I_2'$: a pair of collapsed RDAs.
2  $I_1'$= $\emptyset$, $I_2'$= $\emptyset$ ;
3  for each $i$ in $k$ do
4      $\Psi$ = $\emptyset$ ;
5      for $j \in [(i-1) \cdot m + 1, i \cdot m]$ do
6          if $I_1[j]$.ID == $I_2[j]$.ID AND $I_1[j].\Delta > 0$ then
7              $\Psi = \Psi \cup \{j\}$;
8      if $|\Psi| \geq 2$ then
9          for $c \in [2, |\Psi|]$ do
10             p = $\Psi$ $[2c-1]$, q = $\Psi$ $[2c]$;
11             B1.ID = $I_1[p]$.ID $\oplus$ $I_1[q]$.ID;
12             B1.$\Delta$ = $I_1[p].\Delta$ + $I_1[q].\Delta$;
13             B1.TS = $I_1[p]$.TS - $I_1[q]$.TS;
14             $I_1'$.add(B1);
15             B2.ID = $I_2[p]$.ID $\oplus$ $I_2[q]$.ID;
16             B2.$\Delta$ = $I_2[p].\Delta$ + $I_2[q].\Delta$;
17             B2.TS = $I_2[p]$.TS - $I_2[q]$.TS;
18             $I_2'$.add(B2);
19 return $I_1'$, $I_2'$ ;
```

Algorithm 3: Estimate the standard deviation.

```
STD($I_1'$, $I_2'$, $\hat{\mu}$)
input : $I_1'$, $I_2'$: collapsed RDAs, $\hat{\mu}$: estimated average latency.
SqSum=0, Count = 0;;
for each $i$ in $|I_1'|$ do
    Count=Count+$I_1'$ $[i].\Delta$;
    SqSum = SqSum+ ($I_1'$ $[i]$.TS - $I_2'$ $[i]$.TS)$^2$;
return $\frac{SqSum}{Count}$ - $\hat{\mu}^2$ ;
```

process does not completely succeed, some buckets may still contain lost packets or reordered packets.

As a result, collapsing physically adjacent buckets leads to two drawbacks: First, if one of the collapsed buckets contains some lost packets or reordered packets, then the collapsed bucket will be useless, since it still contains these problematic packets. Second, if one of the collapsed buckets contains no packets, then this collapsing is ineffective to derive accurate standard deviation, as packets in the collapsed bucket should be assigned randomized signs.

In order to address the above limitations, we propose a new Algorithm 2 to selectively collapse buckets for a pair of RDAs. Lines 5 to 7 select nonempty buckets that do not contain problematic packets. Lines 8 to 18 collapse selected buckets. If the number of available buckets is smaller than two, no collapsing is feasible, so we move to the next bank. Otherwise, we collapse every two buckets from lines 10 to 18, and store the collapsed bucket to a vector of buckets. We can see that two buckets being collapsed may not be adjacent with each other in the original RDA. Since we need to iterate over each bucket, Algorithm 2 requires $O(km)$ time.

After collapsing all banks, we next compute the standard deviation based on Eq (3). Algorithm 3 shows the computation using a pair of vectors of collapsed buckets.

VI. RDA Theoretical Guarantees

We state performance guarantees for RDA in this section. Detailed derivations can be found in the Appendix, which is available in the online supplemental material.

Theorem 2. *Let S_S and S_R be the set of packets recorded at the sender and the receiver, respectively. Let $d = |S_S \oplus S_R|$ be the cardinality of the set difference. Let k be the number of hash functions. Let the number m of buckets per bank be $2d$. The failure probability to reconcile all lost and reordered packets $S_S \oplus S_R$ is at most $O(d^{-k})$.*

We next analyze the number of useless packets for latency measurement due to the decoding failure.

Lemma 1. *For a RDA with k banks of buckets, where each bank is of size m. Let n be the total number of packets that are recorded into this RDA. Let $\{L_i\}$ for $i \in [1, k]$, $L_i \in [0, m]$ denote the numbers of buckets that cannot be repaired in each bank. The expected number of useless packets for the latency measurement amounts to $n \cdot \frac{\prod_{i=1}^{k} L_i}{m^k}$.*

RDA preserves most packets using multiple banks. For a RDA with two hash functions, i.e., two banks of buckets, let the percentage of buckets that cannot be repaired in two banks be 0.1 and 0.1, respectively, then the expected percentage of useless packets amounts to $\frac{L_1}{m} \cdot \frac{L_2}{m}$ = 0.1·0.1 = 0.01. Therefore, most packets are useful for the latency measurement.

We next analyze the effect of the skew of the time synchronization on the aggregation accuracy.

Lemma 2. *Assume that a pair of clocks between two measurement points are shifted by a constant δ. Let n be the total number of packets. The estimated average latency will be shifted by δ from the one with the perfect time synchronization, while the expected standard deviation are shifted by $2\delta \cdot \mu\,(n-1)$.*

Having bounded the effect of the time drift, we next ask how many samples are enough to bound the accuracy to estimate the latency metric. Intuitively, if the latency does not change, one sample is enough to compute an accurate average metric and the standard deviation is zero. While if the latency constantly varies, we need more samples to approximate the expected latency metric.

Lemma 3. *Let μ and σ be the actual average and standard deviation of the packet stream, respectively. For $\epsilon, \phi \in [0, 1]$, given $2\sigma^2 \left(\log 2 - \log \phi\right) / \left(\epsilon^2 \mu^2\right)$ sampled packets, the estimated average latency is bounded within $(1 \pm \epsilon)$ times the actual average latency holds true with a probability at least $(1 - \phi)$.*

We further discussed the sampled requirements for approximating the standard deviation, which can be found in the online supplemental material.

VII. Simulation

Having presented the theoretical results, we next evaluate the performance of RDA using real-world traces.

TABLE I
STATISTICS OF THE TRACE.

Metric	DC	Univ
packets	2,041,744	1,694,553
First packet	2009-12-18 00:26:04	2016-11-28 23:23:26

A. Simulation Setup

We built a simulator written in Java that replays real-world traces in the experiments, enforces different loss and reordering rates on the packets, and passively measures the average and the standard deviation using different approaches.

Data Set: The measurement of the latency requires packet traces that involve two monitoring endpoints with synchronized clocks. Unfortunately, no such traces are publicly available to the best of our knowledge. Therefore, we resort to the same network traffic traces collected at one endpoint used in existing studies [22], [28].

We use two data sets for the following simulation, as summarized in Table I:

- DC [7]: This trace is collected at routers that contain the arrival time and the packet header information for packets recorded in the ethernet interface.
- Univ: We collect packets at a server with the tcpdump tool [3] in a small data center located in our laboratory.

Delay Model: We set the delay distribution to the same with that used in LDA and FineComb [19], [22]. We draw the one-way delay using the Weibull delay distribution with cumulative distribution function $P(X \leq x) = 1 - \exp\left((-x/\alpha)^{\beta}\right)$ where α and β denote the shape and the scale parameters, respectively. We use the same shape parameter $0.6 \leq \alpha \leq 0.7$ used in the evaluation of LDA and FineComb.

Network Model: We follow the same loss and reordering models in FineComb [22]: The loss model simulates `random` packet losses, since each lost packet is mapped to a random bucket in the synopsis even for two back-to-back packets; the reordering model simulates the problematic reordered packets occurred at the beginning and the ending period of the measurement interval, while the reordered packets that arrive at the receiver within the same measurement interval does not affect the synopsis, since both endpoints record such packets.

Previous researchers have shown that the synopsis' performance is independent of the loss or reordering distribution [19], [22]. This is because the hashing operation randomizes the mapping locations of incoming packets, as a result, correlated lost or reordered packets are decorrelated after hashing to randomized locations. Therefore, random loss or reordering distributions should be sufficient for simulation.

Compared Methods: As our objective is to estimate the aggregated one-way latency under packet loss or reordering between a pair of measurement points, we compare RDA with two state-of-the-art methods LDA and FineComb. Other studies [27], [28] directly rely on LDA or FineComb to bypass buckets that contain lost or reordered packets. For fair comparison, we set the same storage size for these synopses. Further, FineComb maintains a stash of elements that is of the same size as the number of buckets as recommended in [22]. RDA maintains a cache of packets at both the sender and the receiver.

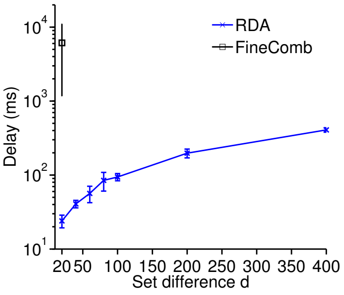


Fig. 6. The decoding time of RDA and FineComb.

Metrics: We evaluate the decoding efficiency and the measurement accuracy using two metrics: (i) **Decoding success probability**: the ratio between the number of packets decoded by RDA and the size of the set difference. (ii) **Relative error**: the ratio between the absolute value of the subtraction between the ground-truth metric f_g and the estimated metric f_e: $\frac{|f_g - f_e|}{f_g}$. The smaller the relative error, the closer the estimated metric to the ground truth.

All experiments are repeated in ten times. We report both the mean value and the 95-th confidence interval of the above two performance metrics. All experiments are performed on a PC with a Intel Core i7-3520 CPU (2.90GHz), 8 GB RAM and a Java runtime environment 1.7.0.

Due to space limitations, we briefly reported the comparison results, more simulation results are presented in the Appendix, which is available in the online supplemental material.

B. Results Comparison

For RDA, we set the number of hash functions to two and allocate an optimized number of buckets based on Theorem 2 when the size of the set difference is known, otherwise, we allocate a static number of buckets per bank. We use the recommended parameters for LDA [19] and FineComb [22]. We replay all packets in the trace.

1) Decoding Time

We compare the **decoding time** of RDA and FineComb. For RDA, we set the number of hash functions to two and the number of buckets to 4,000. For FineComb, we have implemented the proposed brute-force approach to repair the buckets.

Figure 6 shows that RDA is orders of magnitude faster than FineComb. We can see that the decoding time of RDA increases gracefully with increasing number of lost and reordered packets, however, the computation time for FineComb is unacceptably long even for a stash of size 40, since we need to enumerate any combinations of packets in the stash for each impaired bucket in FineComb.

As a result, for the rest of the evaluation, we implemented an "ideal" repairing procedure for FineComb, given that we know the actual set of lost and reordered packets that are inserted at each bucket. Note that this approach is not implementable in practice, as we will not know these values in a real scenario.

2) Varying Set Difference

We next compare the **relative error** of prediction results when the loss and reordering rates are unknown a priori.

We set the number k of hash functions to two and the number of buckets of RDA to 5,000. We dimension LDA and FineComb with the recommended parameters in [19] and [22]. We set the loss rate and the reordering rate to be identical with each other and vary the set difference from 0 to 10,000. Varying the loss or reordering rates changes the curves, but the same conclusion still holds. We report the average latency for brevity.

Figures 7 and 8 show the relative errors of the estimated average latency and the standard deviation. For RDA, some lower confidence-interval values are negative and not shown in the plots. We see that all methods increase the relative errors as we increase the set difference, since more packets are useless for the latency calculation. LDA has the highest relative error, since LDA is agnostic of the packets inserted into each bucket.

RDA consistently outperforms FineComb and LDA. For example, when the size of the set difference is smaller than 2,000, RDA's average-latency estimation incurs over 10 orders of magnitude smaller relative errors than those of FineComb and LDA. This is because when the set difference is smaller than 2,000, we can decode nearly all problematic packets.

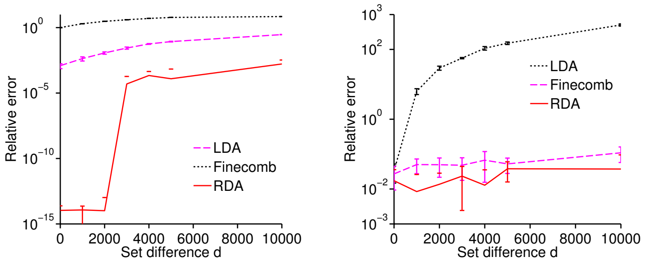
The real-world decoding performance is better than the bound provided by Theorem 2. From Theorem 2, we can see that when the size d of the set difference is smaller than $\frac{5{,}000}{2\cdot k} \approx 1,250$, the decoding fails with a probability at most $O(d^{-2})$.

Moreover, RDA's performance experiences a sharp transition with increasing problematic packets. after the set difference is greater than 2,000, RDA's relative error increases sharply, from 10^{-14} to around 10^{-4}, The sharp transition is due to a fraction of useless buckets that consist of lost or reordered packets. However, RDA is still much more accurate than FineComb since RDA can repair more packets than FineComb. Predicting the sharp transition is still an open problem, as estimating the number of failed decoding is challenging [18]. Our theoretical results in Section VI only loosely bound the failure probability of the decoding process.

We can see that the standard-deviation estimation is less accurate than the average metric, since the estimation of the standard deviation is not precise, while the average metric is an unbiased estimator of the ground-truth number. However, RDA's standard deviation estimation still incurs around 0.5 orders of magnitude smaller relative errors than that of FineComb, since RDA uses more packets to derive the standard deviation.

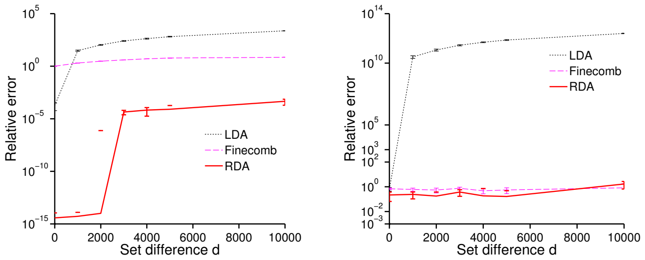
VIII. Implementation

We have implemented the RDA based passive latency aggregation in the software layer. This software captures packets using libpcap [3] from user space, maintains RDAs in the main memory and continuously calculates the latency analytics between a pair of measurement points.



(a) Average (b) Standard deviation

Fig. 7. The relative errors as we enlarge the set difference for LDA, FineComb and RDA on the DC data set. Negative numbers are omitted for the error bar due to the logarithmic-scale y-axis.



(a) Average (b) Standard deviation

Fig. 8. The relative errors as we enlarge the set difference for LDA, FineComb and RDA on the Univ data set. Negative numbers are omitted for the error bar due to the logarithmic-scale y-axis.

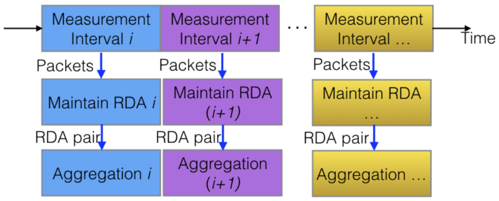


Fig. 9. The pipelined measurement interval.

A. Architecture

We continuously monitor the end to end latency via the pipelined measurement interval, as shown in Figure 9. At the beginning of a measurement interval, each measurement point keeps incoming packets into a separate RDA for this measurement interval. At the end of this measurement interval, the measurement point performs concurrent packet aggregation and triggers the next measurement interval.

The software can be flexibly deployed in data center networks. First, we may deploy the software on servers to track server-to-server one-way latency information. Second, we may place a pair of dedicated measurement nodes that passively capture streams of packets from mirror ports of a pair of switches, which should be less intrusive with respect to server performance.

Identifier Generation and Hash Function: In order to correctly aggregate the timestamp for each packet, hashing should ensure that different packets correspond to different identifiers. Therefore, the hash functions should minimize the collision probability of mapping different packets to identical identifiers, otherwise, the program will incorrectly aggregate the latencies for these packets. Further, as hashing computation consumes CPU cycles, we need to maximize the efficiency of the hashing algorithm to process as many packets as possible.

Each packet contains a number of checksums, including Ethernet-level, IP-level, and transport-level checksums. Unfortunately, checksums have several limitations to be used as identifiers: (i) *A packet's checksum may be modified*. First, as a packet usually traverses multiple Ethernet segments in data center networks, the Ethernet-level checksum needs to be recomputed for each Ethernet segment. Second, since the switch decreases the TTL field in the IP header per routing hop, the IP-level checksum also needs to be recomputed by the switch. Third, when a packet traverses NAT devices or transport congestion options are enabled, some transport-level fields need to be modified, as a result, the transport-level checksum needs to be recomputed as well. Consequently, modified checksums become useless to uniquely identify a packet. (ii) *Checksum calculation leads to high collision possibilities*. Checksums are used for checking errors in the packet header and payload, which is calculated by summing up packet contents for fast processing. Although the sum operator is faster than other alternatives, it increases the collision probability [4]. Consequently, two different packets may have identical checksums. (iii) *Captured packets may not have calculated checksums*. If the packet checksum computation is offloaded to hardware, then the packet checksum captured by libpcap becomes useless to uniquely identify a packet.

Henke et al. [16] have extensively studied the collision performance of a set of hash functions and found that the Bob hash function overall provides the best performance. In this paper, we choose the Bob hash function to create the identifier of each packet and to compute the bucket indexes. We assign a distinct 64-bit **identifier** for each unique packet in the packet stream using the distinct information of the packet, including the entire packet payload at the IP layer to minimize the overhead of extracting packet information.

Further, if NAT is used, then a packet's address becomes non-unique, therefore, we do not use the addresses of packets for creating the identifier. Moreover, if the packet header experiences modifications across the routing path, e.g., the TCP header is modified, then this packet would lead to two different identifiers at two measurement points. As a result, these two identifiers would become problematic for the latency aggregation. Fortunately, we can decode these problematic identifiers, since this pair of identifiers are equivalent to a lost packet and a reordered packet.

Pipelined Measurement Interval: In order to bootstrap the pipelined process, the software sets up the beginning timestamp of the first measurement interval for the sender and the receiver. To that end, the sender selects a timestamp that is larger than both clocks so that both the sender and receiver have enough free time to start the first measurement interval. Then the sender exchanges this timestamp with the receiver. Afterwards, both measurement points register the first measurement interval event at the specified timestamp and the successor measurement interval. When reaching the specified timestamp, both measurement points independently begin the measurement.

Late-binding RDA Maintenance: According to Theorem 2, we need to set the number m of buckets in each bank to twice the size of the set difference, in order to reconcile the lost and the reordered packets with failure probability at most $O(d^{-k})$. Unfortunately, the size of the set difference is unknown a priori until the measurement interval ends.

In order to decode all problematic packets, we adopt a **late-binding** approach to configure the RDA data structure. During the measurement interval, we extract the identifier of each intercepted packet, and store the corresponding timestamp into an in-memory hash table. *The receiver's RDA is transmitted to the sender during the latency aggregation procedure, while the cache is never transmitted and is flushed after the latency aggregation*.

After the measurement interval ends, we estimate the size of the set difference via the MinHash estimator [9], [10] based on a pair of caches at two measurement points. Next, we configure the RDA such that the number of buckets per bank amounts to twice the estimated size of the set difference. Finally, we insert each cached packet into the RDA data structure and trigger the latency aggregation procedure.

Latency Aggregation: After the measurement interval ends, the sender requests and obtains the receiver's RDA. When the receiver receives this request, it sends its RDA data structure immediately back to the sender. Then, the sender determines whether any of the buckets contain lost or reordered packets and decodes them if necessary. After the decoding process, the sender obtains a set of packets identifiers for lost and reordered packets.

Next, the sender needs to request the timestamps of the reordered packets from the receiver if any, since the lost packets are stored in the sender's cache, while the reordered packets are stored at the receiver's cache. Then, the sender deletes the lost packets from its own RDA using its own cache, and deletes the timestamps and the identifiers of the reordered packets from the receiver's RDA. Finally, the sender calculates the average and the standard deviation of the latency using the repaired RDAs.

Timing: During a measurement interval, the sender and the receiver aggregate the identifiers and the timestamps of packets. If two measurement points' clocks drift significantly, then the two measurements may not capture the same set of packets. Therefore, the measurement process heavily depends on a good time synchronization and precise event timing. We implemented the precise timing using IEEE 1588 Precision Time Protocol and events are timed using Linux high-resolution timers.

B. Parameter Configuration

Number of Hash Functions: The selection of the number of hash functions is a trade-off between the decoding failure probability and the time complexity. Let d be the total number of lost and reordered packets. When the number of buckets per bank is at least $2d$, the decoding failure probability decreases exponentially according to Theorem 2. Further, we empirically found that, when we fix the total number $k{\cdot}m$ of buckets in the RDA, increasing the number k of hash functions from one to two increases the percentage of decoded problematic packets, while more than two hash functions decreases the decoding probability. Therefore, we set the default number k of hash functions to two.

Bucket Width: We represent a timestamp using a 64-bit unsigned fixed-point value based on RFC-1305 [1]. We represent the Δ field using a 64-bit long integer. In addition, the ID field takes 64 bits as discussed in Subsection II-B. Thus, a bucket takes up to $64 \cdot 3 = 192$ bits. For an RDA with two banks and 100 buckets in each bank, the storage size is 4.69 KB.

Cache Size: To bound the size of the cache, we can simply adapt the number of stored packets according to the link speed or sample the incoming packets. By varying the sampling rate, we can adapt to different traffic rates. For example, with a 1% sampling rate on a 500,000 packets-per-second link, the sampled packet rate would become 5,000 packets per second in the worst case, and a measurement interval of 1 million packets would last at least 200 seconds. We need to ensure that both the sender and the receiver sample the same subset of packets in order to calculate the one-way delay using the sampled packets. To that end, we sample packets based on a flow identifier that uses the same hash function at the sender and the receiver, which enables the synopsis and the cache to record the same set of packet identifiers. The sampling process reduces the number of packets for latency measurement. As we are interested in the aggregated latency, we are still able to accurately compute the latency.

IX. Prototype Evaluation

In this section, we evaluate the performance of the prototype on two servers in the same rack. Both servers connect to a gigabit top-of-rack (ToR) switch with a 1 Gbits/s Ethernet network interface card. Each server has two Intel Xeon E5-2640 2.5 GHz processors with 12 threads and 48 GB RAM. A server runs iperf to generate traffic (5 TCP flows) to the other server. Other servers in the same rack have been allocated to several tenants that create a variety of background network traffic that compete for the processing capacity of the same rack-level switch.

Our prototype is multi-threaded event-driven, which incurs negligible performance reduction for co-located tasks on the server. We cache all packets in the measurement interval in order to obtain the finest-grained results. For RDA, we set the number of hash functions to two, and set the number of buckets per bank to twice the size of the set difference that is estimated by the Min-wise Estimator [9].

Validation: We first validate whether the software continuously produces useful results. We set the measurement interval to one second and thus report latency statistics of aggregate packets per second.

Figure 10 plots the dynamics of the average and the standard deviation of the one-way latency between the two servers. We can see that the software captures detailed fluctuations of the one-way latency: the average latency approximately centers around 0.05 ms, while the standard deviation varies from 0.05 ms to 1 ms. As a result, the software provides fine-grained information of underlying network flows.

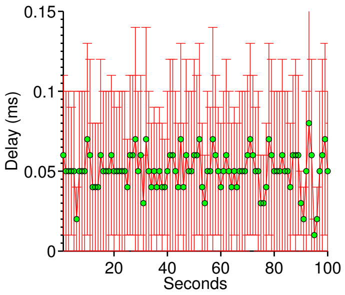
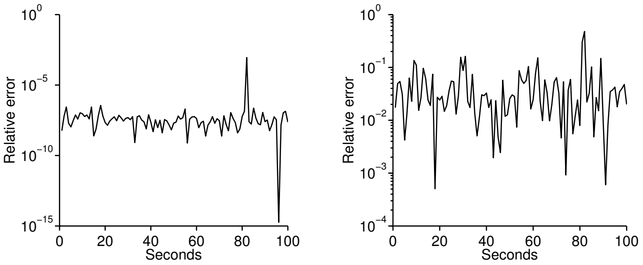


Fig. 10. One-way latency results between two servers reported by RDA based passive latency measurement software.



(a) Relative errors for the average results.

(b) Relative errors for the standard deviation results.

Fig. 11. The variations of relative errors of estimated average latency and the standard deviation.

We next verify the prediction quality of each measurement interval. We stored the cached packets into the disk and extracted the ground-truth one-way latency results by reading packet timestamps that are cached at the sender and the receiver. Figure 11 plots the dynamics of the relative errors of the software compared to the ground-truth one-way latency. We can see that the relative error of predicting the average latency incurs varies from 10^{-7} to 10^{-8}, while the relative error of predicting the standard deviation keeps around 0.01 to 0.1 in most cases. The accuracy varies primarily due to the dynamics of available packets and the percentage of repaired buckets, however, the relative error is low enough to monitor fine-grained one-way latency.

Further, we also aggregated the minimum RTT values via the ICMP protocol based `Ping` that should be insensitive to system noises, which yielded a minimum RTT 0.15 ms and a standard deviation 0.42 ms. As the routing path is symmetric, halving the minimum RTT approximates the one-way average latency in Figure 10.

Overhead: Having shown that RDA captures fine-grained latency dynamics, we next quantify the overhead of producing latency results: (i) **transmitting** the RDA to the other measurement point; (ii)**decoding** the set difference with a pair of RDAs; (iii) requesting **missing** timestamps of decoded packets; (iv) **erasing** timestamps of packets in the set difference; (iv) **computing** the latency statistics with a pair of repaired RDAs.

Figure 12 plots the stacked delays of these subprocesses for each measurement interval. We can see that the overall delay is around 20 ms, which enables fast network troubleshooting. Further, transmitting RDAs and requesting timestamps

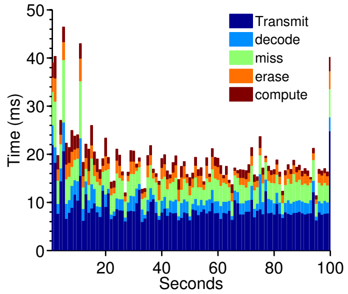


Fig. 12. Stacked delays to produce measurement results.

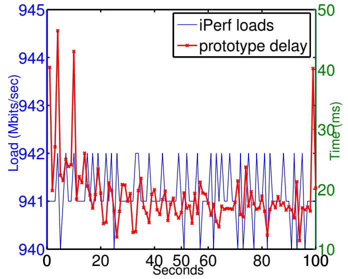


Fig. 13. iPerf generated loads (left y-axis) VS. Overall delays (right y-axis).

of missing packets takes up over 70% of the overall delay, while decoding the set difference, erasing the missing packets and computing the statistics take less time. Recall that the measurement interval is of one second, so a few tens of milliseconds incurs a low overhead.

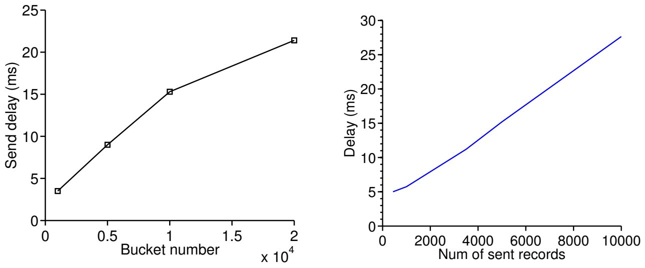
Next, we contrast the measurement overhead with network traffic between two servers. Figure 13 plots the overall delay to produce measurement results and the network traffic generated via the iPerf software. We can see that the traffic vary slightly around 940 to 942 Mbits per seconds, while the overall delay is approximately around 20 ms.

Scalability: Having illustrated that RDA produces useful latency information with modest overhead, we next test the system scalability.

(a) **RDA Transmission**: We first measure the variation of the time needed to send the RDA data structure with an increasing number of buckets. The time required to transmit the RDA data structure includes the processing time at the network stack of the two servers and the network transmission time. Figure 14(a) plots the dynamics of the time needed to transmit RDA between two servers. We can see that the time increases modestly with the number of buckets due to a compact design of the bucket structure.

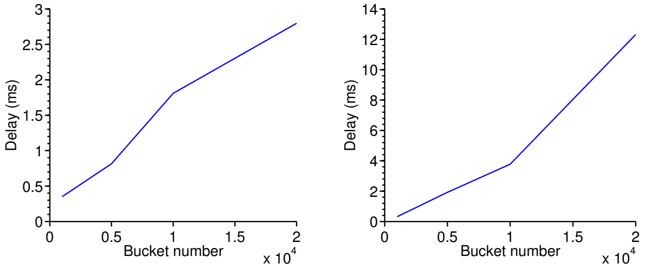
(b) **Missing-Packets Transmission**: We next test the variation of the time required to request missing timestamps of decoded packets. Figure 14(b) plots the requesting delay with an increasing number of sent records. We can see that the transmission delay increases linearly with respect to the number of records, as both timestamp and identifier need to be sent for each missing packet.

(c) **Latency Aggregation**: We next evaluate the processing scalability to calculate the average and the standard deviation of the latency as we increase the number of buckets.



(a) Sending RDA. (b) Sending packet records.

Fig. 14. The time of transmitting RDAs and that of sending the packet records that consist of the timestamps and identifiers of missing packets.



(a) Calculating the average metric. (b) Calculating the standard deviation.

Fig. 15. Computing delays of the average and those of the standard deviation with an increasing number of buckets.

Figure 15(a) plots the time required to calculate the average latency. We can see that the cost is modest, as averaging over 20,000 buckets only requires three ms. Further, the cost increases linearly with the number of buckets, since computing the average requires a linear scan of all buckets. Next, Figure 15(b) shows the time required to compute the standard deviation. We can see that the cost is super-linear to the number of buckets, since the standard deviation calculation needs several passes over the buckets.

X. Conclusions and Future Work

The synopsis based passive latency measurement approach scales well with increasing traffic volumes, however, the estimation accuracy degrades significantly under the presence of reordered or lost packets. Unfortunately, identifying these problematic packets from the synopsis is still a challenging problem. In this paper, we unify this problem within a set reconciliation framework that has been independently studied in the theoretical field. We propose a space-efficient synopsis named RDA that uses a multi-bank data structure to maximize the percentage of useful packets under the lost or reordered packets. RDA accurately estimates the average latency and the standard deviation. Our theoretical analysis shows that RDA preserves nearly all useful packets, while the space complexity is proportional to the number of packets that are lost or reordered. We designed and implemented a passive latency measurement system based on RDA. Our experimental results show that RDA obtains accurate latency statistics under the presence of loss and reordering events with modest overhead. As future work, we plan to extend RDA to support other tail statistics.

Acknowledgements

We would like to thank the anonymous reviewers for their constructive comments. This work was supported in part by the National Natural Science Foundation of China Grant No. 61402509, the National Basic Research Program of China (973) under Grant No. 2014CB340303.

References

[1] Network Time Protocol (Version 3) Specification, Implementation and Analysis. RFC 1305, 1992.

[2] Linux PTP Project. http://nwtime.org/projects/linuxptp/, 2016.

[3] TCPDUMP/LIBPCAP public repository. http://www.tcpdump.org, 2016.

[4] Bob Jenkins' Survey of Hash Functions. http://www.burtleburtle.net/bob/hash/doobs.html, 2017.

[5] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity, Data Center Network Architecture. In *Proc. of SIGCOMM*, pages 63–74, 2008.

[6] 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 SIGCOMM*, pages 63–74, 2010.

[7] T. Benson, A. Akella, and D. A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *Proc. of IMC*, pages 267–280, 2010.

[8] D. Borman, B. Braden, and V. Jacobson. TCP Extensions for High Performance. RFC 7323, Internet Engineering Task Force, 2014.

[9] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-Wise Independent Permutations. *Journal of Computer and System Sciences*, 60(3):630 – 659, 2000.

[10] E. David, T. G. Michael, U. Frank, and V. George. What's the Difference? Efficient Set Reconciliation without Prior Context. In *Proc. of SIGCOMM*, pages 218–229, 2011.

[11] J. Dean and L. A. Barroso. The Tail at Scale. *Commun. ACM*, 56(2):74–80, Feb. 2013.

[12] Y. Fu and Y. Wang. DKNNS: Scalable and Accurate Distributed K Nearest Neighbor Search for Latency-Sensitive Applications. *Science China Information Sciences*, 56:1–17, 2013.

[13] Y. Fu, Y. Wang, and E. Biersack. A General Scalable and Accurate Decentralized Level Monitoring Method for Large-scale Dynamic Service Provision in Hybrid Clouds. *Future Generation Comp. Syst.*, 29(5):1235–1253, 2013.

[14] Y. Fu, Y. Wang, and E. Biersack. HybridNN: An Accurate and Scalable Network Location Service based on the Inframetric Model. *Future Generation Comp. Syst.*, 29(6):1485–1504, 2013.

[15] Guo, Chuanxiong and Yuan, Lihua and Xiang, Dong and Dang, Yingnong and Huang, Ray and Maltz, Dave and Liu, Zhaoyi and Wang, Vin and Pang, Bin and Chen, Hua and Lin, Zhi-Wei and Kurien, Varugis. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proc. of SIGCOMM*, pages 139–152, 2015.

[16] C. Henke, C. Schmoll, and T. Zseby. Empirical Evaluation of Hash Functions for PacketID Generation in Sampled Multipoint Measurements. In *Proc. of PAM*, pages 197–206, 2009.

[17] V. Jalaparti, P. Bodik, S. Kandula, I. Menache, M. Rybalkin, and C. Yan. Speeding up distributed request-response workflows. In *Proc. of SIGCOMM*, pages 219–230, 2013.

[18] J. Jiang, M. Mitzenmacher, and J. Thaler. Parallel peeling algorithms. In *Proc. of SPAA*, pages 319–330, 2014.

[19] R. R. Kompella, K. Levchenko, A. C. Snoeren, and G. Varghese. Every microsecond counts: tracking fine-grain latencies with a lossy difference aggregator. In *Proc. of SIGCOMM*, pages 255–266, 2009.

[20] R. R. Kompella, K. Levchenko, A. C. Snoeren, and G. Varghese. Router Support for Fine-Grained Latency Measurements. *IEEE/ACM TRANSACTIONS ON NETWORKING*, 20(3):811–824, 2012.

[21] K. S. Lee, H. Wang, V. Shrivastav, and H. Weatherspoon. Globally Synchronized Time via Datacenter Networks. In *Proc. of SIGCOMM*, pages 454–467, 2016.

[22] M. Lee, S. Goldberg, R. R. Kompella, and G. Varghese. Fine-grained latency and loss measurements in the presence of reordering. In *Proc. of SIGMETRICS*, pages 329–340, 2011.

[23] W. Lewandowski, J. Azoubib, and W. J. Klepczynski. GPS: Primary Tool for Time Transfer. *IEEE Proceedings*, 87:163–172, 1999.

[24] X. Lu, H. Wang, J. Wang, J. Xu, and D. Li. Internet-based Virtual Computing Environment: Beyond the Data Center as a Computer. *Future Generation Comp. Syst.*, 29(1):309–322, 2013.

[25] H. Mi, H. Wang, Y. Zhou, M. R.-T. Lyu, and H. Cai. Toward Fine-Grained, Unsupervised, Scalable Performance Diagnosis for Production Cloud Computing Systems. *IEEE Trans. Parallel Distrib. Syst.*, 24(6):1245–1255, 2013.

[26] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, Low Latency Scheduling. In *Proc. of SOSP*, pages 69–84, 2013.

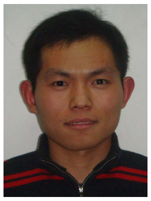
[27] J. Sanjuàs-Cuxart, P. Barlet-Ros, N. G. Duffield, and R. R. Kompella. Sketching the Delay: Tracking Temporally Uncorrelated Flow-level Latencies. In *Proc. of IMC*, pages 483–498, 2011.

[28] M. Shahzad and A. X. Liu. Noise can Help: Accurate and Efficient Per-flow Latency Measurement without Packet Probing and Time Stamping. In *Proc. of SIGMETRICS*, pages 207–219, 2014.

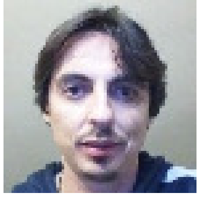
[29] H. Wang, P. Shi, and Y. Zhang. Jointcloud: A cross-cloud cooperation architecture for integrated internet service customization. In *37th IEEE International Conference on Distributed Computing Systems, ICDCS 2017, Atlanta, GA, USA, June 5-8, 2017*, pages 1846–1855, 2017.

[30] Z. Wu, C. Yu, and H. V. Madhyastha. CosTLO: Cost-effective Redundancy for Lower Latency Variance on Cloud Storage Services. In *Proc. of NSDI*, pages 543–557, 2015.

[31] Y. Zhu, N. Kang, J. Cao, A. G. Greenberg, G. Lu, R. Mahajan, D. A. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, and H. Zheng. Packet-Level Telemetry in Large Datacenter Networks. In *Proc. of SIGCOMM*, pages 479–491, 2015.



Yongquan Fu received the B.E. degree in computer science and technology from Shandong University, Jinan, China in 2005, and the M.S. and Ph.D. degrees in computer science and technology from National University of Defense Technology, Changsha, China in 2007 and 2012, respectively. Since 2013, he has been with the Science and Technology Laboratory of Parallel and Distributed Processing, College of Computer, National University of Defense Technology, where he is currently a lecturer. His research interests include network measurement, social networks, and distributed systems.



Pere Barlet-Ros received the M.Sc. and Ph.D. degrees in Computer Science from the Universität Politècnica de Catalunya (UPC) in 2003 and 2008, respectively. He is currently an Associate Professor with the Computer Architecture Department of UPC and co-founder of Talaia Networks, a University spin-off that develops innovative network monitoring products. His research interests are in the fields of network monitoring, traffic classification, and anomaly detection.

Dongsheng Li received the B.Sc. and Ph.D. degrees (Hons.) in computer science from the College of Computer Science, National University of Defense Technology, Changsha, China, in 1999 and 2005, respectively. He is currently a Full Professor with the National Laboratory for Parallel and Distributed Processing, National University of Defense Technology. His research interests include distributed computing, cloud computing, computer network, and large-scale data management. He was awarded the prize of the National Excellent Doctoral Dissertation of China by Ministry of Education of China in 2008.