© 2023 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes,creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. DOI 10.1109/TNET.2023.3251981


# A One-pass Clustering based Sketch Method for Network Monitoring

Yongquan Fu, Lun An, Siqi Shen, Kai Chen, Pere Barlet-Ros

**Network monitoring solutions need to cope with increasing network traffic volumes, as a result, sketch-based monitoring methods have been extensively studied to trade accuracy for memory scalability and storage reduction. However, sketches are sensitive to skewness in network flow distributions due to hash collisions, and need complicated performance optimization to adapt to line-rate packet streams.**

**We provide Jellyfish, an efficient sketch method that performs one-pass clustering over the network stream. One-pass clustering is realized by adapting the monitoring granularity from the whole network flow to fragments called subflows, which not only reduces the ingestion rate but also provides an efficient intermediate representation for the input to the sketch. Jellyfish provides the network-flow level query interface by reconstructing the network-flow level counters by merging subflow records from the same network flow. We provide probabilistic analysis of the expected accuracy of both existing sketch methods and Jellyfish. Real-world trace-driven experiments show that Jellyfish reduces the average estimation errors by up to six orders of magnitude for per-flow queries, by six orders of magnitude for entropy queries, and up to ten times for heavy-hitter queries.**

***Index Terms*—subflow, sketch, hash collision, clustering, heavy hitter**

## I. Introduction

Network monitoring is of paramount importance for traffic engineering, network diagnosis, network forensics, and intrusion detection and prevention in clouds and data centers, which needs diverse measurement tasks [1], [2], such as flow size estimation, flow distribution, and heavy hitter detection. Yet, network monitoring is challenging due to the ever-increasing line rates, massive traffic volumes, and large number of active flows.

Network-monitoring tasks require advanced data structures and algorithms. Many space- and time-efficient approaches have been proposed in the literature, e.g., traffic sampling (e.g., Netflow, sflow), traffic counting [3], [4], traffic sketching [5], [6], [7]. Traffic sketching is an increasingly popular method due to their better trade-off between space requirements and capability of ingesting all packet records to a constant-size data structure.

The sketch maps each incoming packet to a bucket in the bucket array, which is indexed by the hash of the corresponding network flow identifier. Thus all packets from the same network flow would be mapped to the same bucket, which approximately preserves the network-flow counter. Hash collisions are prevalent in the sketch, which are caused by mapping multiple keys to the same sketch entries. Due to the randomness of the hash functions, the probability of hash collisions increases as more keys are hashed into the fixed-size data structure.

The sketch-based network monitoring process incurs a high degree of approximation errors, thus the common approach is to either enlarge the size of the sketch structure, or to maintain multiple independent sketching instances for the purpose of choosing the sketch entries with the fewest hash collisions. Yet, real-world traffic distributions are usually non-uniform, so that the approximation errors are likely to be amplified by the long tails of network-flow distributions.

A natural question to ask is whether we can reduce the approximation errors by replacing key-based hashing process with smart locality-sensitive mapping methods. Several approaches [8], [9] proposed to separate large network flows from the rest to reduce the peak estimation error. LSS [10] maps similar network counters to the same bucket array in order to reduce the approximation error. Unfortunately, as the network flow counters keep increasing, it is challenging to track exact sizes of all active network flows in the main memory, and dynamically map them to different buckets according to their sizes.

This paper proposes the *subflow*, an aggregated metric composed of a subset of packets from the same network flow. The granularity of the subflow can be adjusted with a flexible threshold parameter that bounds the maximal subflow counter. Reducing the thresholds smooths the subflow distributions, but increases the number of subflow records for reconstructing the network-flow level counters. As a result, we can set suitable subflow thresholds for different network-flow distributions.

Replacing network-flow level sketching with subflow-level sketching is non-trivial. A strawman approach for subflow-based sketching is to use conventional sketching methods such as count-min [11] and count-sketch [12] methods. Our experiments in Subsection V-B show that these sketching methods still incur a high degree of approximation errors, since truncated subflow counters are still non-uniform.

We next propose Jellyfish, a subflow stream sketching framework. First, Jellyfish generates subflow records from

This work was sponsored in part by the National Natural Science Foundation of China (NSFC) under Grant No. 61972409, Hong Kong RGC TRS T41-603/20-R, GRF-16213621, ITF ACCESS. This publication is also part of the Spanish I+D+i project TRAINER-A (ref. PID2020-118011GB-C21), funded by MCIN/AEI/10.13039/501100011033. This work also received support by the Catalan Institution for Research and Advanced Studies (ICREA Academia), PCL2022A03-03. *Co-first authors: Yongquan Fu, Lun An. Corresponding authors: Siqi Shen, Kai Chen.*
Yongquan Fu *(yongquanf@nudt.edu.cn)* is with the Science and Technology Laboratory of Parallel and Distributed Processing; College of Computer, National University of Defense Technology, Changsha, Hunan province, China. Lun An *(anlun@bupt.edu.cn)* is with School of Cyberspace Security, Beijing University of Posts and Telecommunications, Beijing, China. Siqi Shen *(siqishen@xmu.edu.cn)* is with Fujian Key Lab of Sensing and Computing for Smart Cities, School of Informatics, Xiamen University, China. Kai Chen *(kaichen@cse.ust.hk)* is with Hong Kong University of Science and Technology, China. Pere Barlet-Ros *(pbarlet@ac.upc.edu)* is with Universitat Politècnica de Catalunya, Barcelona, Spain.

TABLE I
KEY NOTATIONS.

| Notation | Meaning |
|---|---|
| $I$ | Bucket array |
| $X$ | Network-flow counters |
| $\hat{X}$ | Estimated network-flow counters |
| $A$ | Indicator matrix of mapping $X$ to $I$ |
| $N$ | Number of keys |
| $k$ | Number of cluster centers |
| $m$ | Number of buckets |
| $\tau$ | Subflow threshold |
| $f_s$ | Cardinality of subflows of a network flow |

packet streams with a high-performance hash table. Second, Jellyfish maps similar subflow counters to the same bucket array. Third, Jellyfish postpones the reconstruction of the network flow counter during the query process, where each network-flow counter is calculated by the sum of subflow counters estimated from the bucket array for the same network flow.

We perform extensive evaluation in Section V. Real-world trace-driven experiments show that Jellyfish outperforms state-of-the-art sketching methods, with up to $10^6\times$ reduction in average relative errors for per-flow queries, $10^6\times$ reduction in average relative errors for entropy queries, and up to $10\times$ reduction in average F1 scores for heavy-hitter queries. Further, Jellyfish delivers messages within 60ms in most cases, thus the message bus based decoupled sketching framework provides timely delivery in most time.

We summarize our contributions as follows: (i) We present a subflow sketching method Jellyfish. (ii) We present a subflow based network monitoring framework that generates tunable subflow records from packet streams and decouples the sketch structure from the ingestion components to adapt it to the line rates. (iii) We conduct extensive experiments with real-world data sets to show that the proposed sketch method obtains accurate and robust estimation results.

The rest of the paper is organized as follows. Sec. II summarizes related studies on sketch-based network monitoring methods and provides the problem model on the sketch-based network flow monitoring process. Sec. III next presents a practical subflow-stream sketch. Sec. IV presents the implementation details. Sec. V conducts extensive performance evaluation with real-world datasets. We finally conclude in Sec. VI. Table I lists key notations.

## II. PROBLEM STATEMENT

We present the background for the network monitoring process, and the related work that are most related with us. We next analyze the challenges of the sketching approaches.

### A. Sketch based Network Monitoring

For a cloud data center with hundreds of heterogeneous physical and virtual network devices and links, we would like to track network-flow distributions and detect potential network heavy hitters for anomaly detection in real-time. There are two schemes for network monitoring on network flows. For the sequence based sliding window, each interval keeps at most $N$ flow records, and a new interval is generated afterwards; while for a time based window, each interval records the packets during a fixed time period, and a new interval is created after the interval ends. Although these network devices typically provide Netflow or sflow records, the monitoring output is too coarse for real-time monitoring and diagnosis purpose. A more suitable approach would track approximate network flow counters in near real time at low memory footprint.

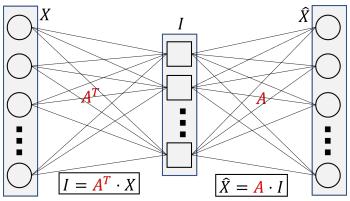



Fig. 1. Insertion ($I = A^T \cdot X$) and query ($\hat{X} = A \cdot I$) processes of a list of network-flow counters $X$ over a bucket array $I$.

A network flow is typically represented as a key-value pair, where the key is defined as the composition of several essential packet-header fields, and the value is defined as the flow's current statistics, e.g., number of packets or flow bytes. A sketch-based monitor inspects the packet header to extract the key and calculate the packet's value, then inserts this record to the sketch data structure. A sketch-based monitoring application typically comprises an *ingestion* component that intercepts incoming packets from the physical network interface and generates key-value input for the sketch, a *sketching* component that feeds the key-value input to a sketch structure that approximates these key-value pairs with one or multiple hash based bucket arrays.

A sketch typically consists of a bank $k$ of bucket arrays of length $m$, where a bucket keeps a counter field. In order to insert an incoming key to the sketch, we first hash the key to $k$ random numbers within the interval from one to $m$ with $k$ *independent* random hash functions, then select the buckets indexed by these random numbers from $k$ banks, and finally adjust the counters of these buckets based on the value of the incoming key. The query process is similar to the insertion process based on the same set of hash functions.

Suppose that a sketch structure randomly maps incoming items to a bucket array uniformly at random. Let $X : N \times 1$ denote the vector of the streaming key-value sequence from the network ingestion component. Let $\mathrm{A} : N \times m$ denote the indicator matrix of mapping the vector $X$ to a bucket array $a$ of size $m \times 1$. Let $\mathrm{A}(i,j) = 1$ iff the $i$-th item $X_i$ is mapped to the $j$-th bucket $I_j$, and $\mathrm{A}(i,l) = 0$ for $l \neq j, l \in [1,m]$. A sketch with one bucket array consists of two phases: the insertion process solves $I = \left(A^T X\right)$, while the query phase solves $\hat{X} = A \cdot I$, as shown in Figure 1. Thus the approximated counters of a sketch can be represented as:

$$\hat{X} = AA^T X \tag{1}$$

We can formulate the approximation error between $X$ and $\hat{X}$ for the sketch:

$$\min_A \left\| X - \hat{X} \right\| = \left\| X - AA^T X \right\| \tag{2}$$

where $\|\cdot\|$ denotes a chosen metric such as the Frobenius norm. To derive optimized solution for Eq (2), assuming that the mapping matrix is a random variable, the Principle Component Analysis (PCA) [13] finds a dimensional-reduction hyperplane with the smallest reconstruction error for a one hidden-layer autoencoder. Unfortunately, it will require to keep the whole stream $X$, which is infeasible for network monitoring context. Moreover, PCA calculates a dense matrix $A$, while the sketch enforces the matrix $A$ to be ultra-sparse.

As we hope to represent the traffic records with low storage overhead, the size of the sketch should be on the orders of magnitude smaller than the number of flow records. Otherwise, the sketch becomes less space-efficient. We can show that (Lemma 1) the hash collisions for each item in the stream follows a binomial distribution $B(\frac{N-1}{m}, \frac{(N-1)\cdot\left(1-\frac{1}{m}\right)}{m})$. The proofs are put in the appendix fore brevity and henceforth.

**Lemma 1.** *Let $\varrho_i$ denote the number of hash collisions with item $i$. The variable $\varrho_i$ follows the binomial distribution with expectation $E[\varrho_i] = \frac{N-1}{m}$, and variance $Var[\varrho_i] = \frac{(N-1)\cdot\left(1-\frac{1}{m}\right)}{m}$.*

### B. Related Work

The sketch has received extensive attentions due to their competitive trade off between space resource consumption and query efficiency. To reduce the approximation error, a popular trick is to choose the least affected bucket from multiple copies of independent bucket arrays as the estimator. UnivMon [6] uses an array of sketch counters to meet generic monitoring tasks. SketchVisor [14] augments the sketch with a fast-path ingestion path to tolerate bursty traffic. ElasticSketch [8] keeps heavy hitters separately with a hash table, and puts the rest of items to a count-min sketch. Thus it is less sensitive to heavy hitters compared to prior sketch structures [12], [11], [15]. SketchLearn [9] separates large flows from the rest based on inferred flow distributions, which incurs additional processing delay for each packet. Nitrosketch [16] reduces the insertion frequency to relieve the processing delay, but introduces uncertainty on the sketching results. OmniMon [17] seeks full accuracy and resource efficiency over collaborated network entities. LSS [10] applies a cluster-preserving approach to reduce the estimation error, by guaranteeing that each network flow is always mapped to the closest cluster. Thus, to process packet streams like traditional synopsis, LSS maintains an in-memory cache for active network flows and dynamically adjusts the clusters for these network flows. Our work dramatically improves the prediction accuracy and memory efficiency using a subflow clustering based sketch method, which avoids the need of adjusting the buckets for active network flows. When full-coverage is hard to reach, Jang et al. [18] presented efficient sampling techniques to preserve important packets to reduce the information loss, which is complementary to full-coverage packet measurements.

### C. Challenges

A sketch-based streaming should meet monitoring accuracy and performance needs.

#### 1) Sketch Approximation via Clustering

The sketch can be viewed as a clustering model. We define a least-square loss function based on Eq (2) by allowing for different mapping matrices for the insertion and query processes:

$$\min_{B} \left\| X - BA^T X \right\| \tag{3}$$

where $A$ denotes the mapping matrix of the insertion process. We next show that this modified loss function is equivalent to the K-means clustering model.

**Lemma 2.** *The minimum-error approximation in Eq* (3) *is equivalent to that the K-means clustering problem that seeks to partition items to a set of groups with minimal variance.*

Based on the clustering interpretation, we see that the mapping matrix $A$ in a sketch can be seen as the clustering indicator in the K-means clustering model. Suppose that we pre-train the mapping matrix $A$ to group similar items to the same bucket, then we the average of the bucket counter, i.e., the cluster centroid will approximate the original items closely compared to methods ignoring the similarity between item counters.

However, clustering over packet streams is challenging. For each incoming packet of a network flow, we have to accumulate this network-flow counter and dynamically adjust the clustering decision for this network flow, since we should guarantee that this network-flow counter is always mapped to the nearest cluster center. The cost of updating these network-flow counters is proportional to the line rates, which does not scale well.

#### 2) Relative Errors and Hash Collisions

Generally, we can quantify the expected number of noisy buckets that have hash collisions. Assume that each key is mapped to a bucket in each bank uniformly at random with the hash function. As Lemma 3 shows that, the probability of hash collisions increases fast with decreasing ratios between the number of buckets and the number of inserted keys.

**Lemma 3.** *Let $m$ denote the number of buckets, $N$ the number of unique keys. For a sketch with $c$ banks of bucket arrays, where each bucket array is of size $\frac{m}{c}$, the expected percent of noisy buckets is $1 - e^{-cN/m} - \frac{cN}{m} \cdot e^{-c(N-1)/m}$.*

We provide lower bounds for both the expectation and the variance of the sum of the approximation errors as follows:

**Lemma 4.** *Let $\mu$, $\sigma^2$ denote the expectation and the variance of the records in $X$, respectively. The sum of the approximation errors in a bucket array satisfies that: (i) the lower bound of its expectation is $\mu(\frac{N^2}{m} - N)$, and (ii) the lower bound of its variance is $\sigma^2(\frac{N^2}{m} - N)$.*

We see that the variance of the approximation error is several times of the variance of the whole stream when $m \ll N$. Therefore, in order to increase the approximation accuracy, we need to mitigate the variance of items that are mapped to the same bucket array.

## III. JELLYFISH: SUBFLOW SKETCH

Having presented the background and challenges for sketch based measurements, we next present the subflow model and the Jellyfish framework, and then present strawman approaches based on popular sketch methods. Next, we present the insertion and query operations. Finally, we present sketching applications and the performance analysis.

### A. Design Goals

Jellyfish seeks to reduce the approximation error by clustering and and improve the sketching efficiency by subflow modeling of raw packets that decouples the packet arrival rate with the sketch data structure.

(i) *Robust over network flow distributions with online clustering model*. The variance of prior sketches is sensitive to incoming flowkeys. While a small number of outlier flowkeys can significantly increase both the trend and the variance of the approximation error for many buckets. As the sketch is equivalent to a clustering process, we can reduce the estimation variance by group similar flowkeys to the same bucket array with clustering algorithms. However, clustering dynamic flowkeys is challenging since the distribution is nondeterministic in general. As a result, we need a scalble approach to bound the distribution of dynamic flowkeys for efficient and online clustering.

(ii) *Supporting diverse sketching innovations with a subflow model*. It is inefficient to cluster dynamic packets due to duplicated network flow records embedded in the packet stream. In contrast, we can create subflow record that aggregates a bounded number of packets and cluster static subflow records without the concerns of duplicates of network flow records. Moreover, the sketch can adapt to the scale of flowkeys, not packet rates, as the packet rate is typically much smaller than the network-flow rate. Thus the subflow records leave plenty of algorithmic innovations for sketches.

### B. Subflow Ingestion

The subflow ingestion framework puts aggregated records of fragments of network flows to the stream of subflow records. A subflow represents an aggregated key-value record for a subset of consecutive packets from the same network flow, where the key identifies this subflow, and the value accumulates the counters of these packets.

*1) Subflow Naming and Membership*

A *subflow-tracker* is responsible to keep the membership of subflows for different network flows. We need to name each subflow record and link the subflow records to different network flows.

First, we identify each subflow record by appending a monotonically-ascending index function Index($x$) to the original network-flow record $x$. Thus, each subflow record now has a distinct representation: for a network-flow key $x$, the subflow identifier is of the form: recordID($x$) = "($x$, Index($x$))", where Index($x$) returns a monotonically-increasing index for the identifier $x$.

As we use the ascending index to identify each subflow, we can trivially get the total number of subflow records for each network flow: We can keep the number of subflow records by keeping an integer-valued subflow cardinality number $f_s$ for each network flow. The parameter $f_s$ accounts for the current number of subflow records. Suppose that $f_s$ amounts to three for a network-flow $x$, then we have inserted three subflow records with keys "($x$, 1)", "($x$, 2)", "($x$, 3)". As the parameter $f_s$ keeps the cardinality of the subflow records for a specific network flow, thus keeping $N$ network flows just needs $N$ integer numbers, i.e., linear to the number of network flows.

Second, we use an approximate data structure to keep the membership of subflows for different network flows in order to avoid keeping an in-memory cache. We attach a Cuckoo filter to track the subflow records, and insert the identifiers of subflow records to the corresponding cuckoo filter in a dynamic approach. The cuckoo filter supports efficient insertion and deletion of items, and is more efficient than the Bloom filter at low false positives [19], [20], [21]. False positives occur when multiple subflow records may be mapped to the same slot in the Cuckoo filter, as we would like to reduce the storage cost at the expense of introducing a degree of false positives. We may use more space-optimized data structures for subflow's membership, such as the count-min sketch, at the cost of introducing a degree of approximation errors for the subflow membership.

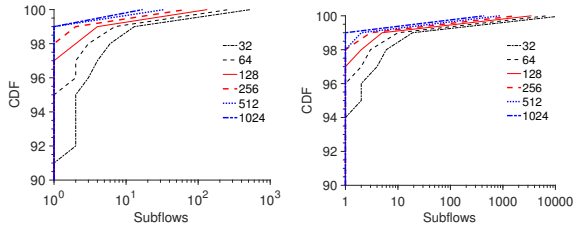
As the cuckoo filter keeps every subflow records, we can obtain the number $f_s$ of subflow records for each network flow by an aggregation approach: It first generate a subflow index $y$ for each network flow $x$ that is initialized to one and increases by one at each round, and next concatenate the subflow identifier as ($x$, $y$), and query the cuckoo filter to test whether this subflow identifier is inserted to this cuckoo filter, if the cuckoo filter returns true for the subflow membership query, then we claim that this subflow identifier is in the cuckoo filter and the variable $f_s \geq y$ and we continue the query with incremented subflow index $y$, otherwise, we claim that $f_s = y - 1$ and returns the subflow record. Knowing $f_s$ immediately enables us to get the set of subflow identifiers. As a result, we can use query to save the storage of the subflow cardinality number $f_s$.

*2) Subflow Generation*

We aggregate packets into the subflow into a key-valued hash table, based on the threshold constant $\tau$ to generate the subflow record. As long as a subflow's accumulated counter exceeds this threshold $\tau$ (128 by default) or a maximal waiting period expires, we publish this subflow immediately. As a result, the subflow record's counter is at most $\tau$. The threshold $\tau$ is adjustable to account for the distribution.

The subflow records of all network flows will be clustered altogether and each cluster will be inserted to the same Jellyfish bucket array. Thus we need to use the same threshold for all network flows to ensure low variance of subflow records in the same bucket array. Identical or similar subflow-valued records will be clustered to the same bucket array to optimize the estimation variance. Since the root causes of the estimation errors are the variance of the input values for Jellyfish.

We need to balance the estimation accuracy and the subflow storage by the choice of the subflow threshold $\tau$. Reducing the threshold $\tau$ will reduce the range of the variance by




(a) CAIDA (b) MAWI

Fig. 2. The cumulative distribution function (CDF) of the numbers of subflows per network flow. The x-axis is plotted in logarithmic scale.

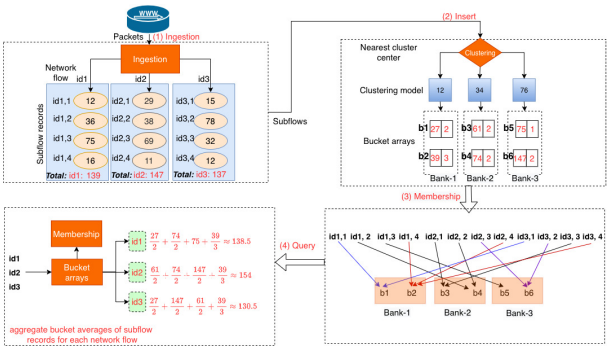



Fig. 3. Overall framework of Jellyfish.

subflow records, which increases the estimation accuracy of Jellyfish. Since Jellyfish needs to track the membership of subflow records in order to reconstruct a network flow, we need more storage with increasing numbers of subflow records as we reduce the threshold $\tau$.

We next count the number of subflows per network flow for different subflow thresholds $\tau$. Figure 2 shows the cumulative distribution functions of the numbers of subflows. We see that over 90% of network flows only have one subflow tuple, since the majority of network flows are small flows. Further, as we increase the subflow threshold $\tau$ from 32 to 128, 99% of the numbers of subflows decrements from 13 to 4 for the CAIDA data set, and decrements from 19 to 5 for the MAWI data set, respectively.

### C. Overview

Jellyfish generates subflow records by aggregating the counters of a subset of consecutive packets from the same network flow. Next, Jellyfish clusters similar subflow records to bucket arrays with respect to an online subflow-clustering model.

Jellyfish puts each subflow record into a bucket array indexed by the closest cluster center towards this record. Next, Jellyfish selects a random bucket from this bucket array, by hashing the subflow key with one hash function, and accumulates the subflow counter to this bucket. Finally, Jellyfish reconstructs the counters for each network flow by estimating each subflow record belonging to this network flow and merging those to the final estimation result.

Figure 3 shows the overall framework of the Jellyfish based network monitoring process: Three network flows denoted as id1, id2 and id3 are divided to subflow records, each of which consists of four subflow records. The sizes of network

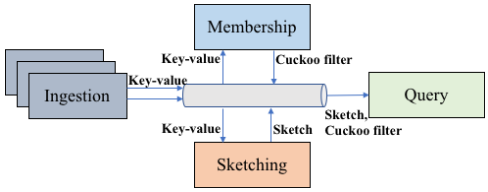



Fig. 4. Key functions in a subflow based network-monitoring architecture.

**Algorithm 1:** Insert operation on the Jellyfish.

1 `Insert` $(x, v)$
**input :** Network-flow key $x$, subflow cardinality field Index($x$), subflow counter $v$.
2 Generate the key $\kappa$ = ($x$, Index($x$));
3 Find the bucket array index $i_\kappa = argmin_i \|v - \mu_i\|$ to the nearest cluster center;
4 Update the bucket indexed at $h(\kappa)$: $I_{i_\kappa}[h(\kappa)].sum+ = v$, and $I_{i_\kappa}[h(\kappa)].count$ +=1;
5 Insert $\kappa$ to the $i_\kappa$-th Cuckoo filter;

flows denoted as id1, id2 and id3 equal the sums of the corresponding subflow records belonged to the same network flow, which are 139, 147 and 137, respectively. Subflow records are clustered based on a clustering model (12, 34, 76). Each subflow record is mapped to the bucket array indexed by the nearest cluster center. Next, a bucket is chosen from this bucket array based on the hash of the subflow identifier with a hash function. Finally, the *sum* field of this bucket is increased by the incoming subflow record, and the *count* field of this bucket is incremented by one. The membership of subflows are kept for query purpose. Step (4): Three network flows are estimated based on the sum of queried results of the corresponding subflow records. The estimation results for id1, id2 and id3 are 138.5, 154 and 130.5, respectively.

From Figure 3, we see that the approximation results closely match the original network flow counters. Moreover, Jellyfish does not need an in-memory cache of historical samples, or a real-time update policy to dynamically adjust the clustering positions of evolving network flow counters.

Figure 4 summarizes key functions in the subflow-based network monitoring process with ingestion, membership, insertion and query functions. The ingestion function produces subflow records. The membership function keeps the mapping of subflow records to bucket arrays. The insertion function inserts subflow records to bucket arrays. Finally, the query function enables the query over network flows.

### D. Details of Jellyfish

Jellyfish mixes subflow records into a bucket array. Thus the physical layout is comprised of a number $k$ of bucket arrays. A bucket array consists of a number of buckets, where each bucket has two fields: (i) A *sum* field that records the sum of subflow values mapped to this bucket; (ii) A *count* field that records the number of subflow records inserted to this bucket. As each subflow has a unique identifier, thus the *count* just increments by one for each incoming subflow record.

**Algorithm 2:** Query operation on the Jellyfish.

```
1 Query(x)
  input : network-flow key x.
  output: Network-flow counter v.
2 Obtain the cardinality field f_s (denoted as x_s) of the key x
    from the subflow tracker;
3 SumCounter = 0;
4 for j = 1 → x_s do
5     Generate the key κ = (x, j);
6     Obtain the bucket-array index i_κ for κ in the bank of
        Cuckoo filters;
7     SumCounter += I_{i_κ}[h(κ)].sum / I_{i_κ}[h(κ)].count;
8 return SumCounter;
```

*1) Insert*

Based on the equivalence between the sketch and the clustering analysis in Sec. II-C1, Jellyfish maps each subflow record $(x,\ v)$ to the nearest cluster center with respect to $v$. We present the insertion process in Algorithm 1. We select the bucket array corresponding to the cluster index of the incoming record $x$, then we find a bucket in this bucket array with the hash of the key $\kappa$ = recordID($x$), and increment the counter of the mapped bucket by the hash of the key $\kappa$ with the subflow counter $v$: *sum* = *sum* + $v$, and *count* = *count* + 1.

**Time Complexity**: The time to query the nearest cluster center is $O(\log k)$ with a binary search tree over the sorted array of $k$ cluster centers. Next, we need one hashing evaluation to locate a bucket in a single bucket array. Finally, we insert the subflow key to the corresponding Cuckoo filter, which can be run concurrently with the bucket array.

During the insertion phase, Jellyfish avoids the maintenance of the volatile network-flow counters in memory, and does not need to adjust the mapping between network flows and the clustering model as well as the bucket arrays. The downside of these benefits, is that Jellyfish needs to save the subflow membership instead of the network-flow membership.

*2) Query*

We next present the query process in Algorithm 2. For a network-flow key $x$, we first query the cardinality field $f_s$ for key $x$ via the subflow tracker, then we construct the set of keys $KEYs(x)$: "$(x,\ 1)$", "$(x,\ 2)$", ..., "$(x,\ f_s)$". Next, for each key $y$ in $KEYs(x)$, we locate the index of the bucket array by querying the Cuckoo filter associated with each bucket array. Next, we select the bucket in the corresponding bucket array with the hash of the key, and return the division $\frac{sum}{count}$ as the approximate counter for this key $y$. Finally, we calculate the sum of approximate counters for each key in $KEYs(x)$, and return this number as the counter for the network flow $x$.

The locations of different subflow keys are independently kept by the cuckoo filter. Suppose that two subflow records of the same network flow are put to the same bucket in varying orders, then Jellyfish will correctly keep the locations of these subflow records in different cuckoo filter records. Jellyfish can correctly recover the counters afterwards: it will first locate the index of two subflow records based on the cuckoo filter and the hash value of two subflow keys, and then calculate the averaged number of the bucket and accumulate these counters of two subflow records since both belong to the same network flow.

**Time Complexity**: During the query phase, we need to obtain the index for each of the subflow key by querying the bank of the Cuckoo filters at $O(kf_s)$ time with respect to $k$ bucket arrays and $f_s$ subflow records. As the query phase is not in the critical path of the network-monitoring application, the speed of the query phase is much less important than that of the insertion phase.

The query process satisfies the *eventual consistency* model [22]: if no new subflow records are generated to a given network flow, then eventually all queries to this network flow will return the result calculated after the last updated value. The eventual consistency ensures the correctness of the query process for network flows. As most network flows are short, over 90% of network flows have less than ten subflow records, the delay to reach the eventual consistency is short for most network flows.

Large network flows may have additional subflow records temporally maintained by the ingestion component, but are not yet inserted to the sketch. Thus the sketch returns the results for outstanding network flows by the last insertion. As a result, we need to combine the results of both the sketch and the hash-table values at the ingestion component, in order to provide real-time result for the corresponding network flow: First, we query the sketch to obtain the network-flow estimation counter from the last insertion. Second, we query the hash table via the network-flow identifier. Third, we sum up these two counters as the estimation result for the network flow counter.

### *E. One-pass Online Clustering Model*

As we fix the subflow threshold, the network flow distribution is bounded. Online clustering is desirable over an offline clustering model that may not be outdated. As it is costly to create a new clustering model before each measurement period. We maintain the subflow clustering model based the K-means clustering method that consists of $k$ cluster centers, due to its simplicity and equivalence to the sketch setting [10]. Let $S$ denote the set of one-dimensional samples. Let the distance measure between two sample $x$ and $y$ ($x, y \in S$) be the absolute difference $cost(x, y) = |x - y|$.

**Definition 1.** *The online clustering model maintains a set of $k$ points (called centers) denoted as $\mu$ to minimize the variance of values within each cluster: $\phi(S, \mu) = \sum_{x \in S} \min_{c \in \mu}(cost(x, c))^2$. The number $k$ of centers controls the number of clusters.*

To adapt to dynamic network flow distributions, we find an online approximate solution to the clustering model to adjust the cluster centers to up-to-date subflow records. We keep an online set of cluster centers based on *k-means++* [23] with a $\log k$-factor multiplicative approximation to the optimal cluster loss.

### *F. Sketching Applications*

The query function performs monitoring queries on sketches from the sketching function with network-flow keys from

the membership function. Typical queries include: (a) **Per-flow frequency and entropy query**. They track the traffic volume of each distinct flow, or count the flow bytes. To query the size distribution of each inserted flow, we iteratively obtain approximation results with identifiers of inserted flows, then we build a list of approximated flow sizes as the flow size distribution. Similarly, we derive the entropy metric as the frequency distribution of approximated flow sizes. (b) **Heavy hitters**. Jellyfish enables the heavy hitter of top-K flows ingesting the most traffic volumes. For a given heavy-hitter detection threshold, we directly list those buckets whose averaged counters exceeds the threshold, since approximated values of inserted flows are derived from the averaged counters of buckets. Next, we query the cuckoo filters to obtain the list of network flows that map to these buckets, as subflow keys can be queried over the cuckoo filter. Based on heavy hitters, we can also find flows spanning multiple windows that fluctuate beyond a predefined threshold, i.e., the heavy changes.

Jellyfish allows for a versatile robust monitoring application for distributed networked environments. Each Jellyfish instance continuously accepts new records from a wide range of network devices and end hosts across multiple measurement intervals as long as the storage of the cuckoo filter is satisfied in the memory. The continuous sketching results from the query applications enables long-term accurate network troubleshooting and anomaly detection.

### G. Theoretical Analysis

Having presented the Jellyfish method, we next bound the accuracy of Jellyfish towards the expectation of the ground-truth values.

**Theorem 2.** *Assume that the $l$-th key is mapped to the $j$-th bucket in the bucket array. Let $\tau$ denote the subflow threshold. Let $Query(X_l)$ denote the estimator of the ground-truth counters $X_l$ for the $l$-th network flow in $S$. Let $f_s(X_l)$ denote the cardinality of subflow records for network-flow $X_l$. Then for a given positive variable $w$, the Jellyfish estimator satisfies that*

$$Pr\left(|Query(X_l) - E\left[X_l\right]| \geq f_s(X_l)w\right) \leq \frac{\tau^2}{w^2} \qquad (4)$$

As shown in Theorem 2, the estimated counter just deviates the expectation of the ground-truth counter within a short interval, which is proportional to the number of subflow records of each network flow.

## IV. Implementation

We implement the Jellyfish sketch in modular blocks based on a publish/subscribe (Pub/Sub for short) framework. The ingestion function aggregates packets at line rates to subflow records [24], and evicts subflow record messages by predefined threshold $\tau$. The hash table reduces packet-processing delay by cache prefetching and batch processing.

We increase the processing parallelism for insertion and query on the data structure. We buffer the insertion and query requests with two concurrent ring buffer queues. Multiple threads pull a bulk of insertion and query requests from the corresponding buffer queue in batches.

We adjust the clustering model in a stream approach: (i) *Initialization*. We initialize the clustering model with a set of subflow samples. The initial samples are collected just once. (ii) *Incremental adjustment*. Next, we incrementally obtain subflow samples to be aware of the variations of subflow distributions. We keep a sample cache to maintain up to $|S|$ timestamp record samples. To bound the storage cost, we attach each sample with a timestamp of the first appearance, and remove the earliest timestamp record counters. We incrementally adapt the clustering model with respect to samples: The first cluster center is sampled uniformly at random from the sample cache, and we sample up to $k$ cluster centers with replacement for each new subflow sample $p$ with probability $\frac{cost(\{p\},\mu)}{\sum_{q\in S} cost(\{q\},\mu)}$. The default number of cluster centers is set to 30 based on empirical performance. Thus the number of cluster centers is modest with respect to the scale of subflow records. The number of clustered subflow samples are set to a constant number in the same cache. We vary the choice of samples from 0.1 to one times the number of subflow records in the measurement interval, and the clustering performance is stable with respect to the numbers of samples in the cache.

## V. Evaluation

Having presented the Jellyfish method, we next report experiment results compared to state-of-the-art methods with real-world data sets. We report key results and put additional experimental results in the appendix.

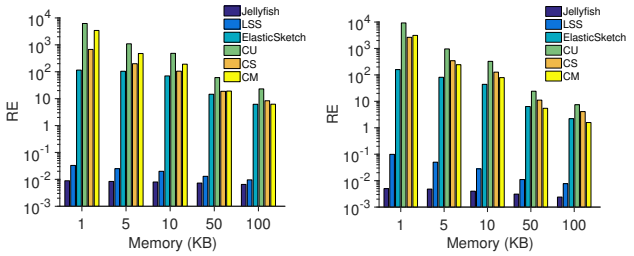
### A. Experimental Setup

**Data sets**: We perform a real-world trace-driven experiment study with two popular data sets: (i) *CAIDA*: it is collected on February 18, 2016 at the Equinix-Chicago monitor by CAIDA [8], with 1799.7 million stream network flows lasting for one hour. (ii) *MAWI*: it is collected on May 20, 2019 at the transit link of WIDE to the upstream ISP [25], with 14.0 million stream network flows lasting for 899.99 seconds. The source IP of each stream network flow serves as the network flow's key.

**Workflow**: We split each data set to ten intervals of equal size. Each interval is replayed to the ingestion function. The ingestion function publishes subflow tuples to sketching components over the Pub/Sub framework. This sketch is queried by the operator after the interval ends.

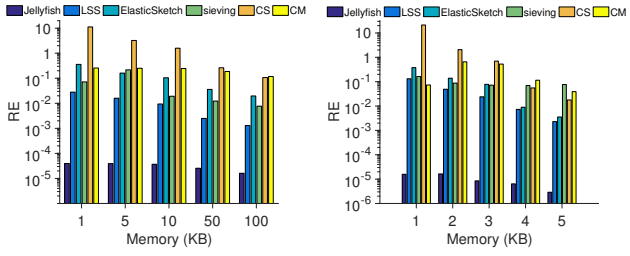
**Metrics**: We evaluate the effectiveness of the sketch methods with three metrics: (i) *Relative error* (RE): We use the relative error to quantify the accuracy of the per-flow query. It is defined as the mean of the relative error of each queried network flow. (ii) *F1 score*: We use the F1 score to quantify the precision for the heavy-hitter query. It is defined as the harmonic mean of the precision and the recall values, i.e., $\frac{2\mathrm{PR}\times\mathrm{RR}}{\mathrm{PR}+\mathrm{RR}}$, where $PR$ (Precision Rate) denotes the percent of true heavy-hitter instances reported, and $RR$ (Recall Rate) denotes the percent of found true heavy-hitter instances.

**Parameters**: We select the default parameters for Jellyfish based on the sensitivity experiments. We set the default



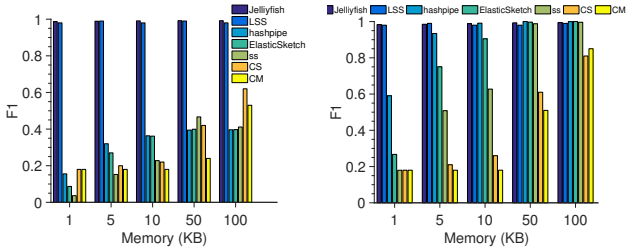

(a) CAIDA (b) MAWI

Fig. 5. The average of the relative errors for the per-flow query. The y-axis is plotted in the logarithmic scale.




(a) CAIDA (b) MAWI

Fig. 6. The average of the relative errors for the entropy query. The y-axis is plotted in the logarithmic scale.




(a) CAIDA (b) MAWI

Fig. 7. The average of the F1 score of compared methods.

number of clusters to 30, and the default number of buckets in Jellyfish to 0.1 times the number of network flows in an evaluation interval. We set the truncated threshold for subflow tuples to 128.

The experiments are repeated in ten times, and we report the average result and the 95-th confidence interval.

### B. Comparison

We first compare Jellyfish with state-of-the-art sketching methods using the same space for fair comparison. We report average metrics as a function of the total amount of memory of the sketch physical structure. We keep the membership of inserted keys with the same set of Cuckoo filters, since all sketching methods are agnostic of the keys by themselves.

**Per-flow Query**: We first test the relative errors of the per-flow query. We compare Jellyfish with five per-flow sketching methods including count-sketch (CS) [12], cusketch (CU) [3], count-min (CM) [11], Elastic Sketch (ES) [8] and LSS [10]. Figure 5 shows the relative errors as we vary the bucket storage from one KB to 100 KB. We see that the relative errors of Jellyfish and LSS are three to five orders of magnitude smaller than those of CS, CM, CU and ElasticSketch. Further, Jellyfish is significantly more accurate than LSS, since the record distributions in Jellyfish are less skewed than those in LSS.

**Entropy Query**: We next compare the relative errors of the entropy query of Jellyfish with five entropy-sketching methods including count-min (CM) [11], count-sketch (CS) [12], Sieving [26], Elastic Sketch (ES) [8] and LSS [10]. Figure 6 shows the relative errors as we vary the sketching storage from one KB to 100 KB. We see that Jellyfish is three to six orders of magnitude more accurate than the other methods. This is due to the fact that Jellyfish clusters less-skewed subflow records to bucket arrays, which closely preserves the global distribution of the network-flow counters.

**Heavy Hitter**: We next compare the F1 scores of the heavy-hitter query with six heavy-hitter methods including count-min (CM), count-sketch (CS), Spacesaving (SS) [27], ElasticSketch (ES), and hashpipe [4] and LSS [10]. We set the threshold of the heavy hitters to the top-5% of network flows. Figure 7 shows the F1 scores as we increase the sketching storage from one KB to 100 KB. We see that the F1 scores of both Jellyfish and LSS are close to one, since the heavy hitters depend on a small set of the largest network flows, and both methods put large items to the same bucket arrays by the clustering process. While hashpipe, ElasticSketch, SS, CS and CM are more sensitive to the mixing of small items with large items.

### C. Message Latency

We set up the experiments on ten servers in two racks connected by a 10 Gbps switch, each server is configured as 8-core Intel(R) Xeon(R) CPU E5-1620, 47 GB memory, and Intel 10-Gigabit X540-AT2 network card. We choose the Pulsar messaging system originally created at Yahoo [28] as the Pub/Sub underlay. We set up the Apache Pulsar 2.2.0 Pub/Sub as a standalone service on a dedicated server. We split nine servers to two groups: (i) Six servers run the network ingestion component to produce subflow records from replayed network trace and publishes to the Pub/Sub framework; (ii) Three servers run the sketching component to maintain the sketch data structure by subscribing to each of six ingestion servers. The sketching component subscribes to the events published by the ingestion component and feeds them to the Jellyfish within the same measurement intervals. We ingest port-mirrored traffic from the switch and feed them to independent sketching instances in parallel.

We evaluate the latency of delivering messages over the Pub/Sub message bus. Figure 8 plots the CDFs of the median, 90-th and 99-th percentiles of the messaging latency distributions in each measurement interval. We see that most messages are delivered within 60 ms, thus the Pulsar message bus provides timely delivery in most time. A small number of messages may be delivered for more than 100 ms due to the queueing in the message bus.

## VI. Discussion and Conclusion

*Discussion*: The sketch is equivalent to a clustering process that needs complex algorithmic tradeoffs between ac-

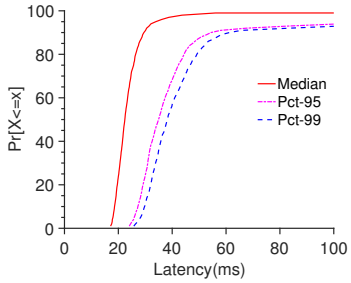



Fig. 8. CDF of delivering latency in the message bus.

curacy, speed and resource budgets. The need of coping with line-rate packets increases the resource contentions of the sketch structure with colocated deployed systems. Many sketching oriented measurement systems are moving towards programmable network devices, e.g., P4, FPGA, SmartNIC and software switches, which embed the sketch data structure into the programmable packet processing pipelines. As more and more packets must be inspected, the sketch's space and time complexity must be tuned comprehensively to trade off between the accuracy, speed and query types. However, the forward pipelines of programmable switches are less coherent to the real-time feedback loop based operations. As a sketch only produces flow-level estimation results, Jellyfish feeds subflow-level record streams to the sketch data structures, which are kept at distributed resource-rich nodes that allow for flexible algorithmmic innovations for the sketching design.

*Conclusion and Future Work*: We present a locality-sensitive subflow sketching method Jellyfish based on subflow-clustering model. Real-world data sets based experiments show that Jellyfish significantly reduces the per-flow query errors by orders of magnitude with low variance within bucket arrays. The implementation is publicly available [1]. We plan to find more space-optimized data structures to track subflow membership.

## REFERENCES


[1] Y. Fu and X. Xu, "Self-stabilized distributed network distance prediction," *IEEE/ACM Trans. Netw.*, vol. 25, no. 1, pp. 451–464, 2017.

[2] Y. Fu, D. Li, P. Barlet-Ros, C. Huang, Z. Huang, S. Shen, and H. Su, "A skewness-aware matrix factorization approach for mesh-structured cloud services," *IEEE/ACM Trans. Netw.*, vol. 27, no. 4, pp. 1598–1611, 2019.

[3] C. Estan and G. Varghese, "New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice," *ACM Trans. Comput. Syst.*, vol. 21, no. 3, pp. 270–313, 2003.

[4] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, "Heavy-hitter detection entirely in the data plane," in *Proceedings of the Symposium on SDN Research, SOSR 2017, Santa Clara, CA, USA, April 3-4, 2017*, 2017, pp. 164–176.

[5] G. S. Manku and R. Motwani, "Approximate frequency counts over data streams," in *VLDB 2002, Proceedings of 28th International Conference on Very Large Data Bases, August 20-23, 2002, Hong Kong, China*, 2002, pp. 346–357.

[6] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One sketch to rule them all: Rethinking network flow monitoring with univmon," in *Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016*, 2016, pp. 101–114.

[7] G. Cormode, "Data sketching," *Commun. ACM*, vol. 60, no. 9, pp. 48–55, 2017.

[8] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, "Elastic sketch: adaptive and fast network-wide measurements," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2018, Budapest, Hungary, August 20-25, 2018*, 2018, pp. 561–575.

[9] Q. Huang, P. P. C. Lee, and Y. Bao, "Sketchlearn: relieving user burdens in approximate measurement with automated statistical inference," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2018, Budapest, Hungary, August 20-25, 2018*, 2018, pp. 576–590.

[10] Y. Fu, D. Li, S. Shen, Y. Zhang, and K. Chen, "Clustering-preserving network flow sketching," in *IEEE INFOCOM*, 2020.

[11] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *J. Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[12] M. Charikar, K. C. Chen, and M. Farach-Colton, "Finding frequent items in data streams," in *Automata, Languages and Programming, 29th International Colloquium, ICALP 2002, Malaga, Spain, July 8-13, 2002, Proceedings*, 2002, pp. 693–703.

[13] Y. Bengio, A. C. Courville, and P. Vincent, "Representation learning: A review and new perspectives," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 35, no. 8, pp. 1798–1828, 2013.

[14] Q. Huang, X. Jin, P. P. C. Lee, R. Li, L. Tang, Y. Chen, and G. Zhang, "Sketchvisor: Robust network measurement for software packet processing," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017, Los Angeles, CA, USA, August 21-25, 2017*, 2017, pp. 113–126.

[15] G. Cormode and M. Hadjieleftheriou, "Finding the frequent items in streams of data," *Commun. ACM*, vol. 52, no. 10, pp. 97–105, 2009.

[16] Z. Liu, R. Ben-Basat, G. Einziger, Y. Kassner, V. Braverman, R. Friedman, and V. Sekar, "Nitrosketch: robust and general sketch-based monitoring in software switches," in *Proceedings of the ACM Special Interest Group on Data Communication, SIGCOMM 2019, Beijing, China, August 19-23, 2019*, J. Wu and W. Hall, Eds. ACM, 2019, pp. 334–350.

[17] Q. Huang, H. Sun, P. P. C. Lee, W. Bai, F. Zhu, and Y. Bao, "Omnimon: Re-architecting network telemetry with resource efficiency and full accuracy," in *ACM SIGCOMM*, 2020.

[18] R. Jang, D. Min, S. Moon, D. Mohaisen, and D. Nyang, "Sketchflow: Per-flow systematic sampling using sketch saturation event," in *39th IEEE Conference on Computer Communications, INFOCOM 2020, Toronto, ON, Canada, July 6-9, 2020*. IEEE, 2020, pp. 1339–1348. [Online]. Available: https://doi.org/10.1109/INFOCOM41043.2020.9155252

[19] B. Fan, D. G. Andersen, M. Kaminsky, and M. Mitzenmacher, "Cuckoo filter: Practically better than bloom," in *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies, CoNEXT 2014, Sydney, Australia, December 2-5, 2014*, 2014, pp. 75–88.

[20] D. Zhou, B. Fan, H. Lim, M. Kaminsky, and D. G. Andersen, "Scalable, high performance ethernet forwarding with cuckooswitch," in *Conference on emerging Networking Experiments and Technologies, CoNEXT '13, Santa Barbara, CA, USA, December 9-12, 2013*, 2013, pp. 97–108.

[21] H. Dai, Y. Zhong, A. X. Liu, W. Wang, and M. Li, "Noisy bloom filters for multi-set membership testing," in *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science, Antibes Juan-Les-Pins, France, June 14-18, 2016*, 2016, pp. 139–151.

[22] S. Burckhardt, *Principles of Eventual Consistency*, principles of eventual consistency ed., ser. Foundations and Trends® in Programming Languages. Now Publishers, October 2014, vol. 1. [Online]. Available: https://www.microsoft.com/en-us/research/publication/principles-of-eventual-consistency/

[23] D. Arthur and S. Vassilvitskii, "k-means++: the advantages of careful seeding," in *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2007, New Orleans, Louisiana, USA, January 7-9, 2007*, 2007, pp. 1027–1035.

[24] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, "Trumpet: Timely and precise triggers in data centers," in *Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016*, 2016, pp. 129–143.

[25] "Packet traces from wide backbone," http://mawi.wide.ad.jp/mawi/, 2019.

[26] A. Lall, V. Sekar, M. Ogihara, J. J. Xu, and H. Zhang, "Data streaming algorithms for estimating entropy of network traffic," in *Proceedings of the Joint International Conference on Measurement and Modeling*


[1] https://github.com/yongquanf

*of Computer Systems, SIGMETRICS/Performance 2006, Saint Malo, France, June 26-30, 2006*, 2006, pp. 145–156.

[27] A. Metwally, D. Agrawal, and A. El Abbadi, "Efficient computation of frequent and top-k elements in data streams," in *Database Theory - ICDT 2005, 10th International Conference, Edinburgh, UK, January 5-7, 2005, Proceedings*, 2005, pp. 398–412.

[28] "Apache pulsar framework," http://pulsar.apache.org, 2018.

**Yongquan Fu** received the Ph.D. degree in computer science from National University of Defense Technology, China, in 2012. He is an associate professor at National University of Defense Technology, China. His current research interests lie in network computing, network measurement, and distributed systems.

**Lun An** is a Ph.D. candidate at Beijing University of Posts and Telecommunications, Beijing. His current research interests lie in cybersecurity.

**Siqi Shen** received the B.S. and M.S. degrees from the National University of Defense Technology, China, in 2007 and 2009, respectively, and the Ph.D. degree from the Delft University of Technology, The Netherlands, in 2015. He is currently an Assistant Professor with the Xiamen University. His research interests include Network Measurement, Reinforcement learning, and 3D vision.

**Kai Chen** received the Ph.D. degree in Computer Science from Northwestern University, Evanston, IL, USA, in 2012. He is currently a Professor with the Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Hong Kong. His research interests include data center networking, high-performance networking, machine learning systems, and hardware acceleration.

**Pere Barlet-Ros** is a Full Professor with the Computer Architecture Department of the Universitat Politècnica de Catalunya (UPC), Barcelona, Spain, and Scientific Director at the Barcelona Neural Networking Center (BNN-UPC). He received the M.Sc. and Ph.D. degrees in Computer Science from UPC, in 2003 and 2008, respectively. From 2013 to 2018, he was Co-founder and Chairman of the machine learning startup Talaia Networks. His research has been integrated in several open-source and commercial products, including Talaia Polygraph, Auvik TrafficInsights, Intel CoMo and SMARTxAC. In 2015, he was awarded as the best entrepreneur of the UPC School of Informatics (FIB). More recently he has received the ICREA Academia award (2023-2027).