Politecnico di Torino

ScuDo
Scuola di Dottorato ~ Doctoral School
WHAT YOU ARE, TAKES YOU FAR

Doctoral Dissertation
Doctoral Program in Electrical, Electronics and Communications Enginering (36th cycle)

Sketch-based In-Network Monitoring of Data Centers and Programmable Networks

Alessandro Cornacchia
* * * * * *

Supervisors
Prof. Paolo Giaccone, Supervisor
Prof. Andrea Bianco, Co-supervisor

Doctoral examination committee
Prof. Gianni Antichi, Politecnico di Milano – *Referee*
Prof. Franco Callegati, Università di Bologna
Prof. Claudio Casetti, Politecnico di Torino
Prof. Stefano Salsano, Università degli Studi di Roma Tor Vergata – *Referee*
Dr. Luca Vassio, Politecnico di Torino

Politecnico di Torino
18th April, 2024

I hereby declare that, the contents and organisation of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

Alessandro Cornacchia

.....

Turin, 4th April, 2024

Summary

The reliability and performance of data centers play an essential role for the success of plenty of today's business and entertainment activities, while heavily depending on the effectiveness of the employed monitoring solutions. Data center operators need the ability to accurately monitor and dissect the behavior of their networks, while the tenants of distributed cloud-native applications also strive for pervasive monitoring intelligence to understand application performance.

Unfortunately, the rapid increase of networks scales and link speeds, and the radical transition from monolithic applications to microservice architectures, have significantly raised the expectations on both network and applications monitoring, as well as their complexity. With ultra-fast link speeds, network telemetry reports can contain millions of monitored events even on short timescales. Since a high monitoring timeliness and accuracy is often translated into an increase in the frequency and volume of telemetry reports, network monitoring can come at a huge overhead. At the same time, the multi-platform multi-layer distributed microservice design has increased the exposure of applications to failures, which also resulted into the explosion of the volume of application-level telemetry, ultimately leading to a significant data bloat issue for *cloud-native application observability*. Overall, monitoring at hyperscale is a problem still far from being solved.

In this thesis, a major effort has been devoted towards devising in-network solutions with the goal of increasing the monitoring performance for both the network and cloud-native applications, while keeping overheads and costs disposable. Towards this objective, we developed a set of sketch-based algorithms and systems, which helps to extract insightful statistics from high-speed data streams directly from programmable network devices, such as Protocol Independent Switch Architecture (PISA) switches and SmartNICs.

We first addressed the task of measuring the flow cardinality in a traffic stream, which is a common input to many fundamental network management tasks, ranging from attack detection to network planning. Specifically, we focused on turning some popular cardinality estimation sketches (e.g., HyperLogLog) into continuous-time sketches, i.e., add the ability to answer queries at arbitrarily time instants according to a sliding window model. Our proposed schemes allow overcoming the insensitivity to data recency of existing sketches, and enable better timeliness with minimal

interactions with a remote monitoring plane, due to the ability to continually compute real-time statistics directly in the network switches.

Despite their memory-efficiency, sketches are practically limited by the amount of SRAM memory available on a single switch, which is typically scarce for commodity hardware. Therefore, a second contribution relates to the collaboration of multiple switches to increase the accuracy of flow size estimation sketches. More specifically, we looked into the concept of disaggregated sketches, in which fragments of a logically single sketch are distributed across the switches. We focused on characterizing the interdependencies between the traffic patterns and the distribution of the measurement workload across the fragments along the flow's network path. We showed that the estimation accuracy can be significantly improved only by carefully choosing a subset of fragments to update.

In our final contribution, we complemented our sketch-based network monitoring suite with a new sketch-based framework for cloud-native applications observability. We tackled the observability data bloat problem by proposing a novel three-tier architecture to monitor cloud-native applications, which leverages the proximity of SmartNICs to the applications' microservices to mitigate the high overheads of observability. Our system stands out from the conventional observability tools by incorporating local metrics processing stages at every server within a sketch-based lightweight data plane running on SmartNICs. To the best of our knowledge, it is the first attempt to accelerate observability processing tasks through the offloading to a SmartNIC. As demonstrated on a production-grade Kubernetes cluster, our framework can help operators narrowing the focus only on informative data, and can proactively trigger actionable signals that anticipate Service-Level Agreements (SLAs) violations.

Altogether, we built a comprehensive suite of in-network monitoring tools and sketches to troubleshoot data center performance end-to-end.

Acknowledgements

I would like to thank all the referees for their valuable feedback, and the whole examination committee for the time dedicated to the discussion of my work.

This thesis finalizes a long journey full of challenges, emotions and successes, but also failures which demanded patience and trust. I am deeply thankful to my two advisors Prof. Paolo Giaccone and Prof. Andrea Bianco, who closely accompanied me throughout this journey. Their guidance and support have been technically excellent, and, more importantly, have always been conducted in a warm and welcoming style. I believe their generosity, openness and availability are hard to match, and are some fundamental (and quite unique) traits any student would aspire to find in its advisor. I am very grateful for the trust they have placed in me, especially in the most critical and tough moments. They contributed to my research and personal growth by providing me with invaluable tools to face the challenges of the academic world. Last but not least, sharing common hobbies and leisure activities was a real blessing, and I will cherish for good the experiences we enjoyed together.

I would also like to express my gratitude to all the colleagues and academic peers I had the opportunity to collaborate with. First and foremost, thanks to Politecnico di Torino and the TNG Group members, who consistently make possible for their PhD students to approach top-tier venues and connect with worldwide leading experts of the field. Big kudos to Prof. Marco Canini and its amazing research group at King Abdullah University of Science and Technology (KAUST), for how they set up my long stay there. While having the chance to work on very exiting projects and cutting-edge technologies, I also felt very welcomed and integrated in the group. A special thank to Prof. Giuseppe Bianchi for the fruitful collaboration we had and the brilliant ideas he shared with me, which inspired a big part of my research. Finally, a distinctive token of appreciation to Franco Galante, who has been a super inspiring office mate all this time long, and a great friend who offered relief at the time I needed. I will miss our constructive discussions and will not forget the beautiful days spent together.

Thanks to my family, who unceasingly supported me throughout my studies, and to the community of Moena, its majestic mountains and peaceful woods, which eventually played a key role to help me mature a better self-awareness and balance.

Part of the computational resources for this work were provided by HPC@POLITO (http://www.hpc.polito.it).

To my family.

Contents

Chapter 1

Introduction

Society is experiencing an enormous digital revolution. Online services and applications are pervasively employed in all aspects of society, changing the way we live, work and interact each other. Communication networks represent the key enablers of this revolution. Since the rise of the World Wide Web in the past century, communication networks have evolved according to a decentralized ownership model such as the Internet. In the last decades, the advent of clouds and private cloud networks has partially reshaped this model, by massively concentrating heterogeneous services operated by both enterprises and private users to big interconnected Data Centers (DCs), owned by few hyperscalers. As such, data centers have gained pivotal importance in the modern digital economy. This trend is not expected to change in the near future, as the explosion of artificial intelligence and machine learning (AI/ML) applications demands unprecedented computational power and storage capacity for training models with trillions of parameters.

Consequently, data centers must guarantee uninterrupted availability and high performance. Towards this goal, monitoring the performance of services running in the data center, by means of collecting and processing *data center monitoring*, is a fundamental task to ensure the quality of operations and respond to failures or performance degradation. Data center monitoring should identify and localize performance issues in real time, thus increasing customer satisfaction and minimizing revenue loss. However, the scale and the complexity of modern data centers pose several non-trivial engineering challenges to data center monitoring.

From a network perspective, the massive amount of traffic flowing through the network makes it difficult to quickly identify and diagnose network performance issues with traditional tools. With the latest standards (e.g., 400GbE) reaching link speeds above 100Gbps, data center networks can carry hundreds of terabit-per-second traffic with billions of flows, thus implying significant overheads to monitor network connections. In addition, not responding timely to network issues may have larger impact than in the past. Because of the higher link speeds, negative effects can affect a larger amount of user traffic before the remedies are brought into effect.

Similarly, from an application-layer perspective, monitoring applications is also very daunting. Modern applications involve distributed interacting components (e.g., microservices, serverless) and several software abstractions running on top of the operating system (e.g., orchestrators). Given the distributed nature and the associated software complexity, it is not clear which performance metrics reflect the events of interest and where they should be collected. In general, capturing the metrics relevant to debug incidents or analyze performance typically result in huge overheads.

To tackle the above-mentioned challenges, this dissertation will focus on the design and implementation of novel algorithms and systems for data center monitoring. We embrace the ability of new programmable network devices, specifically programmable data plane switches and programmable Network Interface Cards (NICs), to host monitoring functions that were previously executed on dedicated servers, and achieve higher coverage of relevant system events, better accuracy and lower costs.

The remainder of this introductory chapter addresses the following preliminary objectives. First, we provide a brief overview of the evolution of data center networks and emphasize the latest advances towards end-to-end programmable Data Center Networks (DCNs). Second, we revise the role of monitoring in modern data centers and discuss existing solutions through the lens of programmable networks, which motivates the relevance of this thesis. Finally, we outline the structure of the dissertation and summarize its new contributions to the state-of-the-art.

1.1 Programmable data planes

In 2008, McKeown et al. published the cornerstone paper "OpenFlow: enabling innovation in campus networks" [1], which introduced the concept of Software Defined Networking (SDN). SDN decoupled the control plane from the data plane, and moved the control plane into a logically centralized controller. This separation of tasks allowed network operators to manage the network in a more flexible and programmable way: control plane algorithms could have a global network view at the central controller, and enforce their policies to the network by communicating with the data plane via a standard interface like the OpenFlow protocol. As soon as an ever-growing number of SDN applications started to emerge, some missing features of "traditional" SDN became evident. The OpenFlow protocol was designed to be a simple protocol that allows the controller to programmatically configure the forwarding rules of the switches. However, the set of supported rules and protocols was still limited by fixed-function switches and aligned to the OpenFlow standard, which limited the ability of network programmers to quickly roll out new protocols and features.

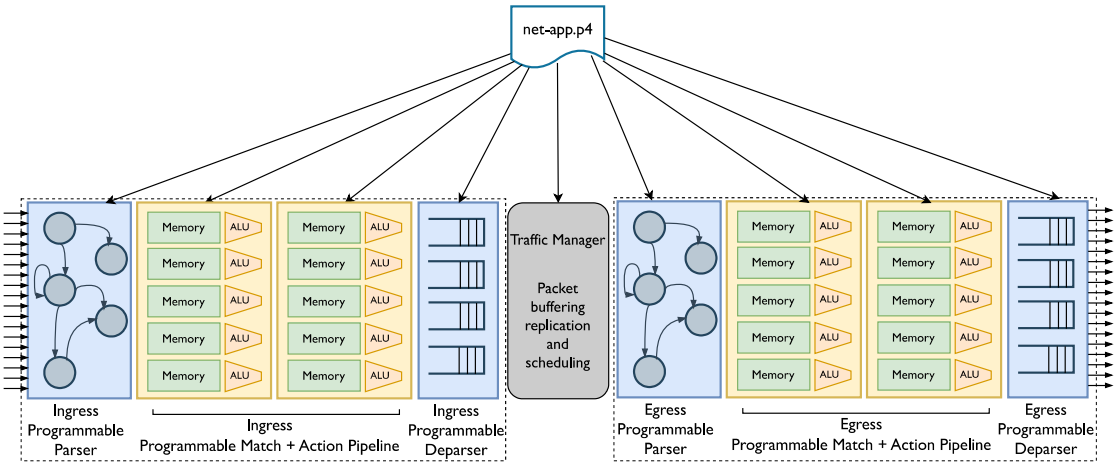


Figure 1.1: A P4 architecture known as Tofino Native Architecture (TNA) [2]. It includes ingress and egress stages (dashed boxes), both according to the PISA reference model. The P4 program `net-app.p4` defines the behavior of each component in the P4 architecture (long arrows) when processing packets, which logically traverse the architecture from left to right (short arrows).

Protocol Independent Switch Architecture (Protocol Independent Switch Architecture (PISA)) and P4

This limitation was a major bottleneck for the network, which struggled to evolve with the same rapid pace of services and applications. Thus, it fostered the creation of a *language-based* approach to program the network data plane. Over the last decade, the PISA [3] abstraction was introduced to provide network programmers with a device-independent reference model to write custom programs to be run in the network data plane. PISA is as a reference logical machine to represent the pipeline stages involved into packet processing in the network data plane. Figure 1.1 depicts a high-level overview of the PISA model, which includes three major components. The programmable parser permits to specify the structure of the header to be extracted from the packets, and where in the bit stream these headers are located. Then, a sequence of match-action units constitute the core programmable pipeline. Each stage of the pipeline can be programmed to match the header fields identified at the parser and intermediate results computed by previous stages. For the matched headers, the programmer can specify the actions to execute. A single match-action stage has multiple memory blocks (e.g., tables, registers) and Arithmetic Logic Units (ALUs), which the data plane program can access to implement custom (stateful) logic. Finally, the deparser is the block that re-serializes the packet metadata into the packet before it is transmitted on the output link. Notably, the deparser can add new headers to the packet, computed by the match-action pipeline, which enables new protocols to be added to the network without changing the hardware.

Essentially, programming a switch data plane consists in defining the desired behavior of every component in a PISA-like architecture. Currently, P4 [4] is the most popular high-level language to program PISA devices. The P4 program, which describes the PISA logic, is then compiled into low-level platform-specific directives and installed in the data plane of a *target* device. Different targets feature different P4 compilers, which are distributed by manufacturers along with the architecture-specific libraries. Several platforms supporting P4 are emerging, including software switches (the bmv2 Simple Switch [5], T4P4S [6] and DPDK [7]), ASICs (Intel Tofino Series [8]), FPGA boards (NetFPGA [9]) and SmartNICs[1] (Pensando DSC [10]). Every platform exposes PISA-like P4 architecture, which associates the P4 language with the specific P4 target. Many P4 architectures, such as Tofino Native Architecture (TNA) [2] (depicted in Figure 1.1), replicate the reference PISA model both at the ingress and the egress, as highlighted by the dashed boxes.

P4 triggered a second generation of SDN which enabled a top-down approach to develop network applications and quickly add new features without having to wait for long release cycles because of the interaction between vendors, operators and institutions. Importantly, data plane programmability opened the door to a completely new generation of monitoring tools based on P4. This is still a very active research area, whose efforts are mainly devoted to solve the challenges of implementing complex monitoring logic in P4 switches.

Programming constraints of PISA switches. Programming a P4 switch implies dealing with several implementation constraints, which arise from the need to keep up with terabit per second line rates. Without delving into the specifics, we outline the main constraints using the following two categories.

- *Language constraints.* P4 targets do not support loops neither floating point arithmetic. To iterate over a set of elements, the programmer must unroll the loop and manually replicate the logic for each element. Similarly, floating point arithmetic is not supported, and the programmer must use fixed-point arithmetic, which ultimately leads to the creation of original approaches to relax the dependence from floating-point arithmetic [11], [12].
- *Memory access constraints.* Due to strict timing requirements, when a packet arrives, it can access only a few addresses in the per-stage memory, but not read or write the entire memory block. In addition, each stateful memory block can be accessed only from a single stage of the pipeline, therefore a packet can access the same memory block only once as it moves through the pipeline.

We will see in the next section that these constraints have actually turned into an opportunity and stimulated the originality of numerous researchers to develop new (monitoring) algorithms and data structures tailored to P4 switches.

[1] This family of programmable network equipment may also be referred to as Infrastructure Processing Unit (IPU) or Data Processing Unit (DPU), depending on the hardware manufacturer.

1.2 Data center monitoring

Before describing the structure of the dissertation, we briefly characterize the data center environment and its workloads, which will motivate the increasing expectations on the performance of the data center monitoring systems. Then we define the terminology of the thesis and outline its research context.

Navigating the data center's ecosystem

Data centers host a wide spectrum of services and applications, which include social networks, e-commerce, video streaming, AI/ML model training and inference, virtual conferencing, online gaming, and more. This workload is increasingly diverse, and evolves frequently driven by an always expanding user base and its requirements. In order to match the scale, heterogeneity and velocity of this evolution, both the interconnection network and the applications' software have being designed with the *divide et impera* principle in mind, today resulting in large-scale *modular* systems. Data center networks can comprise thousands of switches and several thousands of links, which are continuously expanded "horizontally" by deploying additional equipment, rather than upgrading existing instances to higher grade devices [13]. Similarly, the applications' software infrastructure has largely transitioned to a distributed design, where several interacting components, such as microservices, serverless functions, etc. run on top of several layers of abstractions including orchestrators, virtual machines and containers.

Unfortunately, while modularity promotes flexibility for system upgrades and lowers down the costs, it also increases dramatically the likelihood of wrong configurations, software bugs, hardware failures and hotspots. This is especially critical for today's applications for at least two reasons.

First, because of the distributed design, differently from the past, applications heavily hinge on network communication to fulfill their logic. As a result, application performance is tightly coupled to network performance. Criticalities in the network, e.g., bandwidth declines, microbursts or long-tailed latencies, translate into severe slowdowns for the application more easily than in the past. This is especially critical today, with ultra-high link speeds (e.g., 100-400Gbps) and a rapid development of fast network stacks at the end-hosts (e.g., RDMA, DPDK, etc.), because the impact of even small incidents in the network is amplified. For a data transfer on a 400Gbps link, some tens of μs of variation in the network latency may be negligible for a traditional TCP connection, for which the processing time at end host is the main limiting factor. However, they have big incidence on a Remote Direct Memory Access (RDMA) connection, which expects a very low latency from the network since it bypasses the host's CPU.

Second, even not considering problems in the network, failures or anomalies happening in one of the interacting software components may quickly propagate to other

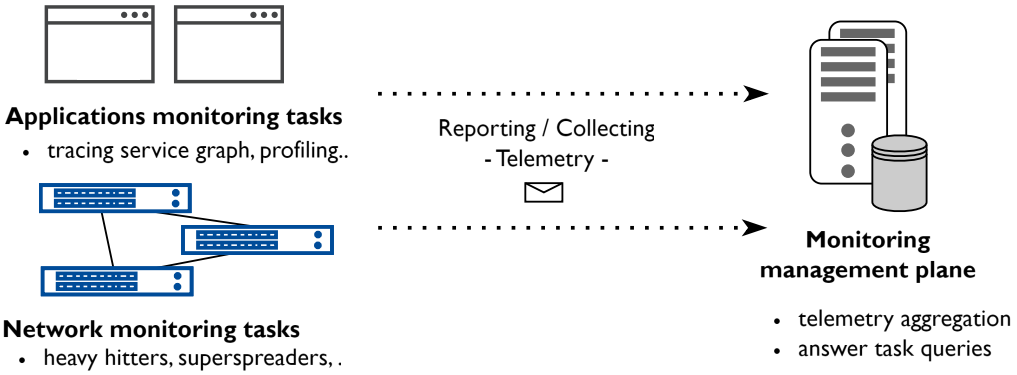


Figure 1.2: The notation used in the thesis about monitoring systems.

components, thus causing cascaded application errors or slowdowns. Some of these failures — called *gray failures* — are hard to detect, because they do not result in components crash, yet have a significant impact on the performance of the application. As a simple example, consider a backend service (e.g., key-value store) which processes slowly the requests coming from upstream fronted services (e.g., due to memory leak). In this case, the slow downstream service may progressively cause large queue buildups in the upstream frontend services, which in turn may become slow. Thus, an issue of a single software component may slow down the entire application, without any visible symptom in the root-cause component.

Within this ecosystem, a fundamental prerequisite to guarantee high availability and performance is to continuously *monitor* the data center across several layers and components. The holistic dimension is what significantly complicates monitoring compared to the past. Traditional monitoring tools are specialized, with focus on a very circumscribed context and producing only component-specific local information. To pinpoint the cause of a problem in today's distributed scenario, local information is not sufficient and monitoring must integrate data from multiple context and tools.

1.2.1 Terminology, roles and objectives

Figure 1.2 illustrates the organization of a typical monitoring system and clarifies the difference between monitoring and telemetry. *Monitoring* can be intended as the wide process of analyzing data center health and performance in order to ensure correct operations. We focus on monitoring of the interconnection network and the distributed applications running on top of it as primary targets. Monitoring is composed of a set of specific *monitoring tasks*, such as heavy hitter and superspreader detection for network monitoring, or tracing the interactions between component in a distributed application. Monitoring is achieved by aggregating and processing a set

of measurements obtained from the monitor targets, usually on a dedicated management plane (e.g., SDN controller). The output of the monitoring tasks is eventually useful for answering operator's queries or automatically triggering alerts if deviations from the desired behavior are detected. In parallel, *telemetry* refers to the procedures of reporting (or collecting) the measurements useful to monitoring — also referred to as *telemetry data*. Thus, telemetry is mainly concerned with the data collection itself. We will adopt this terminology throughout the thesis.

The quality of the monitoring can be understood according to the following four dimensions.

1) Accuracy. Quantifies how much the outputs of a monitoring task reflect the ground truth of the task's objective.
2) Timeliness. The speed (inverse of the delay) at which the monitoring output is made available for analysis and action, which is essential for quick detection and response.
3) Visibility (or *coverage*). The scope of the outputs of a monitoring task, in terms of covered system elements, e.g., per-flow, per-sampled flow, etc.
4) Overhead. Resource consumption and related costs required for implementing the monitoring system, i.e., telemetry and telemetry aggregation/processing.

Ideally, the monitoring should achieve high visibility, high accuracy, low overhead, and high timeliness, simultaneously. Unfortunately, the previous discussion already highlighted how achieving all goals simultaneously is challenging in the data center ecosystem, where it exists a fundamental tension between visibility and overhead. On the one hand, the large-scale modular design enforces to gain ever-more fine-grained visibility in order to get insights and explain complex system behaviors. For example, Google [14] and Alibaba [15], [16] have recently quantified for their production data centers how performance degradation occur in multiple points, including applications, kernel/TCP stack, virtual or physical networks, and are difficult to diagnose. On the other hand, the way the distributed modular design is conceived implies that collecting and processing telemetry from several sources makes difficult to sustain monitoring at disposable overheads. While a simple practical solution is to sample data to reduce the overhead, this may lead to a loss of accuracy. In this thesis, we will study techniques to achieve a lower telemetry overhead, and consequently monitoring overhead, without sacrificing accuracy or visibility.

Besides, the studies of Google and Alibaba have also disclosed how, even among hyperscalers, monitoring tools are rather compartmentalized and disconnected, with some tools looking into the *network* and others into *distributed applications*. Admittedly, this rigidity makes still an open issue understanding which segment of system causes problems that manifest to the application. Network monitoring tools have limited or zero visibility into end hosts, and vice versa. Historically, the monitoring systems for the network have matured independently of the monitoring systems for distributed applications, and have been devised by different communities. For this

Method type	Representative examples	Performance objectives			
		Accuracy	Visibility	Overhead	Timeliness
Sample & mirror (fixed-function)	SNMP stats [17] NetFlow [18], sFlow [19], EverFlow [20], cSamp [21]	low	low	medium	low
Packet telemetry	INT[22], PINT [23], IOAM [24], AM-PM [25], *Flow [26], NetSight [27]	high	high	high	medium
In-network offloading	NetSeer [16], PacketScope [28], Sonata [29], BurstScope [30], HashPipe [31], QPipe [32], ConQuest [33], BeauCoup [34], ElasticSketch [35]	medium	high	low	high

Table 1.1: The landscape of switch-based passive network monitoring methods.

reason, in the following we review the two separately to introduce the background useful to this dissertation.

1.2.2 In-network monitoring with programmable switches

Monitoring the network infrastructure and its traffic (e.g., identifying congestion hotspots, detecting lossy links, forecasting security threats) is indispensable to the successful management of hyper-scale data centers. The output of the monitoring process is used to dispose network management actions, including the optimization of traffic control schemes, capacity planning, users' bills accounting, among others. While network monitoring has been studied since the rise of communication networks, the advent of programmable network devices has brought new opportunities and challenges to the field, leading to a surge of research works in the field over the last few years. This thesis focuses on passive switch-based network monitoring, as its methods are directly impacted by the programmability of the data plane. In the following, we quickly review the evolution of passive network monitoring from fixed-functions switches to programmable switches, classifying existing solutions into three main categories.

Sample & mirror (fixed-functions) methods . Black-box switches only support a restricted set of fixed telemetry data, which consist in coarse-grained counters aggregated per-port, per-device or per-sampled-flow granularity, typically collected via protocols such as SNMP [17]. Another widely adopted tool is NetFlow [18], which was developed by Cisco and can collect per-flow statistics, e.g., packet counts, flow start and finish time. High-end routers (e.g., Cisco Catalyst series) implement NetFlow using dedicated silicon [36] to maintain the active set of flows and update data structures in hardware, while sustaining high data rates. However, these ASICs implementation sacrifice flexibility, by hard-coding monitoring functions. For example, NetFlow

summarizes traffic at the fixed granularity of the 5-tuple and does not allow expressing user-defined statistics. Besides, data center networks have traditionally opted for commodity switches equipped with merchant silicon and not expensive custom ASIC. For them, NetFlow can execute in software in the switch control plane. However, in order to limit the stress on the switch CPU and the interference with other control plane functions (e.g., routing), only a few packets are sampled and mirrored from the data plane to NetFlow in the control plane [37], thus resulting in poor accuracy.

To overcome the rigidity of NetFlow, the final solution with black-box switches was (selective) packet mirroring or traffic replays, which consists in sending a copy of (selected) packets headers to a remote collector for later processing. In this way, operators have the flexibility of extracting arbitrarily per flow statistics and perform advanced network-wide monitoring tasks. Packet to be mirrored can be selected by means of random sampling [21] or pre-configured matching rules on packet headers. For instance, EverFlow [20] samples every SYN packet and Cisco ERSPAN [38] protocol allows filtering by VLAN identifiers. However, also in this case the sampling rates are typically high in order to saturate the network with mirrored traffic, resulting in poor performance.

Packet telemetry methods. They obtain fine-grained telemetry on a per-packet basis. The main advantages of packet telemetry is that network devices can append arbitrary metadata to packet headers, which can provide per-packet per-hop visibility. These advantages are becoming more prominent as the data-plane becomes more configurable. For example, with In-band Network Telemetry (INT) [22], [39] programmable switches can query and report detailed information about their internal structures or stateful logic, e.g., queue occupancy. Thus, INT achieves both full-visibility of network state and a temporal resolution in the order of magnitude of microseconds. However, packet telemetry methods are faced with the high costs for reporting and aggregating telemetry data to the servers where the monitoring plane runs. INT can mirror packets at each switch (*postcard* mode) or mirror packets only at the sink switches (*passport* mode), but in both cases the traffic is at least doubled, which is a huge overhead for the network. Moreover, only for data aggregation, INT can saturate more than 100 production-grade server's CPU cores in a hyper-scale DCN [40]. Recent advancements, such as Direct Telemetry Access (DTA) [40], allow programmable switches to populate data structures residing in the servers' RAM without involving the servers' CPU. While DTA mitigates the telemetry aggregation overhead, INT still takes excessive network bandwidth for mirroring packets to the servers.

In-network offloading methods . The rise of programmable data plane switches has initiated a new generation of network monitoring tools and opened a space for entirely new research works in the field. The ability to execute custom logic in the data plane has put forward several opportunities for improvement over traditional methods. First, differently from fixed-function methods that either mechanically mirror sampled packets to the control plane or require expensive ASICs, modern switches

support custom measurements in the data plane. Because measurements can be taken on a per-packet granularity, programmable switches enable the computation of more accurate traffic statistics (e.g., flow counts) and with higher visibility. Second, (part of) the monitoring logic itself can be moved from a remote management (i.e., "monitoring") plane directly into the network, thus offloading the monitoring functionalities from dedicated servers to the switches. For example, researchers have recently proposed solutions to discover large flows entirely in P4 switches [31], and even more general frameworks that can partition the executions of Spark-like queries between the control and the data plane [29]. Embedding monitoring logic in the switches allows reporting ready-to-use results, as opposed to raw telemetry reports, saving bandwidth in the network and processing cycles at the remote servers. Third, programmable switches can take local decisions based on monitored traffic and enforce reactive measures without any interaction with the remote management plane (e.g., dropping, re-routing, scheduling, etc.), which increases the monitoring timeliness.

Remark

With data plane programming languages, e.g., P4, operators can embed the monitoring logic to *(i)* **measure**, *(ii)* **process** and *(iii)* **react** to network events directly in network devices. This paves the way to new monitoring tools with higher **accuracy**, lower **overheads** and better **timeliness**, as it reduces the dependencies on dedicated remote servers.

In this thesis we do not consider host-based methods, which leverage active probes and/or measurements at the host stack to infer network performance. Active probe methods [41], [42] inject additional probe packets into the network and combine measurements from multiple probes to infer (e.g., via network tomography algorithms) where issues (e.g., packet drops, latency increases, black-holes, etc.) happen in the network. However, they cannot deduce which user flows are impacted by failures, because probe traffic is independent of application traffic. Passive host methods monitor the socket parameters at the host stack to infer network performance [43]. However, they have limited visibility for network events happening on short time scales, providing milliseconds time resolution at best. Since these methods are less aligned with programmable switches, they are out of the scope of this thesis.

Sketch algorithms for in-network monitoring

The new capabilities of in-network offloading methods acted as a major driving force for the proliferation of P4-based monitoring tools. The main challenge for these tools consists in deriving sophisticated traffic statistics, while accommodating the implementation constraints of programmable switches (Sec. 1.1). As a consequence, the last decade has seen a surge of research proposals [32]–[35], [44]–[46] aimed at this objective, through the design of appropriate algorithms and data structures. In this

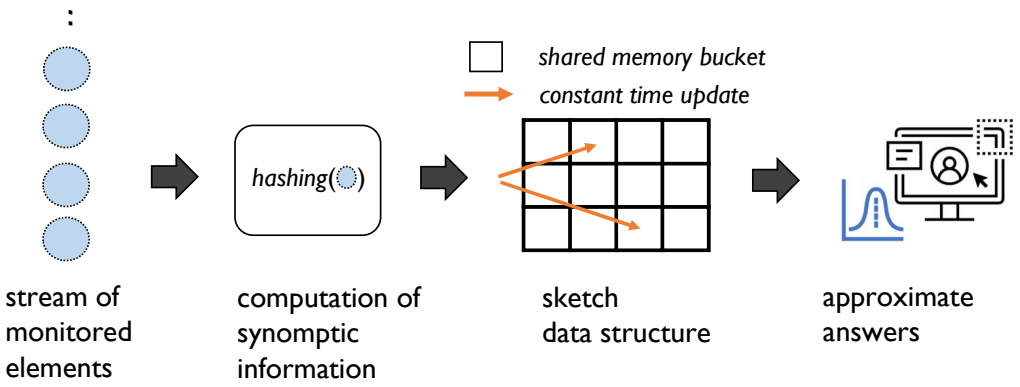


Figure 1.3: Generic representation of sketch algorithms for monitoring elements from a high-speed data streams, typically used in network monitoring tasks.

thesis, we also contribute to this research area to capitalize the opportunities of in-network monitoring offloadings.

Sketches are a family of fast and memory-compact data structures, thus they represent a popular choice for the above-mentioned objectives. In a nutshell, a sketch (summarized in Figure 1.3) is a mechanism that uses a constant space and constant update time data structure, while monitoring a large number of elements (e.g., network flows) coming from high-speed data streams (e.g., network traffic). To achieve its goal, the sketch aggregates in a compact memory space the statistics from multiple stream's elements. In other words, a pool of buckets (or counters) is multiplexed across different elements of the stream. A common way to multiplex memory buckets and keep the synoptic information is via *hashing* techniques. We will discuss explicit methods in the details in Chapters 2 and 3 for several sketches. Finally, in order to provide answers to per-element queries, the sketch typically leverages probabilistic techniques to recover per-element statistics from the "sketched" synoptic information available in the shared memory buckets. Therefore, the user will generally receive *approximate* answers, where the approximation error is due to the collisions between elements multiplexed to the same bucket, and can be bounded by the sketch's design.

Today's sketch-based tools for network monitoring, such as BurstScope [30], ElasticSketch [35], QPipe [32], cover most of the measurement tasks related to network monitoring, including heavy hitter detection, flow cardinality estimation, microburst detection, delay estimation, etc. While sketch structures have been studied for long in the past, the non-trivial implementations challenges of programmable data planes has stimulated new attention. Among others, it is worth mentioning that one challenge with sketches arise from their tight coupling with the statistic they measure: typically a sketch is specifically tailored to a single statistic (e.g., element frequency, stream cardinality). As a consequence, to support several measurement objectives simultaneously, many heterogeneous sketches should execute in parallel, which might

put pressure on the switch data plane. Notably, many of these problems are still unsolved and remain active research areas.

1.2.3 Cloud-native applications observability

In this section we introduce the context of cloud-native applications and the meaning of observability in this context.

Cloud computing, and more recently edge and fog computing, have gained traction owing to their ability to mask the provisioning and management of IT resources to business organizations and software developers. This paradigm shift has also apported significant changes to the way applications are designed, deployed, and evolved, with the microservice/serverless architecture becoming the de-facto standard for modern software development. The Cloud-Native Computing Foundation (CNCF) [47], as a vendor-neutral organization bringing together developers, users and vendors, and putting together a large ecosystem of open-source projects[2], is the most evident testament to the ambition of making these technologies ubiquitous. Among the various challenges acknowledged by the CNCF, *observability* is one of the most critical.

Evolution of distributed applications: from monoliths to cloud-native

With cloud computing, the IT infrastructure has become commodity to application developers. Physical hardware resources are virtualized and can be dynamically allocated at needs, introducing elasticity in the infrastructure provisioning. To fully take advantage of this agility, the software industry has transitioned from monolithic applications to distributed microservice-based applications. In a traditional monolithic application, software is developed and shipped as a single execution unit, which tightly integrates all its functionalities. While simple to deploy, this architecture is cumbersome to upgrade and scale. Any change to application's functionalities translates into the need of re-building and re-deploying the entire executable, introducing business interruptions and dependencies among different teams.

The adoption of microservice architectures, together with the DevOps design philosophy, had as a main goal to overcome this rigidity, which contrasted with the elasticity offered by the cloud. A microservice application is structured as a collection of multiple loosely coupled polyglot services, which are small application units independently deployable and scalable. To collectively realize application's functionalities, these services coherently communicate with each other through HTTP APIs, or the like. As opposed to monolithic applications, this architecture permits making

[2] counting up to about 180 in the early Dec 2023: https://landscape.cncf.io/.

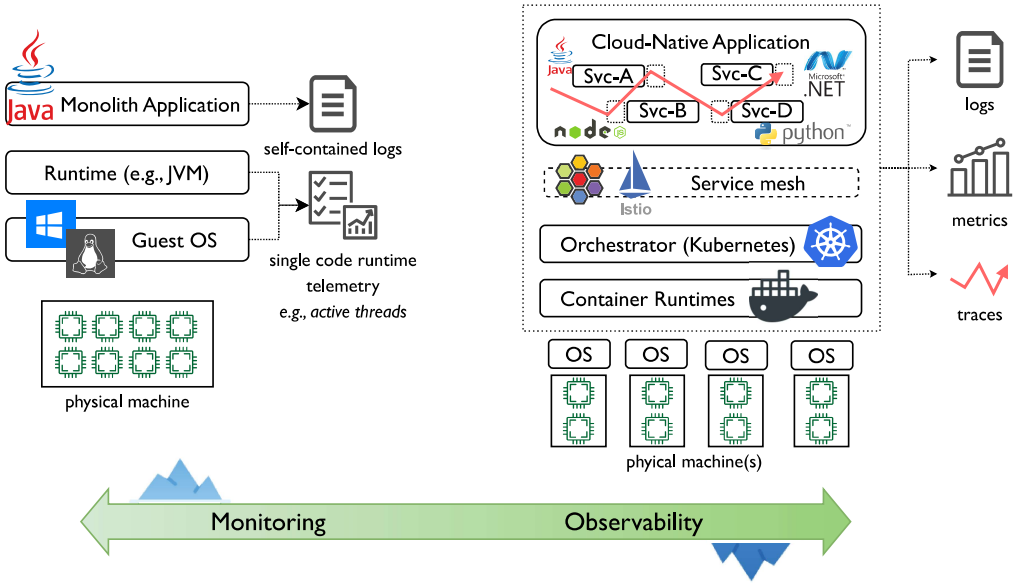


Figure 1.4: Comparison between the traditional monolithic application and the cloud-native scenario. The intensity of the green arrow quantifies the presence of latent (and unknown a priori) failure events. On the right, the three pillars of observability: metrics, traces and logs.

frequent changes with minimal effort, and, combined with infrastructure virtualization, to scale different services according to individual demands, thus also promoting cost efficiency. Likewise, it enabled a set of new practices for software development, known as DevOps. In DevOps, small teams of developers (*Dev*) can be organized around specific business capabilities and can concentrate only on the continuous agile development and testing of services, ignoring their actual installation and operation. In parallel, operations team (*Ops*) can focus on managing service runtime, including infrastructure provisioning and performance optimization.

All together, the modern distributed microservice-based application design and the set of DevOps practices are referred to as *cloud-native* technologies.

Observability in cloud-native applications

Cloud-native technologies turned building complex applications into an incredibly easy development task. At its extreme, developing a web application scalable to thousands of users is as simple as putting together ready-to-use components, such as a backend web server (e.g., nginx [48]) and a database service (e.g., MongoDB [49]), generating some frontend pages, and configuring orchestrators, such as Kubernetes (k8s) [50], to automatically rescale the number of service instances depending on the load volume.

However, the simplicity of development is counterbalanced by the complexity of

monitoring and troubleshooting performance. Aspects of monitoring that were routine, including debugging, profiling, and performance management, are now orders of magnitude more complex to realize. Figure 1.4 illustrates the difference between a traditional monolithic scenario, on the left, and a state-of-the-art cloud-native scenario, on the right. Monitoring a traditional monolithic application was, broadly speaking, a relatively static and deterministic problem. Performance bottlenecks could have been tested for, and intercepted, during development cycles, and the application was either in an up or down state at runtime. Operators were used to set up automated checks (e.g., via Nagios [51]) which could access a few handful self-contained logs data. Moreover, being monolithic, the application was deployed over a single runtime, like a Java Virtual Machine (JVM), which could be configured to provide rich monitoring information about code execution.

In contrast, in a microservice architecture, services are developed over a plethora of diverse runtimes and frameworks, and with extremely rapid development cycles. The operations teams have often limited understanding of applications code and, more importantly, they might not have access to how the services generate their logs. Service instances are volatile and continuously deployed and migrated across servers with potentially heterogeneous runtime conditions. Lastly, every application functionality is accomplished via the interaction of multiple services and the network, which makes it difficult to understand the root cause of performance glitches. Overall, the application state cannot anymore be explained by checking few pre-determined conditions, but requires a global birds-eye view of the system.

Monitoring and observability. Observability is a fresh concept that has recently matured as a remedy to this complexity. While observability and monitoring are, especially in the academic community, often erroneously used to refer to the same concept, they actually indicate to two different — but complementary — things. As explained in Sec. 1.2.1, monitoring is the process of accomplishing a set of pre-defined tasks to verify system's health and performance. It implies that we know in advance what we are looking for, and we collect a set of telemetry data finalized to this objective (e.g., check CPU utilization). On the other hand, observability — defined as "*the ability to infer the internal state of the system only by its external outputs*" [52] — has the more ambitious objective of understanding for any failure, potentially never seen before, where and why it occurred. In other words, while monitoring is concerned with verifying a set of pre-formulated hypothesis (i.e., proactive), observability is more concerned with dealing with unknown events, providing granular insights into the system's internals, as well as enough context to dissect complex unknown faults (i.e., reactive). The two concepts though are complementary: observability does not replace monitoring, and monitoring is a crucial prerequisite for observability, which can leverage several monitoring frameworks to check the telemetry data that we describe next.

Metrics, traces and logs. In its current implementations, observability hinges on a pervasive collection and analysis of three types of telemetry data, which are delivered

by practically all vendors [53]–[57] and are commonly referred to as the three observability pillars: metrics, logs and traces.

1) Metrics are numerical data that describes applications and system performance over time. They are collected from a variety of software layers, including orchestrators, sidecar proxies that handle communication between services [58], [59], and the services themselves, which can define custom metrics specific to their business logic. Examples of metrics include service CPU utilization, memory usage, network traffic, active and terminated connections, etc. They are typically collected at regular intervals and labeled with metadata, such as the timestamp, the service name, the instance ID, etc., allowing Ops teams to aggregate and filter them according to several dimensions. Metrics are typically displayed over live dashboards and are used to trigger alerts based on automated analysis tools [60]. Operators can eventually correlate metrics across the components of infrastructure to get a more comprehensive view of system health.

2) Traces. Metrics alone do not provide any information about the interactions between services occurring to realize the application functionalities. Traces are the detailed recordings of the user requests that have committed work to do to the application. A trace is a collection of events that describe the execution of a single user request across chains of multiple services. Thus, a trace describes the causal dependencies among visited components and the end-to-end structural flow of request executions (e.g., in the form of a directed acyclic graph). Traces were pioneered by Google with its Dapper tracing infrastructure [61] and have become lifesavers to identify the services' contribution to the end-to-end request latency, and to detect the root cause of performance issues. Some popular open source tools are Jeager [62] and Zipkin [63], which follow the model pioneered by Google's Dapper.

3) Logs are typically unstructured textual data (or semi-structured) that are generated when specific part of codes execute. Logs are the most detailed but voluminous source of information, and are now re-gaining traction thanks to the advancements of natural language processing capabilities (e.g., Generative Pre-trained Transformer models, abbreviated as GPT). In the distributed context of cloud-native, contrarily to monolithic applications, logs are no longer self-contained files containing consequential information generated by a single executable, but they are generated by multiple services, runtimes and frameworks. This factor is what makes difficult to correlate logs telemetry across the application.

A synthesizing example. Overall, observability combines data from metrics, logs and traces to dissect the internals of a cloud-native application and go beyond fixed monitoring tasks. As an illustrative example, consider a service A that tries to access a key-value store used as an in-memory cache (e.g., Redis [64]) and falls back to a (slower) persistent storage if the desired content is not found in the cache layer. Suppose that service A starts making use of the cache service after a recent service update, and that the cache memory limit is now misconfigured as it does not account for the cache accesses of A. As a result, more cache entries would be evicted, thus causing A's request

to fall back to the persistent storage and increase pressure to it. By looking at metrics, an Ops teams would know about an increase in cache evictions and a corresponding increase of the number of HTTP requests to the persistent storage. However, it could not understand the root cause, i.e., service *A* also using the cache. The traffic generated by *A* could also fall under the radars of network monitoring tools — e.g., heavy hitter detection — without triggering any alarm. However, thanks to traces, the Ops team would know that service *A* started using the cache service and how intensively. In addition, the application logs of service *A* could reveal why the service started using the cache and take informed decision to restore performance.

In this thesis, we focus on improving the monitoring aspects of observability and the quality of collected telemetry data, leveraging programmable NICs at the hosts. As we will anticipate in the next section (and address in the details in Chapter 4), we propose NIC-local sketch algorithms to monitor and filter relevant metrics. This explains the reason why we adopt the terminology *monitoring* for the thesis contribution[3], even in the context of cloud-native applications.

1.3 Structure and contributions of the thesis

In this dissertation we focus on the design and implementation of novel sketch-based algorithms for single-switch network monitoring (Chapter 2), cooperative cross-switch network monitoring (Chapter 3) and cloud-native application observability (Chapter 4). Overall, the novelties proposed in this thesis constitute an end-to-end scheme, which advance the state-of-the-art in data center monitoring on both the network and application layers.

This is achieved by seizing the opportunity of placing monitoring logic into emerging programmable network platforms (Sec. 1.1) at every layer of the data center. In the network, our solutions build upon the ability of P4 switches to run sketches that derive fine-grained traffic statistics at line rate. We demonstrate through three novel sketches how to improve the *reactiveness* and *accuracy* of some important network monitoring tasks, including Distributed Denial of Service (DDoS) attack detection and identification of heavy-hitters and superspreaders flows (e.g., network port mappers). Similarly, for applications, we present the first attempt of offloading expensive cloud-native observability operations to IPU accelerators, e.g., Nvidia BlueField2 SmartNIC, showing either noteworthy cost savings or better accuracy for the tenants' applications. Figure 1.5 provides an overview of the topics covered in the thesis.

In the first contribution (Chapter 2), we limit our attention to a single switch and introduce two new algorithms for flow *cardinality estimation* that support queries on a temporal sliding window. Cardinality estimation is a required input to many classical problems in network monitoring, such as detection of DDoS attack of horizontal

[3] apart from conforming to others academic studies [60], [65]–[67].

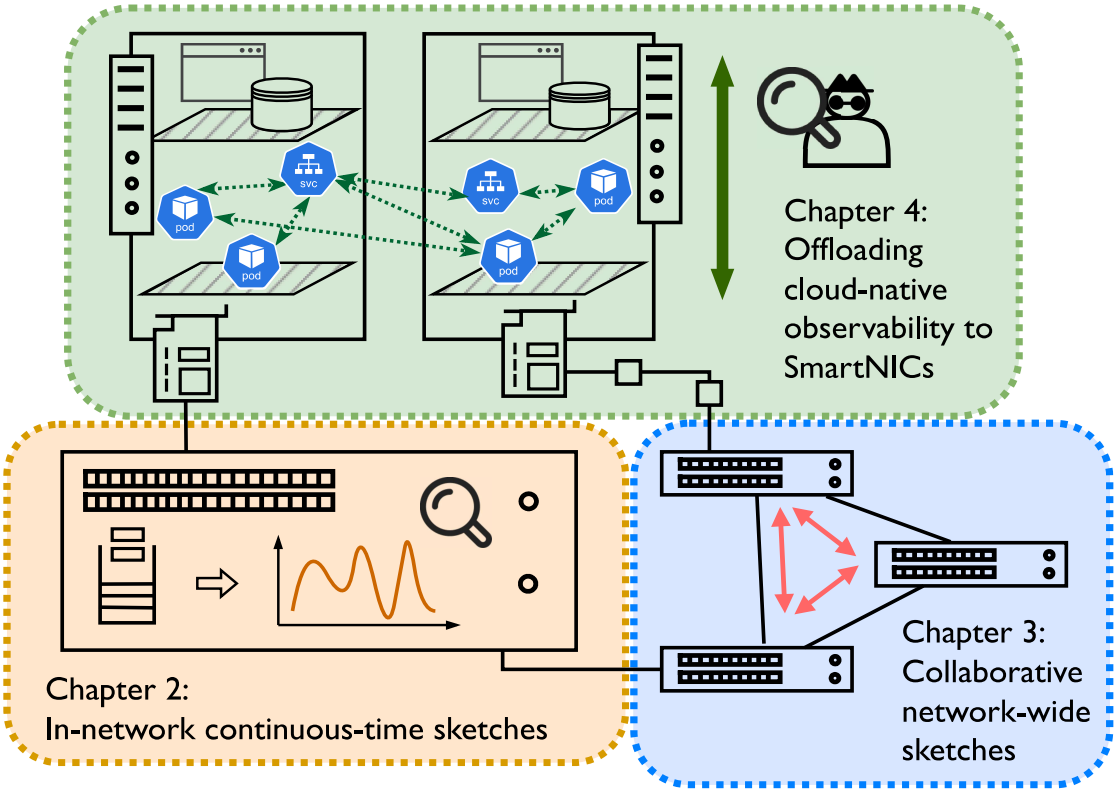


Figure 1.5: Roadmap of the thesis, covering solutions for both network and application monitoring via programmable data-plane devices.

network scanning activities, e.g., subnets or port mappers. As an example DDoS traffic is characterized by the sudden increase in the number of distinct source IP addresses contacting a victim server. Once detected this abnormal behavior, the network can block the attack by dropping or rate-limiting traffic towards the victim server. Counting the number of distinct source IP addresses for every desired server, i.e., cardinality estimation, is key to undertake repairing actions.

While existing literature have vastly studied efficient solutions for cardinality estimation over traffic streams, only few have focused on supporting queries on a temporal sliding window. In contrast, the practical assumption many systems make is to define an observation window of interest and periodically reset the sketch structure at intervals equal to a window duration. This inevitably leads to issues related to estimation accuracy. As a trivial example, consider a DDoS attack started halfway the reset interval and lasts for an observation window. Notice that the cardinality within every interval is halved. In this case, even in the best-case hypothesis that an operator has guessed right an alarm threshold exactly equal to the cardinality of the attack [4],

[4] trivially, this would be the best possible choice.

this threshold would be double of the one capable of detecting the attack. Therefore, using a simple approach such as periodic reset, the attack would go undetected and potentially cause performance degradation or, worse, total service disruption.

A second gap of existing solutions is the lack of support to *continuous-time* queries. In other words, existing solutions do not allow to estimate the flow cardinality of a traffic stream at any arbitrary point in time, but only in correspondence of the end of every reset interval. As a consequence, state-of-the-art cardinality estimation sketches may incur large delays before the detection of anomalous events, resulting in poor timeliness. Notice that reducing too much the reset period can have dramatic impact on detection accuracy due to counting very few flows within the interval. Therefore, existing solutions either sacrifice accuracy or timeliness.

We propose two novel sketches as a first contribution: TimeStamp-augmented PCSA (TS-PCSA) and Staggered HyperLogLog (ST-HLL), which address the preceding issues. The former is a novel sketch that augments the Probabilistic Counting with Stochastic Averaging (PCSA) [68] data structure with a timestamp mechanism to support continuous-time queries. The latter moves a step further and achieves the same objective without incurring the timestamp memory overhead. Notably, the basic mechanism underlying ST-HLL is general and can be adapted to other similar data structures. Through an extensive analysis we characterize analytically our solutions. Then, we evaluate their performance against real traffic traces showing that they outperform state-of-the-art solutions for the same memory footprint.

Chapter 2 contributions highlights

We advance the in-network monitoring state-of-the-art by designing two ***novel sketches*** for cardinality estimation that support ***continuous-time queries*** over a temporal sliding window.

While TS-PCSA and ST-HLL greatly boost the accuracy of continuous-time cardinality estimation sketches, they're performance are still capped by the small-sized SRAM memory available on single switches. Programmable switches available on the market, like the Intel Tofino line [8], are equipped with dozens of megabytes of memory at most. While this amount of fast SRAM memory is remaining basically constant over generations, the number of concurrent flows in the data center keeps increasing, pushing the feasibility of the these methods to their limits.

Our second contribution build on the observation that a major current limitation is that every sketch is placed as a whole on a single switch, and multiple switches work independently. Thus, the accuracy of the monitoring tasks is constrained by the memory available only in a single switch. Moreover, packets that pass multiple switches are counted repeatedly, drastically increasing the redundancy in the collected telemetry.

To close these gaps, in our second contribution (Chapter 3) we extend the scope of the problem from a single-switch sketch to a collaborative cross-switch setting. In

our solution multiple switches cooperate to the construction of a logical network-wide sketch. Specifically, we disaggregate a logical sketch on multiple physical smaller sketches that are distributed along a flow path. We characterize the accuracy of this solution for the frequency estimation monitoring task, which can be used to detect heavy-hitter flows. The main outcome of our investigation is that is possible to optimize the monitoring accuracy by selecting for every flow a subset of physical sketches to be updated. With respect to state-of-the-art solutions, we point out that in such a disaggregated setting, when the selection of the physical sketches to be updated also consider the traffic matrix and flow routing, the monitoring error can be reduced by almost a factor 2.

Chapter 3 contributions highlights

We advance the state-of-the-art of sketches for frequencies estimation by studying the collaborative approach of ***disaggregated sketches***, where multiple physical sketches along the flow paths contribute to a larger logical network-wide sketch, and showing the impact of traffic patterns on the monitoring accuracy.

Our previously described contributions have focused exclusively on network monitoring tasks. In Chapter 4, we complement our sketch-based network monitoring solutions with a new sketch-based framework for cloud-native applications observability. Altogether, with this thesis we aspire to build better telemetry tools that can help data center's administrators to inspect performance end-to-end. Specifically, we will focus on mitigating the overheads that microservice observability (Sec. 1.2.3) poses on system resources and the contentions that this causes with the monitored services. To understand the dimension of the problem, consider that Netflix collects about 2M metrics [60], Uber aggregates 500M metrics/s and stores the resulting 20M metrics/s globally [69]. Furthermore, using AWS-managed Prometheus [70] backend, ingesting 500M metrics/s would cost 21M$/month, and storage costs with 150-day retention would be 210k$/month. Therefore, coping with the escalating telemetry data represents an increasing financial strain. This observability bloat happens because, driven by the common-sense rationale that larger telemetry data volumes increases the chance of having useful data for future needs, many organizations just try to maximize the amount of collect data from their microservices. However, most of the collected telemetry data contain clutter information which is rarely useful in practice.

In our last contribution, inspired by what happened in the network monitoring space, we propose for the first time a new three-tier architecture to the problem of cloud-native application observability, which aims at reducing the costs and increase the quality of telemetry data collected from distributed microservices. Towards this goal, our main thought is to introduce the SmartNIC as an intermediate tier, sitting between the monitored nodes and the centralized monitoring server(s), to host observability functionalities. Our relatively intuitive but previously unexplored idea is that SmartNICs, thanks to their ability to run custom software, and their proximity to

the monitored services, can help to narrow the focus on *informative* data and filter out clutter telemetry data. Chapter 4 will develop a framework, named μView, and demonstrate the effectiveness of this idea in a production quality cloud-native application. In this chapter, we will solve the challenges related to accessing and manipulating the *monitored* telemetry data residing in the servers where applications run, from the *monitoring* logic residing in the external SmartNIC accelerator. Moreover, we will innovatively apply sketches to the microservices' observability domain, and present a prototype implementation on a BlueField2 SmartNIC.

Chapter 4 contributions highlights

We advance the state-of-the-art of cloud-native applications observability with a framework, μView, that combines ***streaming sketches*** and ***SmartNICs*** to ***offload observability*** tasks and improve the quality of telemetry data, while reducing overheads.

Chapter 2

Continuous-Time Sketches for Flow Cardinality Estimation

Part of the work presented in this chapter has been published in:

- A. Cornacchia, G. Bianchi, A. Bianco, and P. Giaccone, "Staggered HLL: Near-continuous-time cardinality estimation with no overhead", *Computer Communications*, vol. 193, 2022.
- A. Cornacchia, G. Bianchi, A. Bianco, and P. Giaccone, "Designing Probabilistic Flow Counting over Sliding Windows", in *2022 IEEE 11th IFIP International Conference on Performance Evaluation and Modeling in Wireless and Wired Networks (PEMWN)*, 2022.

Given a data stream which contains repeated items, the goal of cardinality estimation (i.e., distinct counting or count-unique) consists on finding how many items are distinct.[1]

For network monitoring, measuring the number of distinct flows that are active within a network traffic aggregate is a crucial task. Several applications, including intrusion detection [73]–[75], traffic engineering [76], packet scheduling and router design [77], can benefit from a fast and accurate estimation of flow cardinality. Notice that different applications adopt different *flow* definitions, and monitor the spreading behavior towards/from multiple *target* streams in parallel. For example, in a DDoS attack, several sources flood a victim host with a huge amount of connections in order to make it unavailable. Therefore, a DDoS detection system counts the number of distinct source IP addresses (i.e., flows) that are currently active within the portion of traffic destined to a single host (i.e., target stream). If the intrusion detection system

[1] In this chapter, we focus on sketches for network monitoring over traffic streams. In this context, an item is represented by a network flow, and the terminology *item* or *flow* might be used interchangeably with the same semantic.

is deployed upstream multiple target hosts, it monitors them in parallel. Similarly, detecting a port scanning attack can be performed by counting the distinct destination ports (flows) in a set of packets from the same IP subnet address space (target stream). Furthermore, the spreading nature of a flow can be taken into account also for traffic engineering, e.g., by applying specific routing policies to superspreaders [76].

Existing literature about *streaming algorithms* for cardinality estimation [68], [78]–[80] has vastly addressed how to get accurate count estimates by processing the input traffic stream using a constant-time per-packet operations and logarithmic (or sub-logarithmic) memory footprint compared to the input stream size. Unfortunately, most of these widely adopted count-unique sketch data structures do not provide natively the possibility to devise a *sliding window* approach in a way to forget outdated information and consider only "recent" traffic. Yet, the capability to track in continuous-time the traffic statistics of interest is an essential property for the aforementioned network monitoring algorithms. The sudden increase (or decrease) in the number of flows can bring evidence about anomalies or attacks, or reveal patterns in users' network activity. Refactoring such structures so as to permit them to operate using a sliding window or an exponential smoothing coefficient would provide this "short term" memory. While this problem has been studied for other sketch-based data structures [81]–[84], an analysis of the state-of-the-art shows that, among hundreds of works focusing on count-unique sketch data structures, only a couple specifically tackle the problem of devising a sliding-window-based cardinality counters [85]–[87].

To this end, in this chapter we present two novel mechanisms that enable continuous-time (CT) operations for two popular state-of-the-art cardinality estimation sketches. The first mechanisms that we propose is Staggered HyperLogLog (ST-HLL), which adds CT support to the well-known HyperLogLog (HLL) [79] sketch. The most noteworthy feature of ST-HLL is that it does not require any timestamp mechanism to discard outdated information. We refer to this property as *timestamp-free* sketch. At its core, ST-HLL leverages a single timeout, available on most commercial hardware, to circularly reset HLL registers one by one. Thus, it approximates a sliding triangular-shaped window at zero extra memory cost compared to vanilla HLL.

In the second part of the chapter, we focus on timestamp-augmented approaches, and we consider specifically Probabilistic Counting with Stochastic Averaging (PCSA) (also known as FM sketch). We propose a second CT scheme, the TimeStamp-augmented PCSA (TS-PCSA) sketch, which extends PCSA sketch to support CT queries. The key novelties of TS-PCSA are a set of optimizations aimed at compressing the number of bits for the timestamp representation. We numerically quantify the performance of ST-HLL and TS-PCSA (Sec. 2.4) at the end of the chapter. We evaluate the achieved trade-off between memory and accuracy and confront our approach against the current continuous-time HLL [85] under realistic settings.

In short, the main contributions presented in this chapter are the following:

- we propose ST-HLL, a new LogLog sketch that enables near-continuous-time measurements at no extra cost with respect to a vanilla HLL. To the best of our knowledge, ST-HLL is the first timestamp-free sketch for cardinality estimation;
- we propose TS-PCSA, an algorithm to enable PCSA counting distinct flows over sliding windows and optimized its memory footprint;
- we compare the accuracy of the two solutions, and show their different trade-off between accuracy and memory footprint. For the same memory budget, ST-HLL and TS-PCSA are up to 55% and 25% more accurate than state-of-the-art solutions, respectively.

2.1 Probabilistic count-unique sketches

Efficient cardinality estimation of a target stream via probabilistic data structures is a problem pioneered by Flajolet and Martin as early as 1985 [68], and then further addressed and improved in many subsequent works, including [78]–[80], [88]–[90]. In this section we provide the background on some of the most popular sketches and their extensions to support continuous-time measurements.

2.1.1 Static time window scenario

We first consider the problem of counting unique flows on a pre-determined and static time window. We refer to these baseline sketches as *cumulative counting* techniques, as they can only increase their estimation over time with the contribution of new arrivals. In other words, cumulative counting sketches do not provide any means to forget "old" flows within the window.

Streaming sketches for cumulative distinct counting is a long-standing research problem for which several solutions have been proposed. Some well-known examples include *Linear Counting* [88], *PCSA* [68], *MinCount* [80], [91], *Multiresolution Bitmap* [92], *LogLog* and *SuperLogLog* [78] and the latest evolutions *HyperLogLog* [79] and *HyperLogLog++* [93]. A comprehensive overview and quantitative comparison can be found in the survey papers of Metwally et al. [94] and Harmouch et al. [95]. In this section we focus on two widely adopted sketches, namely Probabilistic Counting with Stochastic Averaging [68] and HyperLogLog [79], and provide the necessary background required for the remaining of this chapter. We chose these sketches as they are implemented in several commercial Big Data query engines. For instance, HLL is part of Google BigQuery [96], Microsoft Kusto Query Language [97] and Facebook distributed SQL engine [98].

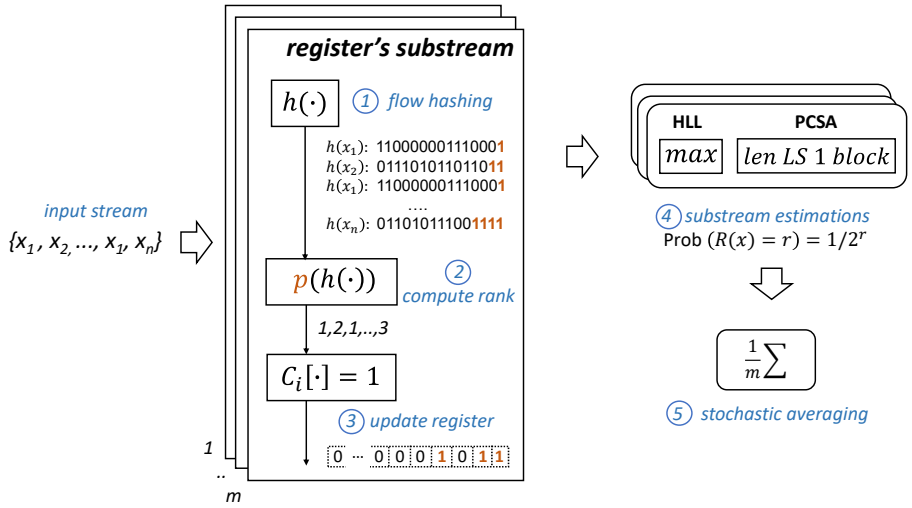


Figure 2.1: The workflow of the two most common cardinality estimation sketches: HLL [79] and PCSA [68] to estimate the cardinality of a set of flows from an input traffic stream.

HyperLogLog (HLL) and Probabilistic Counting with Stochastic Averaging (PCSA)

Figure 2.1 illustrates the logical scheme adopted by both PCSA and HLL to estimate n. While some operations reported in the figure can be avoided depending on the algorithm, in the interest of comprehensiveness it is useful to abstract both algorithms under a common logical framework and highlight the implementation details when relevant.

The goal is to evaluate the cardinality n of a set of flows X, i.e., with $|X| = n$, contained in a stream of packets. The packet stream can be viewed as a flow multiset, i.e., a set where each element can appear multiple times (in the example of Fig. 2.1 this is $\{x_1, x_2, .., x_1, x_n\}$). Each packet in the input traffic stream is assigned uniformly at random to one out of m traffic substream, each substream updating a different memory register as described in the following. Let's initially focus on a single substream, with cardinality n/m on average by construction. For each packet in the substream, these sketches evaluate (step ①) a hash function $h(x)$ on the flow identifiers $x \in X$. Then, by looking at the binary representation of $h(x)$, step ② consists in extracting the left-most position[2] $p(h(x))$ where a bit equal to 1 is found. This position is referred to as the *rank* of a flow x and is the key quantity used to estimate the cardinality of the substream. In the example of Figure 2.1, the resulting ranks are 1,2,4 for flow x_1, x_2 and x_n, respectively. Next, the ranks are stored in a bitmap C_i, with boolean entries $C_i[b]$, where b is the bth Least Significant (LS) bit. For each flow x, the bitmap is modified as $C_i[R(x)] = 1$ to store the flow rank (step ③). Note that packets belonging

[2]We assume indexes are counted starting from one.

to the same flow result into the same hash value and thus, after the first packet of a flow, subsequent packets have no effect on the bitmap state (e.g., flow x_1). In other words, every flow is counted only once by construction.

Let $R(x) = p(h(x))$ be the random variable associated with the rank of a flow x and observe that $\text{Prob}(R(x) = r) = 2^{-r}$, i.e., $R(x)$ follows a geometric distribution, given the uniformity property of the hash function. This suggests the main idea behind the HLL and PCSA's estimation. Consider the state of a bitmap C_i, after the insertion of 2^r flows. *Ideally*, i.e., if we could average over a large number of realizations of the traffic stream, we would expect to observe in C_i a block of zeros in the Most Significant (MS) bits and a block of r ones in the LS bits. Thus, according to the property discussed before, we would estimate the cardinality as 2^r where r is the number of ones in C_i.

While this holds on average, in practice we are provided with a single realization of each substream. Due to the randomness of the traffic and hash function, C_i is typically composed of a block of zeros (MS bits), followed by non-continuous sequences of zeros and ones, and finally a block of only ones (LS bits). PCSA and HLL provide two different — although very similar — techniques to interpret the bitmaps and estimate cardinality (steps ④ and ⑤).

The next step (④) consists in choosing which information in the bitmap to use to estimate cardinality within the i-th substream.

- In **HLL**, consider only the left-most bit equal to 1 in the bitmask, i.e., the maximum rank $R^i_{\max} = \max_{x \in X} p(h(x))$, and approximate the distinct number of flows in a substream as $n/m = 2^{R^i_{\max}}$.

- In **PCSA**, consider the entire bitmask of ranks (i.e., register) and approximate the number of distinct flows in a substream as $n/m = 2^{k_i^{(1)}}$ where $k_i^{(1)}$ is the size of the rightmost one block, or equivalently $k_i^{(1)} + 1$ corresponds to the position of the first zero in C_i starting from the least significant bit;

It is worth noticing that, from an implementation perspective, PCSA requires the entire bitmask C to be stored in a memory *register*, whereas HLL only to keeps track of the maximum value of the rank. Since only the position of the maximum within the bitmap needs to be stored, the asymptotic memory cost of HLL is $m \log_2 \log_2 (n/m)$ bits, whereas for PCSA is $m \log_2 (n/m)$ bits. Thus, HLL can be deployed using smaller memory registers than PCSA, by a logarithmic factor in the number of bits.

Up to now we have focused on individual substreams. However, it's easy to see that within a single stream the above techniques are characterized by a large variance. In HLL, a single occurrence of an outlier flow, i.e., a flow with $R(x) \gg \log_2 n$, would significantly bias the estimation. Likewise, in PCSA it's enough the "absence" of a single rank in the bitmap to underestimate cardinality. *Stochastic averaging* [68] is a technique that was introduced to reduce the estimation variance. It leverages the m traffic

substreams and mimics m independent estimators[3]. The estimations obtained from the m substreams are averaged (step ⑤) to reduce noisy fluctuations.

- In **HLL**, the harmonic mean of the substream estimations $2^{R^i_{max}}$ is used to filter the impact of outlier flows. The overall cardinality estimate $\hat{n}$ is given by:

$$\hat{n} = \alpha_m \frac{m^2}{\sum_{i=1}^{m} 2^{-R^i_{\max}}} \tag{2.1}$$

  where α_m is a constant factor that depends on m and derived in [79]. Notice that since the harmonic mean should be near n/m, it is further scaled by a factor m to estimate the overall stream cardinality n.

- In **PCSA**, the substream estimates $2^{k_i^{(1)}}$ are multiplied each other and scaled by a factor m/α to obtain the overall cardinality estimate $\hat{n}$.

$$\hat{n} = \frac{m}{\alpha} \cdot 2^{\frac{1}{m}\sum_{i=1}^{m} k_i^{(1)}} \tag{2.2}$$

  where $\alpha = 0.77351$ [68] corrects a systematic bias. Essentially, PCSA uses the arithmetic mean of the rightmost one-blocks lengths $k_i^{(1)}$ to estimate the "average" substream cardinality. Then, the overall cardinality is obtained by multiplying the substream estimates and scaling by a factor m/α.

Error guarantees . In terms of accuracy, Flajolet et al. [79] demonstrated that for HLL, the harmonic mean across register estimations gives a relative error to $1.04/\sqrt{m}$ on average. Notably, this guarantee is almost 30% better than its predecessor, LogLog [78], whose relative estimation error is $1.30/\sqrt{m}$. Instead, PCSA, while requiring more memory, it's more accurate, with an average error as low as $0.78/\sqrt{m}$. Intuitively, this is because PCSA uses the rightmost one-blocks lengths $k_i^{(1)}$ to estimate substream cardinality, which is a more robust statistic than the maximum rank.

Time binning

The preceding sketches very efficiently address the main obstacle behind the distinct counting problem, namely how to efficiently remember which items have already been seen in the past to avoid double counting. In practice network managers need to collect measurements not only on a single interval, but for the entire up-time

[3]Stochastic averaging can be implemented to mimic the effect of multiple independent estimations using just a single hash function. The first $\log_2(m)$ bits of $h(x)$ are used to assign flow x to a substream and the remaining bits to extract the rank. In this efficient implementation, the hash computation shown in Figure 2.1 would be performed before, and not within, the register's substream block.

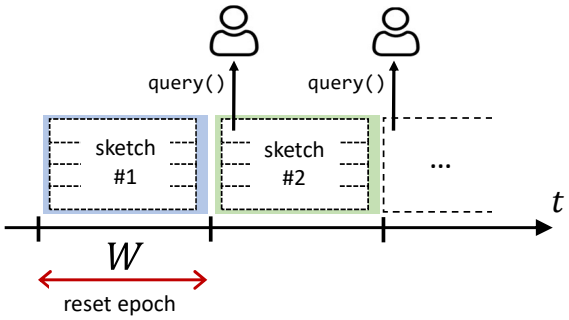


Figure 2.2: Time binning: the data structure is reset at the end of each reset interval (epoch) and the cardinality is estimated on the entire interval.

of the network. A typical deployment strategy is to operate the sketch structure on slotted time intervals (i.e., *epochs*) and reset the entire structure at the end of every interval before starting the next one, as shown in Figure 2.2. However, this solution, which we refer to as time binning, presents several downsides. An immediate and inherent drawback is that time binning fails to detect the spreading behavior of a traffic aggregate whenever this happens in the middle of two consecutive epochs. This is because in this case the corresponding flow cardinality would be split across two independent estimates.

Second, by working with static and non-overlapping measurement intervals, this simple solution cannot answer queries about a past observation window at arbitrary point in time, but only synchronously to interval boundaries (i.e., reset times). This is because the accuracy of the results is determined by the number of registers and their size, which is in turn chosen depending on the expected number of flows within the observation window.

Third, it is challenging to set the proper interval size, as it requires to strike a delicate balance between latency and detection capabilities. A large interval would introduce high reaction delays, being results available only at the end of the interval. At the same time, short intervals might not be able to detect a slow spreader.

2.1.2 Sliding window scenario

Many network monitoring application would rather benefit from the capability of "tracking", in continuous time, the spreading behavior of traffic aggregate. Unlike in the classical and consolidated time binning strategy, an alternative solution are sketches that natively — i.e., by initial design — support continuous-time operations.

Existing sketches that by design offer this capability can be referred to as *timestamp-augmented*. This is because timestamp-augmented algorithms [77], [85], [99] typically build upon a sketch for cumulative counting and augment the information maintained in the sketch data structure with timestamp information. Since the sketched

information is tagged temporally, these techniques can distinguish and disregard outdated information with arbitrary precision by choosing a sufficiently high resolution for representing the time. In fact, by properly choosing the resolution, they can precisely use only the knowledge coming from traffic within the current observation window. As a downside, timestamp-augmented solutions are characterized by a remarkably high resource consumption, due to the need of storing timestamps and the complexity of managing outdated entries. As such, timestamp-augmented solutions pose a few challenges for their implementation on resource-constrained programmable switches [100].

HyperLogLog over sliding windows

In this section we review existing timestamp-augmented extensions of the HLL algorithm to support continuous-time measurements over a sliding window. An obvious solution for this problem is to buffer all the ranks observed within the current observation window, for every substream. In this way, at any point in time the ranks of all "recent" flows are kept in memory. Therefore, this straw man would be capable of deriving the m maxima for every substream, and answer time-range queries using the standard HLL mechanisms discussed in Sec. 2.1. However, this simple solution doesn't scale well, as the amount of ranks to keep in memory can grow fast for increasing window sizes. *Sliding HLL* [85] improves the straw man solution, thanks to the intuition that only the ranks eligible to become maxima in the future need to be maintained. Sliding HLL keeps m distinct lists called List of Possible Future Maxima (LPFM), containing pairs of the kind $\langle$*timestamp, rank*$\rangle$. When the rank of a new item is inserted into one of the LPFMs, all ranks smaller than the new one are evicted from the list, together with the ranks oldest than a past window. Therefore, differently from the straw man solution, the arrival of a large *possible future maxima* evicts several smallest entries and permits freeing up the LPFMs. As a consequence, Sliding HLL is functionally equivalent to the exhaustive storage solution (i.e., straw man), but with significantly lower memory consumption needed to maintain recent information.

As shown by the authors, the asymptotic memory cost of Sliding HLL can be upper bounded by $(b+\log_2\log_2(n/m))m\ln(n/m)$ bits, being n the maximum number of flows per window and assuming b-bit timestamps. Compared to the asymptotic cost of $m\log_2\log_2(n/m)$ of vanilla HLL, this cost is higher due to storing the LPFMs entries.

2.2 The Staggered HyperLogLog continuous-time sketch

Timestamp-augmented solutions are characterized by augmenting the synoptic information contained in a counting sketch with information about the insertion time. Managing timestamps involves (1) additional memory overhead, which grows with the timestamp resolution, and (2) high complexity to ignore outdated entries during

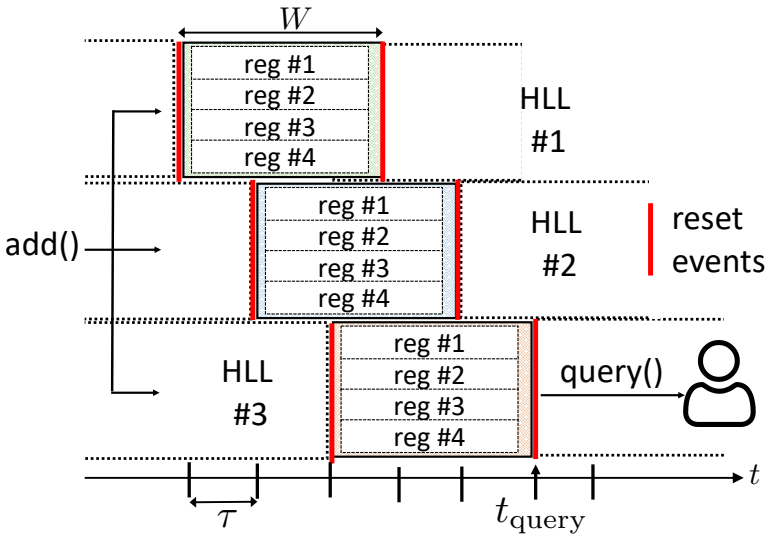


Figure 2.3: Staggering three HLLs, each of them with 4 registers. The query reads a single HLL and can happen just before resetting the register.

queries and periodically prune them. Together these two factors make timestamp-augmented solutions challenging to implement on resource-constrained programmable switches.

This leads to the question that motivates our first contribution: is there a way to *turn* a HLL-like data structure into a continuous-time one, *at no extra resource cost* in terms of increased number of internal counters, or increased per-counter memory, or extra hardware? We introduce a novel solution that satisfies all preceding requirements, To the best of our knowledge, our solution is the first sketch for cardinality estimation that supports continuous-time queries over sliding windows while keeping the same complexity of vanilla HLL. We denote our solution as *Staggered HyperLogLog (ST-HLL)*, as it is based on the idea of staggering the internal registers of a single HLL to approximating a sliding window filter without dealing with timestamps. ST-HLL initiates a new family of cardinality estimation algorithms, which we refer to as *timestamp-free*. While applied to HLL, our staggered approach could in principle be adapted to other algorithms, provided they preserve the same functional architecture based on registers (e.g., MinCount [80] and LogLog [78] alternatives).

The next sections describe how ST-HLL achieves our goals. In Sec. 2.2.1 we highlight a general scheme that will serve as a reference solution in the design of ST-HLL. We discuss why ST-HLL takes inspiration from this scheme, but it approximates it at a much smaller memory cost. Then, we explain some analytical findings (Sec. 2.2.3) which are pivotal to properly scale and equalize the registers' outputs when answering queries with our solution (Sec. 2.2.4).

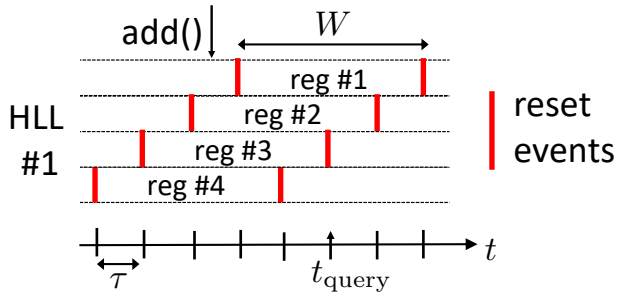


Figure 2.4: Staggering the 4 registers using a single HLL.

2.2.1 Problem formulation

Before delving into the details of our scheme, let us first describe a straw man solution to the problem. If we could afford to deploy and run multiple HLL sketches in parallel, as opposed to a single HLL reset every W seconds, one could rely on a staggered approach of multiple HLL sketches, as shown in Fig. 2.3. Remember that the main problem of time binning was to dimension the time epoch: to achieve low reaction latency, one had to choose a small time epoch, thus compromising accuracy. In contrast, such a combined straw man structure would permit to employ a possibly long measurement window W, but would also allow to track the traffic dynamics at a finer timescale $\tau = W/N$, where N is the number of parallel HLLs deployed. Each HLL would in fact "learn" about the incoming new items (equivalent to new traffic flows), but only one "active" HLL (the "oldest" one) would have accumulated arrivals for W time and would be in charge to report the current cardinality estimation (the bottom one in Fig. 2.3). Hence, the straw man solution could achieve both high accuracy and low reaction latency.

Such a straightforward approach is practically ruled out by the unbearable N-fold increase in required resources, as N fully fledged HLL should be deployed instead of a single one. Notice that for $N \to \infty$ this straw man would attain arbitrarily low time granularity ($\tau \to 0$), thus mimicking an ideal sliding window filter.

Despite its unbearable complexity, this baseline approach naturally suggests the following apparently naïve idea:

would it be possible to achieve the same result by staggering the internal registers of a single HLL?

As shown in Fig. 2.4, this operation would come along with *zero* additional resource cost - it would suffice to deploy a staggered timer which periodically resets only *one single HLL register at a time* - anything else would be left unmodified.

While being a relatively simple intuition, we faced a major technical caveat to turn such a naïve idea into an effective approach. Since the registers are staggered in time, they now "count" items on different time windows. Hence, we cannot neither resort

to the classical HLL stochastic averaging process to reduce the estimation error, nor their simple aggregation would work. It is trivial to check (see also discussion at the end of Sec. 2.2.2) that the estimation error would *increase* rather than *decrease*. In the following, we devise an analytical framework to model this behavior and tackle the issue of aggregating staggered registers.

2.2.2 Notation and assumptions

Similarly to a standard HLL data structure, let us consider m registers. We introduce two different time scales: i) a *smoothing timescale* W, and ii) an *updating timescale* τ. W is a window size which defines the target memory depth of the data structure. $\tau \ll W$ is the timescale at which the HLL is updated. For reasons that will become clear later on, it is convenient to set $\tau = 2W/m$. Considering that m is usually set to a relatively large value, say 512 or 1024, a relatively long window W, e.g., 4 minutes, would be updated at a rate of about 1 or 2 times per second, thus producing a near-continuous-time effect for monitoring applications which aim to follow traffic dynamics and updates at a timescale in the order of seconds.

Our staggered HLL approach builds upon the idea of resetting only *one* among the m registers at each time slot τ. More specifically, we reset counters in a circular fashion, as shown in Fig. 2.5. This implies that each register in the HLL will track a *different* time period: At the time in which a register is reset, the previously reset one will have tracked its fraction of traffic arrived in the latter slot τ, the second previous one will have tracked a time interval $2 \cdot \tau$, and so on. Remember that in HLL (Sec. 2.1) the registers contain the maximum observed rank R^i_{max}. For ease of notation, it is convenient to rank all the registers based on the reset time. Let's denote with M_i the value (i.e., R^i_{max}) stored in the i-th most recent register to have been reset, with $1 \leq i \leq m$. Owing to the above convenient notation, at an arbitrary time instant t, register M_i will have counted arrivals in the interval $(\lfloor t/\tau \rfloor \tau - (i-1)\tau, t]$.

Let us now assume throughout the remainder of this chapter that *the number of new items recorded by each register is proportional to the size of its measurement period* - see also Fig. 2.4 (we'll discuss this assumption in more depth in Sec. 2.4.2). If we "read" the status of all registers at a time t exactly in the middle of the updating time slot τ, and we assume that the overall rate of new arrivals is λ items per second, then register i, which tracks $1/m$-th of the traffic, will have recorded a fraction λ/m of new arrivals for a time period $(i-1/2) \cdot \tau = (i-1/2) \cdot \frac{2W}{m}$. Hence, by *summing* the content of all the m counters, we would trivially obtain an estimate[4] of the number of new

[4]This explains why we have specifically selected $\tau = 2W/m$. With such setting, the "shorter" time of the most recently reset counters is compensated by the "older" counters which can account for up to a time interval of $2W$ before being reset.

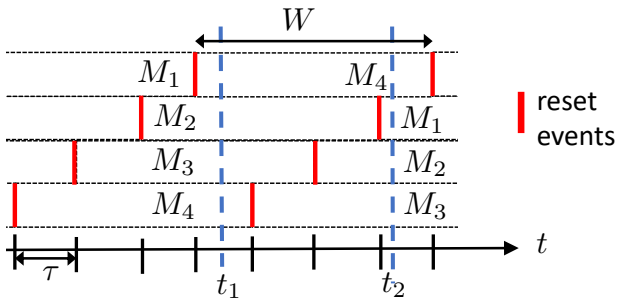


Figure 2.5: Register-based Staggered HyperLogLog with 4 registers sampled at time t_1 (when the sorted sequence of registers is $[M_1, M_2, M_3, M_4]$) and at time t_2 (when the sorted sequence of registers is $[M_2, M_3, M_4, M_1]$).

arrived items in a time period W, as shown in the following:

$$\sum_{i=1}^{m} \frac{\lambda}{m}(i-1/2)\cdot\frac{2W}{m} = \lambda W$$

Unfortunately, summing the content of all counters is not a viable approach, as the variance of the so-obtained estimator would dramatically *increase* rather than decrease. The estimation error would reduce by taking a stochastic average, but this is not anymore straightforward in our case, as the registers record estimates taken on different time periods, hence they ultimately estimate *different* quantities.

2.2.3 Analytical model of heterogeneous registers

As anticipated in the previous section, the above described construction brings about a crucial difference with respect to the classical HLL data structure. In standard HLL, the uniform split of the traffic across the m deployed registers makes such registers *statistically homogeneous*, i.e., each register yields an estimate of the same quantity. In our case, each register instead used a *different* measurement window, and therefore accounts for a different number of items.

Since in the following analysis we are considering a single register, let us simplify notation and avoid indexing the i-th register with the superscript, e.g., $R := R^i$, etc. To establish a quantitative insight on how each register' statistics depend on the number of accounted items, it is instructive to note that this relation would become trivial if the value in each register, instead of being an integer geometric random variable R, were approximated by a *continuous random variable* denoted as $\tilde{R}$. Indeed, for such "continuous"-valued register, the probability that a new arriving item "hits" a given (real-valued) register position would now follow an exponential law instead of the geometric one introduced in Sec. 2.1, i.e., $\text{Prob}(\tilde{R} > x) = 2^{-x}$.

Let us now assume that n items are accounted by the register. It readily follows that the random variable R_{max} representing the current register's state, i.e., the maximum

value among the n values drawn from the above exponential distribution, has cumulative probability distribution given by the product of the n exponential distributions; in formulae:

$$\text{Prob}(\tilde{R}_{\max} \leq x) = \left(1 - 2^{-x}\right)^n$$

Such continuous distribution is very convenient, as it yields very simple closed-form expressions for the statistical moments[5]. Routine computation indeed yields the register's expected value:

$$\mathbb{E}[\tilde{R}_{\max}] = \frac{H_n}{\ln(2)} \overset{n\to\infty}{\approx} \frac{\gamma}{\ln(2)} + \log_2(n) \tag{2.3}$$

where $H_n = \sum_{i=1}^{n} 1/i$ are the well known Harmonic numbers H_n, and the approximation follows from the definition of the Euler constant $\gamma = \lim_{n\to\infty}(H_n - \ln n) = 0.5772$.

For our purposes, it is important to further note that the variance of the register does not diverge for large n, but rather converges to a constant quantity:

$$\text{Var}[\tilde{R}_{\max}] = \frac{H_n^{(2)}}{\ln(2)^2} \overset{n\to\infty}{\approx} \frac{\pi^2}{6\ln(2)^2} = 3.424$$

where $H_n^{(r)} = \sum_{i=1}^{n} 1/i^r$ is the Harmonic number of order r, and $\lim_{n\to\infty} H_n^{(2)} = \pi^2/6$. In Fig. 2.6 we show the coefficient of variation of $R_{\max}$, denoted as:

$$\text{Cv}(\tilde{R}_{\max}) = \frac{\sqrt{\text{Var}(\tilde{R}_{\max})}}{\mathbb{E}(\tilde{R}_{\max})}$$

which can be seen as the relative error of a HLL for a given number of flows counted in a HLL register. From the plot, it is clear that the most inaccurate registers are the ones with few arrived flows. This is a crucial observation, since in ST-HLL, by constructions, the error depends on the register, differently from a standard HLL in which all the registers are fed by homogeneous arrivals, leading to errors identical on average. The main problem arises when combining together the estimations provided by the registers, since the average should take into account the different levels of accuracy characterizing each register.

So far, for simplicity, we have assumed "continuous" - exponentially distributed - register values, whereas a HLL register of course can only assume integer values. However, this is trivially accommodated by adding a constant 1/2 that accounts for

[5] Being a positive random variable, the moments can be directly computed from the complementary cumulative probability distribution as

$$E\left[X^r\right] = r\int_{x=0}^{\infty} x^{r-1} P(X > x)\,dx = r\int_{0}^{\infty} x^{r-1}\left(1 - \left(1 - 2^{-x}\right)^n\right)dx$$

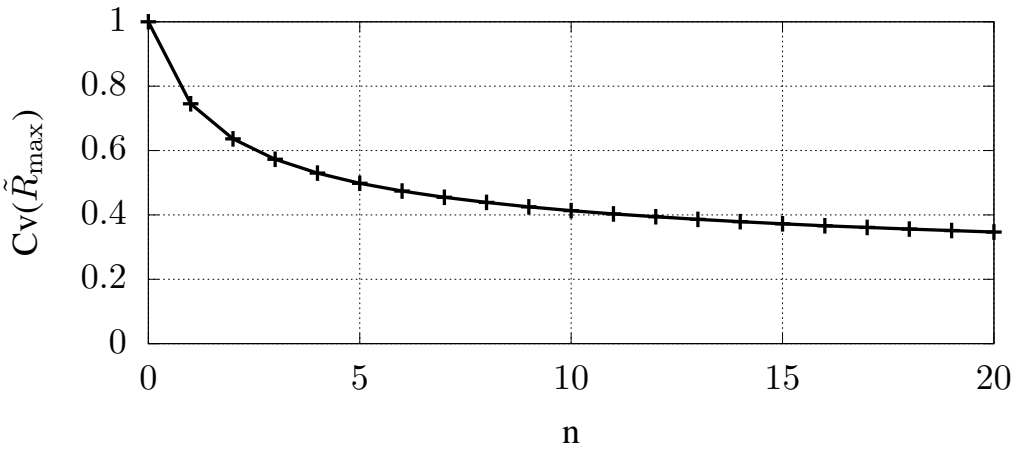


Figure 2.6: Evaluation of relative error $\mathrm{Cv}(\tilde{R}_{\max})$ in a HLL register given a number n of recorded flows.

such quantization. More formally, this extra constant yields from eq. (2.8) in [101], which provides an explicit approximation for the expectation of the maximum of i.i.d. geometric distributions. It follows that equality (2.3) is readily adapted to the discrete case as

$$\mathbb{E}[R_{\max}] \approx \frac{1}{2} + \mathbb{E}[\tilde{R}_{\max}] = \gamma' + \log_2(n) \tag{2.4}$$

where the superscript g refers to the geometric distribution of the actual HLL registers, and $\gamma' = 1/2 + \gamma/\ln(2) = 1.332$. Observe that (2.4) provides the most accurate formula we will need in the following for actual value stored in a HLL counter, given n recorded flows. Fig. 2.7 shows the true value for $\mathbb{E}[R_{\max}]$ and its approximation according to (2.4). The relative error is very small (e.g., $< 4\%$ for $n = 5$) and decreases with n. Thus, the approximation (2.4) is accurate also for small values of n.

2.2.4 Algorithm design

Querying heterogeneous registers: scaling and equalization

We now have all the tools necessary to specify our proposed ST-HLL scheme, described in details in the pseudocode of Fig. 2.4. In a nutshell, we address the hurdle introduced in the previous section by "equalizing" the registers by a proper scaling, in such a way that all registers will estimate the rate on the "same" time window. This permits to exploit the well-known results for classical HLLs and compensate for the systematic errors with standard techniques.

Assume, without loss of generality, that we are interested in reading the overall HLL count at a time instant t corresponding to the end of a time slot τ, i.e. right before the "next" register is reset, and define W_i as the time window span of the i-th register M_i,

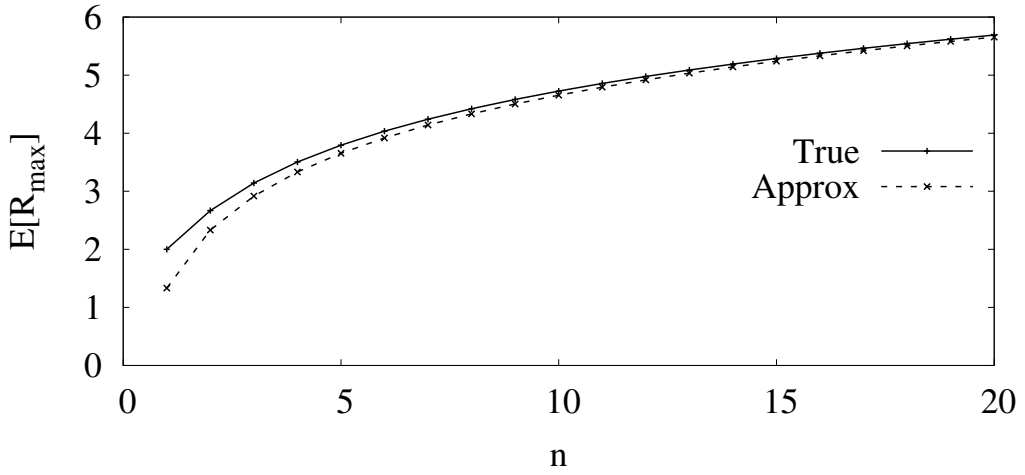


Figure 2.7: Evaluation of the error in $\mathbb{E}[R_{\max}]$ when considering the approximation (2.4).

$i \in \{1, m\}$. Owing to our notation,

$$W_i = i \times \tau = i\frac{2W}{m}.$$

Let now t be the actual HLL reading time. Since each register receives a fraction $1/m$ of the items, the expected number of items n_i accounted by each register M_i in its time window W_i is:

$$n_i = \int_{t-W_i}^{t} \frac{\lambda(t)}{m} dt$$

If we now assume a constant arrival rate within the past W_i interval of time, i.e., $\lambda(t) = \lambda$, then:

$$n_i = \frac{\lambda}{m}\frac{2W}{m}i \tag{2.5}$$

which provides an explicit relation between λ and the expected number of arrivals in a specific register. By inverting (2.5) it, we can derive a local estimation $\hat{\lambda}_i$ of the arrival rate at register M_i:

$$\hat{\lambda}_i = \frac{n_i m^2}{2Wi} \tag{2.6}$$

Thanks to (2.4), we can estimate the expected value of a generic register M_i based on its corresponding window W_i as

$$\mathbb{E}[M_i] = \gamma' + \log_2(n_i)$$

and then, by a first order approximation, we can claim

$$n_i = 2^{M_i - \gamma'}$$

1: **procedure** QUERY()
2: **for** $i \leftarrow 1$ to m **do** ▷ For each register
3: $\hat{\lambda}_i = 2^{(M_i-1)} \frac{m^2}{Wi}$ ▷ Estimate λ locally
4: $\hat{\lambda} = \alpha_m$ HARMONICMEAN$([\hat{\lambda}_i]_{i=1}^m)$ ▷ Estimate λ globally
5: **if** $\hat{\lambda} \leq \frac{5}{2} m$ **then** ▷ Small range corrections - linear counting
6: $v = |\{M_i, i \in \{1,\ldots,m\} | M_i = 0\}|$ ▷ Num. zero registers
7: **if** $v \neq 0$ **then**
8: **return** $m \ln(m/v)$
9: **if** $\hat{\lambda} > \frac{1}{30} 2^{32}$ **then** ▷ Large range corrections
10: **return** $-2^{32} \ln(1 - \hat{\lambda}/2^{32})$
11: **return** $\hat{\lambda}$ ▷ Medium range - no corrections

Figure 2.8: The query algorithm for ST-HLL.

which allows to rewrite (2.6) as follows:

$$\hat{\lambda}_i = 2^{M_i - \gamma'} \frac{m^2}{2Wi} \tag{2.7}$$

It follows that the above equation (2.7) permits to turn the *heterogeneous* registry values M_i into estimators $\hat{\lambda}_i$ of a *same* quantity λ, namely the overall arrival rate of new items to the HLL data structure. We can hence now proceed exactly as in the case of a standard HLL, i.e., compute the harmonic mean of all $\hat{\lambda}_i$ by scaling by a proper factor α_m, which has been computed in [79] and also accounts for γ' in (2.5). Approximately, $\alpha_m \approx 0.7$.

The pseudocode of the query function is reported in Fig. 2.8. At a first step, a local estimation of the rate at each register is evaluated using (2.7) (ln. 2-3). As second step, the local estimations are combined though a harmonic average, according to the standard methodology for HLL (ln. 4). Finally, some well-known correction factors to the rate estimator, as derived in [79] for standard HLL, are applied to consider different level of "occupancy" of each register (ln. 5-11).

In terms of implementation complexity, the proposed solution is identical to a standard HLL, without the need of additional memory as in alternative solutions, as discussed in Sec. 3.4. Thus, the total memory is $m \log_2 \log_2(n/m)$ where n is the maximum number of flows during the observation window. An internal timer must be available to trigger the reset of a single register every τ time.

Low-counter variance compensation

As discussed in Sec. 2.2.3, Fig. 2.6 shows that the accuracy of the estimated number of flows in a register greatly depends on the number of recorded flows, and consequently the estimators referring to the lowest ranked registers $(M_1, M_2, \ldots)$ are the most inaccurate. In order to reduce the variance in the final evaluation of the rate estimator, we propose the following heuristic approach: when computing the harmonic mean (ln. 3 in the pseudocode of Fig. 2.8), we consider only the registers with index i larger than a given threshold $i_{\min}$. Even if $i_{\min}$ requires some tuning, we could observe

a good trade-off between accuracy and temporal granularity by setting $i_{\min} \approx m/8$, i.e., we neglect the 12.5% of the lowest (supposed) registers.

This variant of ST-HLL with the variance compensation technique will be later denoted as ST-HLL+. Notice that in ST-HLL+ all the counters are updated as usual when a new packet arrives, whereas the compensation is applied only at query time during the final average computation. We will show in Sec. 2.4 that this simple approach to variance compensation works well in practice.

Analogy with a triangular low-pass filter

Note our ST-HLL algorithm mimics a "triangular" sliding window of depth W, and not a "rectangular" window. This can be easily understood from the fact that it combines low-index registers, which use window depths lower than W, with high-index registers which instead measure items on a time period which may span up to twice the size of the nominal window W.

More formally, being $\lambda(t)$ the instantaneous arrival rate, a classical sliding window would produce a smoothed measured rate $r(t) = \int_{t-W}^{t} \lambda(x)/W dx$. Instead, when the number of HLL registers is large, our smoothed measured rate would converge to:

$$r(t) = \int_{t-2W}^{t} \lambda(x) \left(1 - \frac{t-x}{2W}\right) dx.$$

i.e., the convolution of $\lambda(t)$ with a triangle gate function. Hence, even assuming an ideal operation, our results are in principle expected to (slightly, if W is relatively small with respect to the traffic dynamics) differ from those obtained by a pure sliding window. We will show in Sec. 2.4 that this approach works well in practice, and outline the effects of this approximation.

2.3 TS-PCSA sketch

In the previous section we have proposed a timestamp-free version of the HLL sketch, which approximates a triangular-shaped low-pass filter by running a periodic *staggered* reset of HLL registers. As a different line of work, we now focus on timestamp-augmented approaches, and specifically we consider PCSA [68]. We show, as a second contribution, how to extend it to support sliding windows operations and discuss several optimizations to reduce the overhead of storing high-resolution timestamps. Our technique allows using few-bit low-resolution timestamps and takes advantage of stochastic averaging to smooth the timestamp quantization error.

In the next section (Sec. 2.3) we present our algorithm TS-PCSA and discuss its time and space complexity, while in Sec. 2.3.2 we propose TS-PCSA+, an optimization of the baseline algorithm to reduce the memory footprint required for storing timestamps.

1: **procedure** QUERY (t, W)
2: $a = 0$ ▷ Init the accumulator for the average
3: **for** $i \leftarrow 0 \rightarrow (m-1)$ **do** ▷ For each counter
4: $\delta_i = \frac{\tau}{m-1} i - \frac{\tau}{2}$ ▷ Compute the temporal offset
5: **for** $k \leftarrow 0 \rightarrow (K-1)$ **do** ▷ For each bit starting from LS bit
6: **if** $T_i[k] + \delta_i < t - W$ **then** ▷ Check timestamp
7: **break** ▷ Leave the search loop if outside the window
8: $a = a + k$ ▷ Accumulate k for the average
9: $n = m \times 2^{a/m}$ ▷ Compute the average and the final count
10: **return** $n/0.775/W$ ▷ Output the final rate with bias compensation

Figure 2.9: Querying at time t a TS-PCSA sketch (• code) or a TS-PCSA+ sketch (•• code), tracking a sliding window W.

2.3.1 Our approach at a glance

We denote our solution TS-PCSA, which stands for TimeStamp-augmented PCSA. We assume to have m arrays (also denoted as "registers" with abuse of language), each of them storing K timestamps. Notice that, as in HLL (Sec. 2.2.2), each register counts $n_i = n/m$, where n is the total number of flows within an observation window. We select m being a power of 2. Since registers now store timestamps instead of bitmaps, in the following let us denote them as $T_0, \ldots, T_{m-1}$, and the timestamp stored in the $(k+1)$th LS bit of the ith register as $T_i[k]$. This structure mimics the standard PCSA with m registers and K bits for each register (Sec. 2.1), but each bit of the PCSA register is instead storing a b-bit timestamp. Our algorithm does not restrict the capabilities of a standard PCSA, and it supports the same operations. First, we describe the ADD() and QUERY() operations in TS-PCSA and discussing their complexity.

Adding a flow is substantially equivalent to a standard PCSA. The ADD() operation exploits the $\log_2 m$ LS bits to choose a register (i.e., select the substream), while the remaining bits are used to compute the rank of the flow. Differently from PCSA, TS-PCSA updates the register in the position identified by the rank with the flow arrival time, instead of with a single bit.

Querying the flow cardinality in the last W observation window at time t is supported through the QUERY() operation, whose pseudocode is reported in Fig. 2.9. We temporarily ignore the code sections referring to TS-PCSA+, that will be clarified later in Sec. 2.3.2. Let's denote as *valid* all the timestamps that fall within the observation window. The main idea is to find across all the registers the average position at which the timestamp becomes invalid, starting from the rightmost position. The main loop (ln. 3-8) finds for each register such position (equal to k in ln. 8) and computes the average a/m, from which the cardinality is derived $\propto 2^{a/m}$ (ln. 9). Finally, an estimate of the flow arrival *rate* is computed by applying the same bias correction of PCSA and dividing by the length of the observation window.

Example. Fig. 2.10 shows a toy example of a TS-PCSA with 2 registers, each storing 7 timestamps. The traffic is constituted by a sequence of 64 flow arrivals, with flow x_i arriving at time t_i. For ease of explanation, we assume flows consisting of a

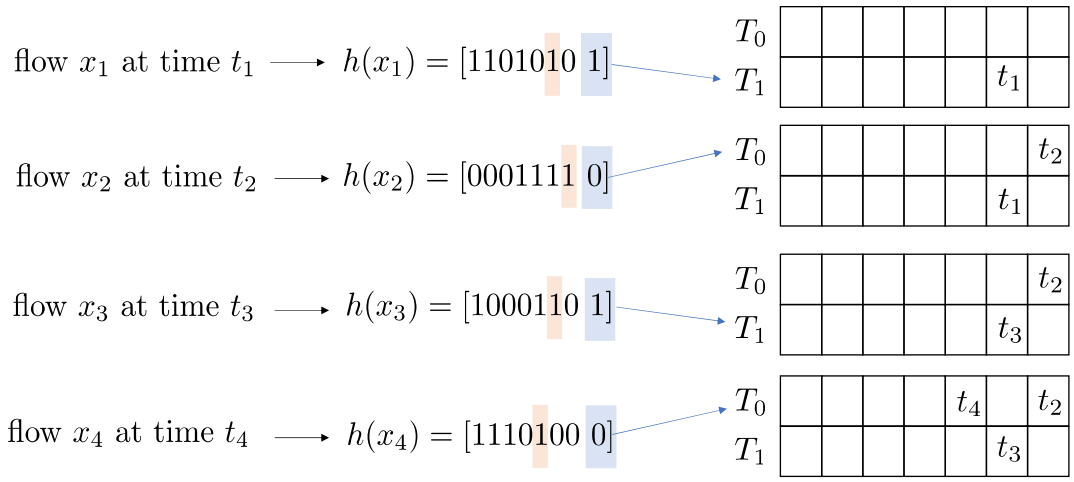


Figure 2.10: TS-PCSA example when inserting the first 4 flows.

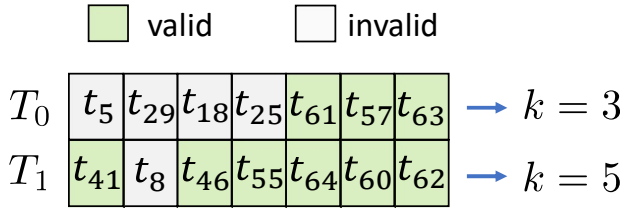


Figure 2.11: The TS-PCSA sketch after inserting all 64 flows.

single packet. By applying the hash function $h(x_i)$, the last bit is used to select the register, whereas the first 1 in the remaining binary string, starting from the LS bit, i.e. rank, identifies the position in the register where the timestamp is updated. The figure shows the step-by-step state when the first 4 flows are added with ranks (2,1,2,3). The final state at time t_{64}, after having inserted all flows, is shown in Fig. 2.11, where we highlight that most of the initial timestamps have been overwritten by the most recent ones, especially in low-rank positions which are more likely to be updated. Assume at this time to query the sketch using an observation window of length $W = t_{64} - t_{32}$. For register T_0, the first invalid timestamp is found at the 4-th register entry ($k = 3$), since $t_{25} < t_{64} - W$, while for register T_1 in the 6-th entry ($k = 5$), since $t_8 < t_{64} - W$. Thus, the average number of continuous blocks of valid timestamps is $n = (3+5)/2 = 4$, that is used to estimate the total number of flows as $2 \times 2^4 = 32$ (by chance, corresponding to the exact value of flows observed in W). For simplicity, in this example, we have not considered the bias correction factor.

Algorithm overhead and complexity

We now discuss the memory overhead related to the timestamp representation and the computational complexity of the ADD() and QUERY() operations.

Memory consumption. The memory footprint of a TS-PCSA sketch using m arrays of b-bits timestamp and capable of counting up to $2^K = n_i$ unique flows per register, is $m \times log_2(\frac{n}{m}) \times b$ bits. This is larger by a factor b of the memory occupancy of a standard PCSA sketch, and (asymptotically) by a factor $b\log_2(\frac{n}{m})$ of the memory occupancy of the ST-HLL sketch (Table 2.1). Choosing the size of b for a binary representation of the timestamp is not trivial. Notably, representing time with infinite precision would require an infinite number of bits. Thus, it is necessary to set a *time resolution*, defined as τ. Now the observation window W can be seen as divided into W/τ time slots, for which at least $\lceil \log_2(W/\tau) \rceil$ bits[6] are required. The timestamps will be wrapped to the maximum integer representation chosen for the timestamp, and we need to be sure to properly compute differences between time slots. Assuming to prune all the invalid timestamps periodically once every αW time (i.e., all timestamps before $t - W$ are reset), with $\alpha > 0$, we need to cover an interval of time $(1+\alpha)W$ with distinct time slots to properly compute the difference of time. We need also an additional bit to tag a timestamp as invalid. Thus, the total number of bits is $b = 1 + \lceil \log_2((1+\alpha)W/\tau) \rceil$.

Time complexity. As regards the ADD() operation, TS-PCSA preserves the same $O(1)$ average time complexity of PCSA. The QUERY() operation is a bit more involved. TS-PCSA requires finding the first invalid timestamp in all registers to average their positions. Thus, querying the TS-PCSA sketch with m registers of K bits has complexity $O(mK)$ due to a linear search in each register. We observe that in PCSA the linear search can be avoided with simple workarounds, like keeping a pointer to the first invalid position within each register. This is practicable because blocks of contiguous 1s cannot fragment once they have built up. In our algorithm the timestamps may become invalid after W time units have elapsed since when they were stored. Thus, a block of contiguous valid timestamps will likely fragment, hindering the use of such a simple technique. This substantial difference represents a limitation of our approach.

2.3.2 Practical optimizations

The previous discussion about TS-PCSA's space and time complexity, has just highlighted that the former is strongly dependent on the timestamp resolution τ. Reducing the resolution can be a memory-saving measure; however, it comes with the downside of introducing rounding errors in time representation. These inaccuracies can impact the precision of determining whether a register entry falls within the observation window(line 5 of the pseudocode in Figure 2.9). Thus, the following question naturally arises: can we leverage the stochastic nature of the PCSA sketch to smooth the error introduced by adopting a low timestamp resolution? In this section, we aim at answering this quest. First, we characterize the impact of the rounding error on estimation accuracy. Then we propose TS-PCSA+, an optimized version of TS-PCSA that allows

[6] We denoted by $\lceil\ \rceil$ the operation of ceil integer rounding

Method	Memory cost [bits]	Query type
HyperLogLog [79]	$m\log_2\log_2(n/m)$	Static
Probabilistic Counting with Stochastic Averaging [68]	$m\log_2(n/m)$	
Sliding HLL [85]	$(b+\log_2(n/m))m\ln(n/m)$	Sliding window
Staggered HyperLogLog [71]	$m\log_2\log_2(n/m)+b$	
TimeStamp-augmented PCSA (TS-PCSA) [72]	$bm\log_2(n/m)$	

Table 2.1: Comparative analysis of the memory requirements of sliding window solutions versus the corresponding baseline methods, which operate on a static observation window. The asymptotic memory cost is expressed in bits, n are the number of observed flows and m the number of register, whereas b is the number of bits used to represent the timestamp.

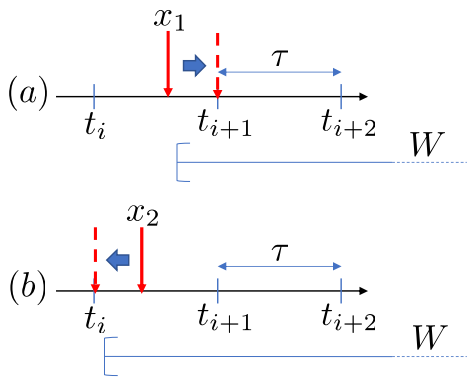


Figure 2.12: Effect of rounding the arrival time on the cardinality estimation: (a) overestimation case, (b) underestimation case.

to reduce the timestamp resolution without affecting the accuracy of the cardinality estimation. Finally, we discuss the possibility of further reducing the memory footprint — which we leave as a future research avenue — by studying the *refresh time* of the register entries.

Timestamp rounding errors

When flow x_i arrives, it is associated with an integer timestamp t_i, which is a multiple of τ. Different ways can be used to round the actual arrival time to the slotted time. We will see later that rounding it to the closest time slot is the strategy that minimizes the counting error, as could be expected intuitively. Nevertheless, the cardinality estimation TS-PCSA still suffers from some temporal rounding errors, highlighted in Fig. 2.12. Indeed, when a flow arrives after the middle of the time slot, as in the case of x_1, the cardinality in W is over-estimated for $< \tau/2$ units of time. On the contrary, when it arrives before the middle of the time slot, as in the case of x_2, the cardinality

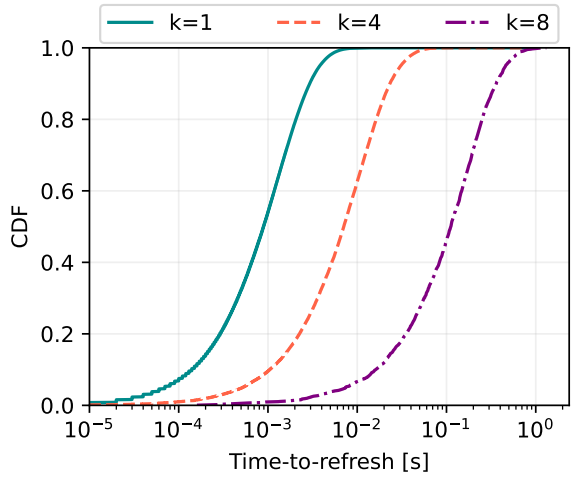


Figure 2.13: Distribution of the timestamp refresh time for three different positions within a register.

in W is under-estimated for $< \tau/2$ units of time. The overall effect, as shown later in our experimental analysis, is that the cardinality estimation appears as a *saw tooth* function around the average.

Register offsetting enables time quantization with few bits

To compensate for the systematic estimation errors due to the timestamp rounding, we propose an enhanced version of the algorithm, denoted as TS-PCSA+. It can be implemented by adding a single line of code to basic TS-PCSA (Fig. 2.9). As reported in the pseudocode, TS-PCSA+ introduces an offset δ_i for each register T_i, computed such that $\delta_0 = -\tau/2$ and $\delta_{m-1} = \tau/2$. Remember that in the PCSA family, individual registers can be seen as independent estimators, each giving contribution $2^{k/m}$ to the final count (see ln. 9). From the discussion in Sec. 2.3.2, it's easy to see that all estimators $2^{k/m}$ follow a saw tooth pattern over time. Now, the rationale is that by anticipating in time half of the estimators and delaying the remaining half — with the average phase offset being null — the phases of the periodic saw tooth estimations combine destructively, averaging out the rounding error. This approach allows to significantly reduce the timestamp size, while preserving accuracy. We will prove its effectiveness in Sec. 2.4.3

Extra optimizations

The amount of bits for each timestamp has been assumed to be constant across all the positions within each register. We wish now to highlight that this is a suboptimal design choice since it is possible to reduce the number of bits, depending on the position within the register.

Scenario	Trace ID	Avg. bitrate	Link rate	Num. packets	Num. flows
CAIDA-2018	equinix-nyc-2018	4.26 Gbps	10 Gbps	37.8M	1.8M
CAIDA-2019	equinix-nyc-2019	4.49 Gbps	10 Gbps	36.7M	1.2M

Table 2.2: Main features of the considered CAIDA traffic traces.

Indeed, consider a toy scenario, with periodic flow arrivals at rate $R = 10^5$ flows/s, and TS-PCSA+ to update $m = 64$ registers. Fig. 2.13 shows the CDF of the refresh time for each position of a register. It can be easily shown that, on average, the timestamp in position k will be refreshed every $(1/R) \times 2^k \times m$, which is coherent with the median value observed in the figure. As an extreme case, looking at the graph the first position (i.e., $k = 1$) is almost surely updated within 0.01 s, suggesting that, by considering any window W larger than this value, storing the timestamp is useless. This suggests that it is possible to reduce the memory footage by never storing the timestamps in such a position. In general, by observing the CDF it is clear that some lower positions within the register can be omitted. At the same time, consider that TS-PCSA+, by construction, does not consider the timestamps within a register above an invalid timestamp (e.g., consider t_8 and t_{20} in Fig. 2.11), thus suggesting that also keeping the full bit representation for such position is useless. In summary, only the timestamp within a "reasonable" central range of positions should be stored to minimize the memory footprint of TS-PCSA+. Furthermore, different time resolutions could be considered depending on the position within the register. The exploration of these optimizations is out of the scope of this chapter, and we defer it to future investigations.

2.4 Numerical evaluation

In this section we evaluate via numerical simulation the proposed algorithms. We show the effectiveness of low-counter variance compensation (Sec. 2.2.4) and of timestamp optimizations (Sec. 2.3.2) on ST-HLL and TS-PCSA, respectively. Then we compare our solutions against existing methods using Internet traffic traces.

2.4.1 Experimental setup

Implementation . We developed ST-HLL and TS-PCSA in Python and publicly released the source code at [102] to make our results reproducible. A crucial point not to bias the experiments is to minimize the number of hash collisions among different flows to the same hash value. If too many collisions happen, it's easy to verify that the effectiveness of HLL and PCSA counting is neutralized, as different flows would contribute to the same bitmap entry. In our implementations, we adopted the `SHA1` hash function and used the first 32 bits of the `SHA1` digest, which are sufficient to practically avoid hash collisions for all the configurations of workload and window sizes we

tested and also provides good insertion speed.

Workloads . We use two kinds of network traffic workloads.

1. *Synthetic.* In this workload, we create ad-hoc traffic to evaluate ST-HLL and TS-PCSA sketches under controlled conditions. Our simulator generates synthetic traffic workloads, where each flow consists of a single packet. Therefore, the traffic stream is composed only of packets belonging to distinct flows, and thus each packet is accounted as a new item by the sketches. Such packets are generated according to a non-stationary Poisson process, whose instantaneous rate is modulated by an arbitrary function $\lambda(t)$. We generated, depending on the scenario, constant, sinusoidal and squared ("clock" wave) arrival rates.

2. *CAIDA traces.* We further validated the performance of all algorithms under realistic network traffic traces, collected from `equinix-nyc` Internet routers and whose main features are summarized in Table 2.2. The traces refer to about 1 minute of traffic and contains about 36 million packets. Without loss of generality, we use the 2-tuple of source IP address and destination IP address as flow key, which results in about 2 million flows.

We tested the algorithms for realistic values of the observation window, e.g., 100 ms – 1 s.

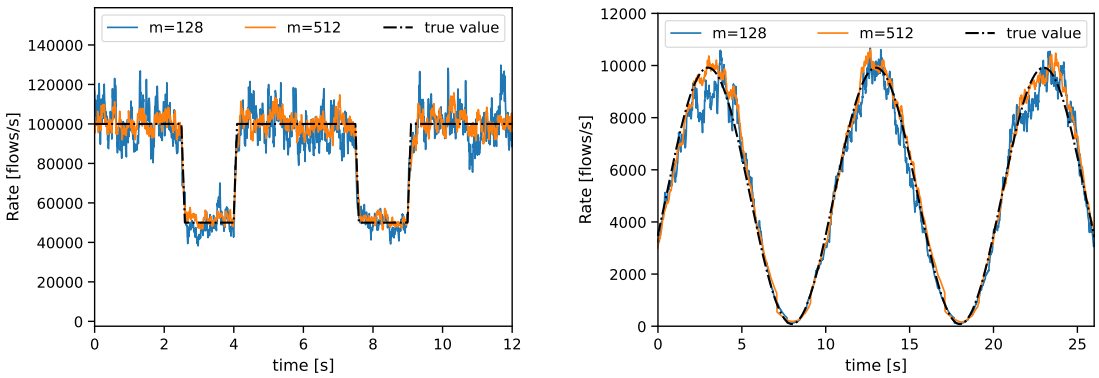
Hardware setup . We run all experiments on Linux machines in the HPC[7] cluster within our institution. Each machine is equipped with two 2.10 GHz Intel Xeon Scalable Processor Gold 6130 CPUs with 16 cores and 384 GB DDR4ECC RAM.

Comparison algorithms. As a term of comparison, in the following we confront our algorithms against Sliding HLL (denoted as "W-HLL" in the following), which is the state-of-the-art solution proposed in [85] and discussed in Sec. 3.4. We implemented W-HLL according to the guidelines provided in the original paper [85], and applied no optimizations to the timestamp representation which stored as *float32* variables.

2.4.2 Staggered HyperLogLog performance

We have assessed the performance of ST-HLL using both *(i)* synthetic traffic traces suitably crafted so as to test how our scheme responds to significant traffic fluctuations (see Sec. 2.4.2), and *(ii)* real world traffic traces, which we used to quantify the effectiveness of our approach also compared with the sliding-window HLL solution proposed in [85] (see Sec. 2.4.2).

[7] Academic Computing Center at Politecnico di Torino - `http://hpc.polito.it`



(a) SQU traffic, $W = 0.1$ s.

(b) SIN traffic, $W = 1$ s.

Figure 2.14: Synthetic traffic scenarios with different depths W of sliding window.

Synthetic traffic streams

In order to assess the effectiveness of ST-HLL in tracking variations in the flow arrival rate, initially we used ad-hoc traffic scenarios where we vary the speed of rate variation. We tested the following two scenarios:

- Squared wave traffic (SQU, Fig. 2.14a), in which $\lambda(t)$ varies between 50,000 and 100,000 flows/s with a period of 5 seconds and a duty cycle equal to 70%. Here, we have used a relatively short window $W = 0.1$ s to specifically assess the ability of ST-HLL to promptly follow abrupt traffic fluctuations.

- Sine wave traffic (SIN, Fig. 2.14b), in which $\lambda(t)$ varies according to a sinusoidal function between 0 and 10,000 flows/s, with a period equal to 10 s - we here used a longer window $W = 1$ s.

The plots shown in the figure are obtained by periodically reading the content of the ST-HLL counter (hence by performing the query procedure which computes the estimated rate sample), with a sampling time multiple of the register's reset time slot τ. We removed the transient phase from our numerical results.

In both figures, we compare the results obtained by two ST-HLL settings ($m = 128$ and $m = 512$) with the *true value* obtained by filtering the nominal arrival rate with a moving average window of duration W - in essence, by comparing with an ideal sliding window counter of depth W. Note that this comparison is somewhat unfair for us, as our ST-HLL filter does not implement a "rectangular" sliding window of depth W, but mimics a "triangular" window, as discussed in Sec. 2.2.4.

From both figures, we remark that ST-HLL remains very close to the true cardinality count. Its gap with the true count remains most of the time within $1.04/\sqrt{m}$, which is in line with the theoretical error bounds derived for HLL. For $m = 512$ this value is 4.59. The estimate remains close to the true value even when the rate $\lambda(t)$ gets close to

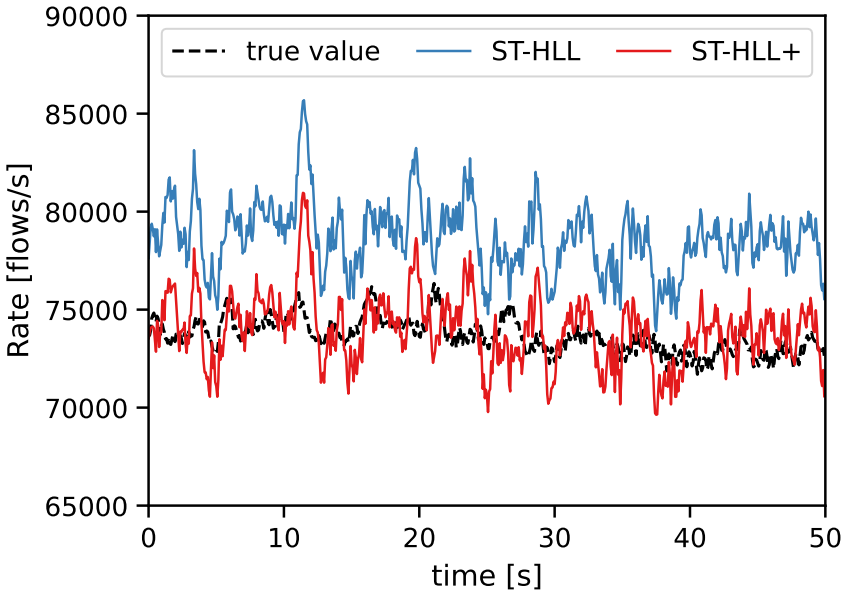


Figure 2.15: Cancelling the over-counting effect of ST-HLL when duplicated items are present. CAIDA 2019, $W = 1$ s, $m = 2048$.

zero. Notice that in this region most of the registers of ST-HLL are empty, so as in [79] we resort to linear counting, for which the accuracy guarantees might not be the same of HLL.

Real Internet traces

To provide more realistic results, we analyze the algorithm's performance over real Internet traces [103], [104]. We considered two traces, CAIDA-2018 and CAIDA-2019, collected in a backbone router link at Equinix-New York and whose main features are reported in Table 2.2.

A major motivation behind such experiments consists in assessing whether our proposed estimation is robust also in the case of real traces. Indeed, unlike the synthetic traffic used in the previous section, real world traces may not anymore closely follow our baseline modeling assumption stated in Sec. 2.2.2, i.e., that *the number of new items recorded by each register is proportional to the size of its measurement period*. In practice, in real world traffic, each flow may in fact appear more than once inside the sliding window, thus leading to the presence of duplicate items, and the frequency and burstiness of such duplicate items largely varies across different flows.

It is intuitive to see that duplicate elements have a much greater impact on short measurement windows than on long ones. For an extreme example, a single persistently recurring flow would always be accounted as "+1" on any measurement window

size. Hence, its impact on the rate estimation would be significantly greater for a short window rather than for a long one[8].

Indeed, the above intuition was confirmed by our experiments on the CAIDA traces. Fig. 2.15 in fact shows a 6-7% bias in the rate estimation obtained by a 1024 register ST-HLL. However, the above discussion also suggests that the very simple heuristic introduced in Sec. 2.2.4 for a different purpose, namely reduce the impact of the more noisy low-index registers, can also effectively mitigate the impact of duplicate flows. We recall that such heuristic trivially consists in discarding a relatively small fraction (1/8 in our experiments) of low-indexed counters when computing the rate estimation. And since low-index registers are those which more severely affect the rate estimation, we expect a significant improvement in the estimation itself. This is experimentally confirmed in Fig. 2.15 by the dramatic increase in the accuracy of the ST-HLL+ plot with respect to the baseline ST-HLL counter.

To gather further quantitative insights on the performance of ST-HLL and of the ST-HLL+ heuristic, we ran an extensive set of results for both CAIDA traces and for a variety of different ST-HLL parameter settings and sliding windows duration. Results are shown in Fig. 2.16, where we confront ST-HLL(/+) against W-HLL. For a fair comparison, results are shown as a function of a fixed overall memory budget. Remember that W-HLL is less memory-efficient than ST-HLL, by a factor $(b + \log_2(n/m)) \times (\log_2 \log_2(n/m))^{-1}$ (Table 2.1), which depends on the number of registers. In practice, we implemented HLL (and so ST-HLL) with a number of registers which is power of two, as per [79] and aligned the register sizes to 1 byte. Therefore, as a "rule of thumb" approximation, Figure 2.16 should be read as follows. Let's make the conservative assumption that for W-HLL $b = 0$, i.e., ignore the cost of storing timestamps in the LPFMs. Then, the above defined ratio is always above 8 for n/m bigger than 40[9]. Thus, the memory demands of W-HLL is (at least) 8× larger than ST-HLL(/+) and, for a fixed memory budget (x-axis), we can compare the two sketches by scaling the number of registers by a factor 8 — e.g., with 4096 bytes available ST-HLL uses 4096 registers and W-HLL 512 registers. Notice that the actual gain in memory efficiency of ST-HLL vs W-HLL is even higher, since storing timestamps in fact occupies memory.

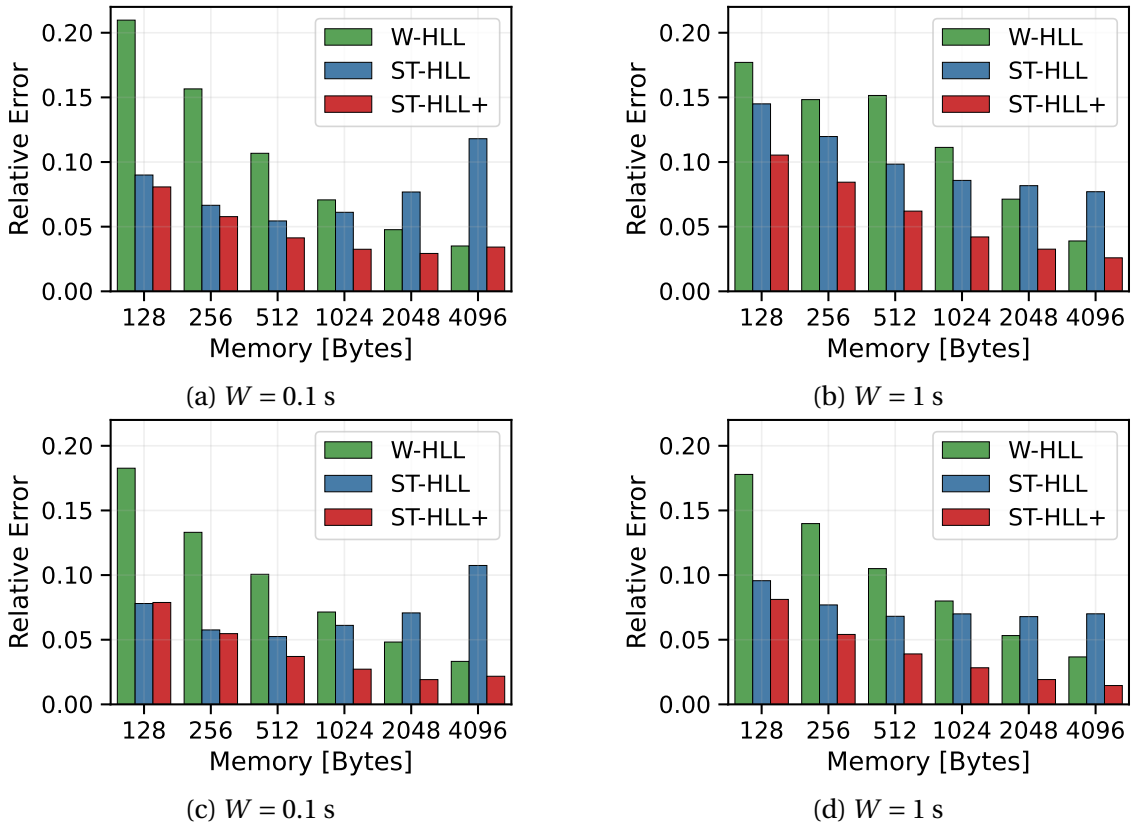
Overall, as shown in Figure 2.16, with less than 1 KB of memory available, both ST-HLL and ST-HLL+ outperform W-HLL in terms of accuracy. Moreover, in all considered traffic scenarios, W-HLL needs at least 4× more memory compared to ST-HLL+

[8]This is a direct consequence of our definition for the local rate estimator (2.4): if we add such extra persistent flow to the remaining n_i measured by the i-th register, i.e., we alter the original estimation $\hat{\lambda}_i$ as:

$$\hat{\lambda}'_i = \frac{(n_i + 1)m^2}{2Wi} = \hat{\lambda}_i + \frac{m^2}{2Wi},$$

then its relative impact would be significantly greater for small values of i.

[9]in our experiments there are around $n = 75$k flows in the largest observation window of 1 s (Figure 2.15), leading to $n/m \approx 18$ when using 4096 registers, and $n/m \leq 18$ otherwise.



(a) $W = 0.1$ s (b) $W = 1$ s

(c) $W = 0.1$ s (d) $W = 1$ s

Figure 2.16: Comparative evaluation of ST-HLL with state-of-the-art over Internet traffic traces. Figs. (a)-(b) refer to CAIDA-2018 traffic scenario, whereas Figs. (c)-(d) to CAIDA-2019.

to provide relative estimation error guarantees as low as 5%.

2.4.3 Effectiveness of TS-PCSA optimizations

We conducted a set of experiments on synthetic and real-world traffic workloads to evaluate TS-PCSA. In this section, we (1) demonstrate the effectiveness of our technique in compensating the overestimation and underestimation errors introduced by low-resolution timestamps, and (2) empirically quantify the memory-accuracy trade-off under real workloads.

First, we provide evidence about how our technique based on register offsets averages out overestimation and underestimation errors introduced by time quantization. We generate packet (i.e., equivalently flows as per Sec. 4.8.1) arrivals according to a deterministic generation process with constant rate $\lambda(t) = \lambda'$. This simple traffic is helpful to clearly highlight the effect on estimation accuracy of quantizing packet arrival times to coarse-grained bins.

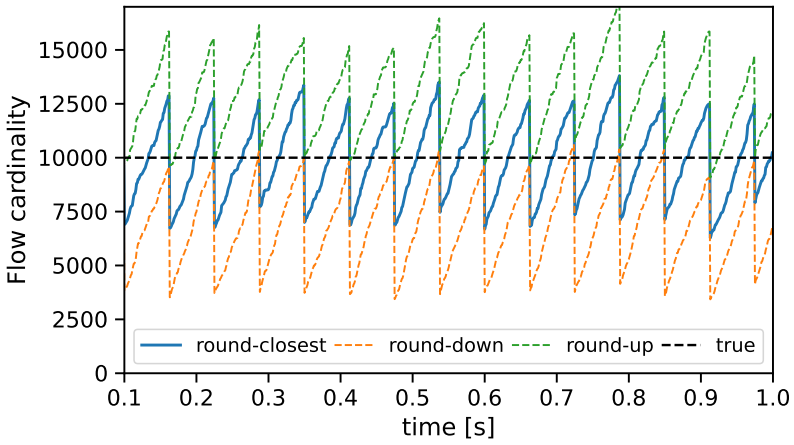


Figure 2.17: Rounding effect in TS-PCSA with $b = 5$ bits to represent the timestamp.

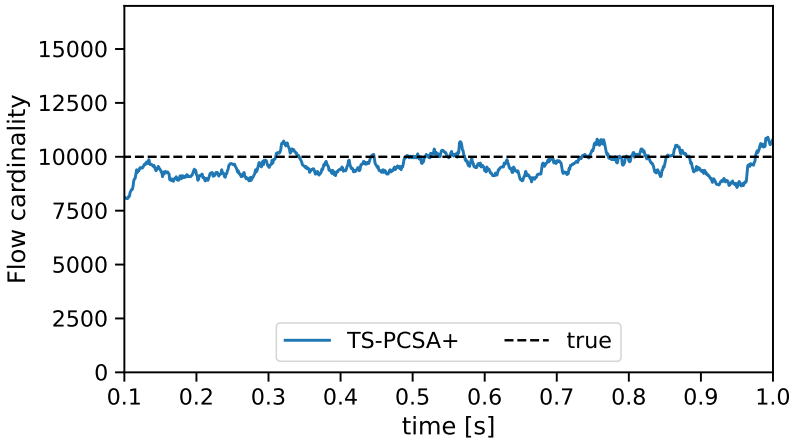


Figure 2.18: TS-PCSA+ algorithm smooths the saw tooth behavior.

In the first experiment, we run on this synthetic workload the baseline version of our algorithm, without the optimizations and register offsetting. Also, we compare round-up, round-down, and round-closest rounding strategies, where packet arrival times are set to the past, next, and closest represented timestamp, respectively. In this experiment, we configured the sketch with 256 registers, the time resolution τ was set to 0.0625 s. Figure 2.17 shows the results. In all scenarios, we observe a saw tooth pattern in the flow cardinality estimate, with 16 peaks in a 1 s interval (the initial transient has been removed), coherently with what was discussed in Sec. 2.3.2. Adopting the round-up strategy, we only suffer overestimation errors. In fact, the minima correspond almost exactly to the true value. The contrary holds for round-up.

Next, we introduce the optimizations presented in Sec. 2.3.2. Figure 2.18 shows what happens when the same workload undergoes TS-PCSA+. We notice the effectiveness of our approach, as a relative time offset between registers of about $\tau/m \simeq$

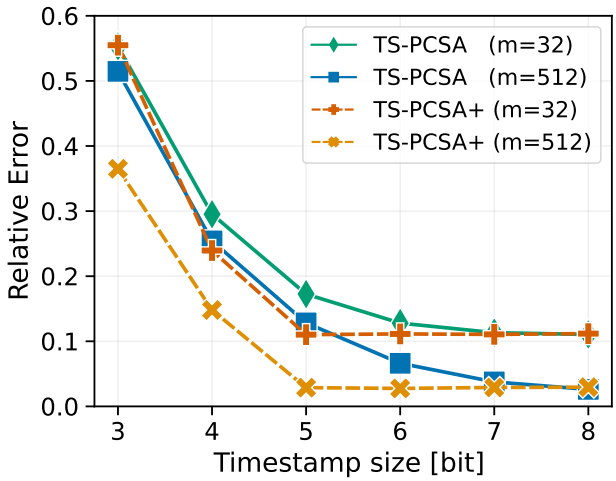


Figure 2.19: Trade-off between accuracy and number of bits used for time quantization for TS-PCSA and TS-PCSA+.

0.24 ms significantly reduces the peaks in the estimation. This is because the overestimation and underestimation errors introduced by quantization are averaged out when combining registers with different time offsets.

How many bits can TS-PCSA+ save?

The next step is to analyze how our algorithm behaves under the realistic traffic workload described in Sec. 2.4.1. As a performance metric, we measure the average estimation error, relative to the true value of the cardinality. We first try to understand how much we can gain in practical scenarios comparing TS-PCSA with its optimized version TS-PCSA+. We want to quantify the gain in terms of how many bits per timestamp we can save, without sacrificing accuracy. Fig. 2.19 shows that for all configurations in the number of registers, TS-PCSA+ requires as much as 38% less memory with respect to TS-PCSA to achieve a relative error below 10%. Performance stabilize at 5 bits/timestamp. We verified that even if timestamps were represented using python's *float32*, the relative error converges close to the same value. This means that our simple yet powerful offset technique closely approaches a system with "ideal" timestamp resolution.

2.4.4 Comparison between the two sketches

We compare TS-PCSA to related continuous-time probabilistic counting solutions, such as W-HLL and ST-HLL that were introduced in Sec. 2.1.2 and Sec. 2.2.4, respectively. Since all these sketches share the same structure based on registers, we compare them as a function of the number of registers. Figure 2.20 reports the results of our experiments on the CAIDA traces with an observation window W of 100 ms.

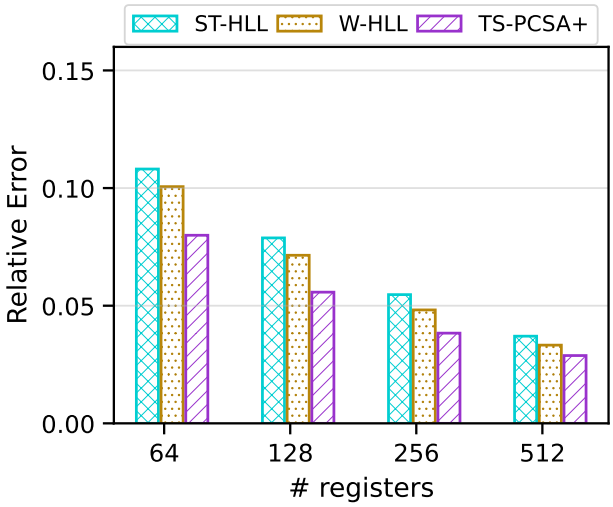


Figure 2.20: Comparison between timestamp-augmented and timestamp-free algorithms over Internet traffic traces Table 2.2.

In respect to timestamp-augmented solutions, TS-PCSA+ outperforms W-HLL by reducing the relative error up to 25%, especially when few registers are available. The reason is that W-HLL bases its estimations solely on the most recent maximum rank, whereas TS-PCSA+ considers an entire block of contiguous ranks not older than W time units. Notably, TS-PCSA+ and W-HLL can be deployed at approximately the same memory cost. The memory cost of TS-PCSA+ is $b \times m\log_2(n/m)$, which can be compared with the cost of W-HLL (in Table 2.1) rewritten as $(b\ln(2) + ln(n/m)) \times m\log_2(n/m)$. Since n in the range between 10-20k and 50-80k for $W = 1$ s and $W = 0.1$ s, respectively, $\ln(n/m) \leq b$ in practice. Moreover, thanks to the proposed optimizations, TS-PCSA+ gives good results with few bits, e.g., $b = 5$ bits/timestamp configuration, whereas W-HLL works with $b = 32$ as per Sec. 2.4.1 and no other timestamp compression techniques have been investigated for it. Overall, this shows that our algorithm improves accuracy upon existing timestamp-augmented methods for the same memory cost.

When compared to timestamp-free sketches, TS-PCSA+ also gives better accuracy than ST-HLL, for the same number of registers. In this case, however, ST-HLL is at least 8× more lightweight in terms of required memory in respect to TS-PCSA and W-HLL, as commented for Figure 2.16.

2.5 Related work

Other streaming algorithms designed for a sliding window model have been recently proposed. Similarly to Sliding HLL, *SWAMP* [83], *WCSS* [82], *Memento* [105] and *Sequential zeroing* [106] are all based on the removal of outdated information

from their data structures, so that only the most recent items contribute to the estimation. However, these works only support event-based windows, usually defined in terms of number of packets, and not time-interval queries. SWAMP can answer set-membership, frequency, cardinality and entropy queries with a single data structure, which is a circular buffer to track fingerprints of recent items, plus an auxiliary counting hash table to store their frequencies. Being designed with generality in mind, SWAMP is memory demanding. Moreover, the use of a *TinyTable* [107] makes complex its implementation on programmable switches. Ivkin et al. [87] devised an elegant sketch-based framework that allows to specify the time frame of interest as a query parameter. Similar to [86], it offers an integrated solution for various measurement types in a single structure, however it needs at least 56 MB of memory to accurately detect a DDoS attack. Our approach limits its scope only to a single task (cardinality estimation), but with much smaller memory footprint. A different class of algorithms [108], [109] down weights the relative importance of aged items with respect to recent ones. *AdaSketches* [108] is a time-aware sketch that emphasizes newly inserted items with a function that monotonically increases with the timestamp of arrival, so providing higher query accuracy to recent events. A customization of AdaSketches tailored to commodity switches can be found in [110], but, aiming at frequency estimation, it is orthogonal to our work.

2.6 Discussion

In this chapter, we have addressed the problem of estimating, in near-continuous time, flow arrival rates for real-time traffic streams. We have focused on two sketch-based data structures, HLL and PCSA, and extended their applicability beyond the traditional time binning approach.

As a first contribution, we have proposed Staggered HyperLogLog (ST-HLL), a register-based probabilistic data structure which does not introduce any memory overhead with respect to vanilla HLL. Thanks to a proper periodic resets of the registers and an equalization of the rate estimators of each register, we showed that ST-HLL supports continuous-time queries over arbitrary observation windows, and quickly captures rate variations. To the best of our knowledge, ST-HLL is the first timestamp-free sketch data structure for cardinality estimation, and can be implemented with the same complexity of HLL using timeout mechanisms available in commercial switches.

Subsequently, we proposed a novel timestamp-augmented sketch based on the PCSA probabilistic data structure and tested it over real-world Internet traffic. Thanks to the simple yet effective strategy to associate a constant temporal offset to the sketch registers, our algorithm remains as lightweight as previous techniques based on timestamp, however, it is up to 25% more accurate.

For both techniques, we validated the proposed approach throughout extensive

simulations, using both synthetic and real traffic traces, and showed that our solutions achieves a better accuracy with respect to other state-of-the-art solutions, given the same memory footprint. For instance, ST-HLL dramatically reduces the relative estimation error by more than 50% with respect to the state-of-the-art W-HLL, with only 128B of memory footprint.

Open issues and future research avenues

We conclude by outlining a set of issues that we left open and that we believe are worth investigating as part of future lines of work.

1) Generalization of register staggering. Extending a vanilla HLL to estimate rates has been achieved thanks to staggering with respect to time its internal structure based on registers. We can see this mechanism as a general solution, which in principle could be applied to other sketches that share the same register-based structure [80], [111], paving the way to the design of a new set of streaming data structures tailored for real-time traffic monitoring.

2) Adaptive register size. We highlighted the opportunity for further compressing the TS-PCSA structure, by leveraging the observations that different positions in the TS-PCSA registers are updated with different frequencies (Sec. 2.3.2). This suggests to privilege "important" timestamps over the others as a possible extra optimization. Since some timestamps might never be updated, their entry size in the register could be reduced, thus adapting the register size depending on the recent frequencies of position updates. For example, different time resolutions could be considered depending on the position within the register.

Chapter 3

Collaborative Flow Size Estimation with Sketch Disaggregation

Part of the work presented in this chapter has been published in:

- A. Cornacchia, G. Sviridov, P. Giaccone, and A. Bianco, "A Traffic-Aware Perspective on Network Disaggregated Sketches", in *2021 19th Mediterranean Communication and Computer Networking Conference (MedComNet)*, 2021.

Modern telecommunications networks are characterized by a high degree of dynamicity in traffic patterns. A continuous rolling out of new network-based applications and sudden changes to existing ones have been shown to have a catastrophic impact on the performance of seemingly unrelated services, and on the whole network infrastructure [113]. Due to this unforeseeable behavior, network monitoring has become one of the most important aspects of modern network management. Complex network monitoring mechanisms have been developed to try to predict and counteract those unexpected behaviors. Among those mechanisms, sophisticated centralized network monitoring schemes enabled by breakthrough technologies such as SDN have found large popularity and applicability. Yet, centralized solutions lack in scalability, as continuously conveying monitoring information from all switches becomes unbearable for large networks.

As a consequence, traffic monitoring distributed across the switches has become the dominant best practice for network monitoring.

Still, measurements are challenging to implement due to the huge number of concurrent flows and the ever-increasing link rates, which force network devices to complete per-packet operations at nanosecond time scales. This implies resorting to expensive SRAM as the only viable solution to store measurement data. Due to the scarce amount of such dedicated memory, it is prohibitively expensive to keep exact per-flow information locally at each switch. Sampling-based techniques such as NetFlow [18] were traditionally employed to limit resource overhead. However, they

are not capable of providing sufficient accuracy and flow coverage, if not adopting extremely high sampling rates [37], [114], [115].

Due to the aforementioned constraints, sketch-based algorithms have found vast applicability in the field of network monitoring. They permit to condense the target flow metrics in compact probabilistic data structures stored inside the switch. This information is then periodically fetched by a central entity and aggregated into a unique, network-wide approximate result. The accuracy of measurements is proportional to the amount of dedicated memory and inversely proportional to the amount of traffic that traverses the switches. While in modern networks the volume of concurrent flows per second keeps growing, the amount of SRAM inside single switches remains constant, and it has to be shared among concurrent measurement services and network functions. Under such a scenario, to increase the monitoring accuracy it is necessary to reduce the number of flows stored inside single sketches. A natural option in this case is to increase the sketch fetching frequency, in order to have fewer flows being condensed in a single sketch within each measurement interval. Yet such an approach has its own limits which are dictated by the resulting communication overhead and, most importantly, by the underlying hardware capabilities of single switches [116], [117].

Recently, in [118] the authors have proposed DISCO, a system of disaggregated sketches able to address the aforementioned issues. At its heart, DISCO employs multiple sketch fragments scattered around the network which are updated by the flows that traverse them. This enables the possibility of reducing the sketch fetching frequency of each switch without loosing accuracy — or to provide higher accuracy with the same amount of memory — with respect to traditional approaches. Yet DISCO does not consider aspects related to traffic distribution and locality that are critical for improving the overall system performance.

Given the previous discussion, in this chapter we analyze the impact of traffic distribution on the performance of disaggregated sketches. In Sec. 3.1.1 we provide the background related to probabilistic data structures for network monitoring. In Sec. 3.2 we discuss disaggregated network-wide sketches and motivate our contribution, which is numerically evaluated in Sec. 3.3. We show that, even in simple scenarios, blindly updating all the fragments crossed by a flow on its path leads to measurement performance degradation. Moreover, we show that, by just selecting a subset of fragments to update it is possible to improve the aggregate monitoring accuracy, and we provide hints on the existence of such an optimal subset.

The interested reader can find in Sec. 3.4 the relevant related work. Finally, in Sec. 3.5, we discuss the main findings and outline future research directions for disaggregated sketches, part of the which have been already followed-up by a recent work in [11].

3.1 Preliminaries

We review the basic concepts of sketch-based algorithms for network monitoring and introduce the concept of disaggregated sketches.

3.1.1 Atomic sketches

Sketches are probabilistic data structures for stream processing, able to estimate flow statistics using a fixed and small number of entries, relatively to the number of processed flows. To distinguish them from the recently proposed disaggregated sketches [118], we refer to them as *atomic sketches*, since not distributed across the network. While a myriad of sketch-based algorithms has been proposed, they all are based on primary components: one or more array of counters, a procedure for their *update* upon new packet arrival and a method to read counters and answer *queries*.

3.1.2 Sketch dimensioning

We take the popular Count-Min Sketch (CMS) as an example [111]. When measuring the network traffic, its counters keep track of flows' occurrences in a packet stream. The available memory budget is organized into d rows of w counters each. The *update* procedure consists on computing d pairwise independent hash functions over the flow identifier (e.g., 5-tuple) in order to associate a flow to one counter per row. Then, the selected counters are incremented by one, so as to add the contribution of the new measured packet to the current flow size. Since multiple flows may collide onto the same counter, the *query* operation returns the minimum among d values as its estimate. Generally, sketch-based algorithms come with provable theoretical guarantees and allow us to tune the parameters to trade between estimation accuracy and memory consumption. Indeed, in a CMS it is possible to derive the required number of rows d and of counters w as function of the error magnitude ϵ and the error probability δ. If $w = \lceil e/\epsilon \rceil$ and $d = \lceil -\log \delta \rceil$, the estimated size $\hat{x}$ of any flow size x is proven [111] to be within the bound:

$$\hat{x} \leq x + \epsilon \|\mathbf{x}\|_1 \tag{3.1}$$

with probability greater than $1-\delta$. Here, x is the true value of the counter, while $\|\mathbf{x}\|_1$ is the ℓ_1-norm of the flow size vector, and it amounts to the total number of packets counted in the whole sketch. A higher number of independent hash functions d are needed in order to reduce the error probability and statistically corresponds to relying upon more estimators, whereas increasing w reduces the error magnitude.

Network monitoring with sketches and their limits

Atomic sketch algorithms have been widely used to cope with memory scarcity while employing switch-local traffic monitoring. Traditionally, sketches are deployed

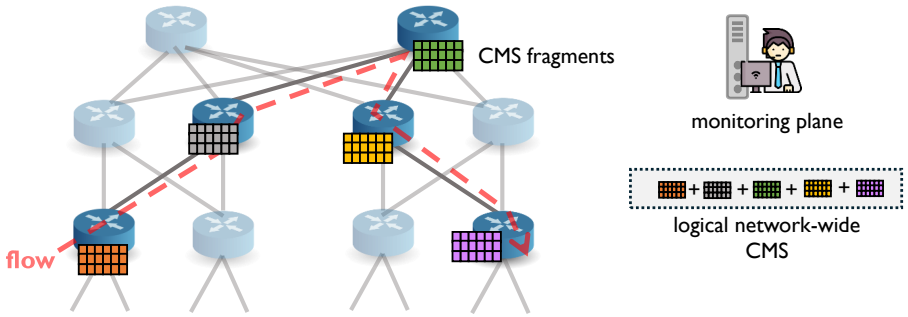


Figure 3.1: Disaggregated sketch architecture for a Count-Min Sketch (CMS).

on multiple distributed monitors, orchestrated by some controller. Individual monitors inside the network periodically convey their local information to the controller. The controller, aggregates telemetry data from multiple switches and derives the final measurements network-wide [35], [37], [115], [119]. A noteworthy application of sketches is the so-called *heavy hitters detection*, which consists into the detection of the network flows that utilize a large amount of bandwidth. However, gathering fine-grained per-flow statistics is highly impractical, if not possible at all. Consequently, CMS has found applicability for this kind of tasks, as it permits discriminating mice and elephant flows up to a given error, while still utilizing small amount of resources.

Yet, as the amount of fast SRAM memory in commodity switches remains constant, the ever-growing number of concurrent flows pushes this kind of approach to its limits, thus making it difficult to maintain an acceptable level of accuracy. In addition, multiple measurement tasks usually execute concurrently and share the available memory. For example, operators may run a heavy-hitter detection task to make intelligent routing decisions and, at the same time, also run some anomaly detection tasks to discover the presence of port-scanners or DDoS attacks. These tasks run in parallel and need a dedicated sketch instance [115], [116]. While increasing the reporting frequency may counteract the previously discussed limitations, it has its own drawbacks as it is constrained by the underlying hardware and the generated network overhead.

3.1.3 Disaggregated sketches

A radically different approach was recently proposed in DISCO [118]. The authors suggested *disaggregating* a large sketch into multiple smaller sketch *fragments* (i.e., a subset of rows and columns) and distribute them across monitor points in the network, as shown in Figure 3.1. Then, a single sketch is logically rebuilt by assembling fragments encountered along network paths. Hence, different paths correspond to different logical sketches. For the case of CMS this approach leads to a very simple implementation by which the current minimum across the fragments is piggybacked in the packet headers and the final estimate is obtained by analyzing the packet header

at the last-hop. This approach leads to an efficient, yet accurate, heavy-hitter detection scheme while keeping the approach realistic enough to be implemented in real scenarios.

Disaggregated sketches support the computation of switch-local heavy hitters, as well as global heavy hitters, which indicate a flow that is heavy hitter after summing the contributions from many paths. For the former, the last hop switch is responsible for the flow size estimation, and only when a heavy hitter is detected the switch informs the remote monitoring plane. For the latter, the remote monitoring plane needs to aggregate the estimation from all last-hop switches for the flow and derive the global flow size, and eventually can identify heavy hitters.

3.2 Traffic-aware disaggregated sketches

While DISCO is capable of outperforming atomic sketches, it still adopts a static fragment update policy, meaning that flows are always counted at all fragments along their path.

We argue that, for a given flow, updating all available fragments is often unnecessary and, in some cases, even harmful. Indeed, DISCO update policy has the side effect of generating, otherwise avoidable, collisions (which we will refer to as *counter pollution*) inside single fragments, ultimately degrading heavy-hitter detection accuracy. On the other hand, counting each flow only in one fragment would obviously reduce collisions, but would also impair the accuracy whenever a collision occurs. We show that there exists a trade-off on the number of fragment updates, which balance tolerance to collisions with pollution on sketch counters.

The effects of our observations are exacerbated by non-uniform traffic patterns in the network. In the specific scenario of data center networks, the traffic workload typically presents a non-homogeneous distribution across different network devices [120], with the majority of flows being rack-local. This implies that different switches observe a different load in terms of packets and flows per second. Now blindly updating all the fragments of a sketch may lead to "overload" fragments (e.g., in top-of-rack switches) and "underutilized" fragments (e.g., in spine switches) and this fact degrades the accuracy of the overall network-wide sketch scheme.

To formalize the considered monitoring scenario, we assume to have a set F of active flows and a network-wide sketch denoted by a set S of fragments distributed in the network. For simplicity, coherently with DISCO, we assume one sketch row (i.e., one hash function) in each switch, but our results can be qualitatively extended to multiple rows per switches. Consider now a flow $f_i \in F$ that traverses a subset $S_i \subseteq S$ fragments along its path. Let k_i^{opt} be the optimal number of fragments to update for f_i, which maximizes the average monitoring accuracy for all flows in F. We will show that it may hold that $k_i^{opt} < |S_i|$ for some $f_i \in F$, i.e., the flow should be not counted in all the fragments. Our goal is to highlight the presence of this phenomenon

and to quantify to which extent it may affect the network measurement performance. Indeed, devising a policy capable of selecting the optimal amount of fragments to update and their location across individual flows' path is still to be investigated.

3.3 Numerical evaluation

In this section, we discuss preliminary results obtained through numerical simulations on the testbed topology of Fig. 3.2. Our goal is to show that the accuracy of disaggregated sketches under different workloads depends on the fragment update strategy and traffic patterns.

3.3.1 Simulation scenario

For our analysis, we employ a discrete-event packet-based simulator built on top of OMNeT++ [121]. For the sake of clarity, we employ a simple bus topology depicted in Fig. 3.2. This choice allows us to effortlessly create a scenario highlighting the previously discussed phenomenon of traffic interference. We assume a set of traffic flows equal to $F = \cup_{k=0}^{7} F_k^{(v)} \cup F^{(h)}$, i.e., comprising eight sets of "vertical" flows and one set of "horizontal" flows, as shown in Fig. 3.2. The horizontal flows in $F^{(h)}$, depicted in blue, updates all or a subset of all the fragments $S = \{s_0, .., s_7\}$ available in the topology. All vertical flows in $F_k^{(v)}$, depicted in red, can exploit only one single monitoring point s_k (i.e., the only switch traversed by them). For all flows in F, we use a heavy-tailed Pareto

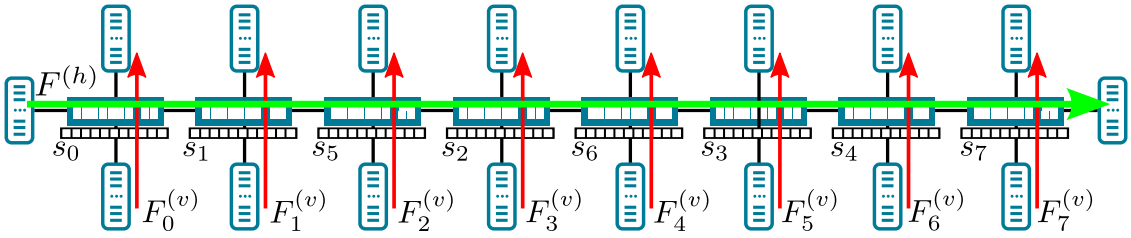


Figure 3.2: Bus topology employed for the simulations.

flow length distribution with shape parameter $\alpha = 1.2$ and mean value 10 packets, to approximate a realistic data center workload [122]. Horizontal and vertical flows are generated according to Poisson processes with $\lambda_h = 100 \times \lambda_v$, respectively. Note that the results are completely invariant with respect to the absolute value of λ_v and on the link capacity and propagation delay, which are assumed homogeneous in the whole network. All switches are equipped with sketch fragments consisting of a single row (i.e., $d = 1$) and $w = 2000$ counters.

As in prior work [35], [123], we evaluate the performances on the basis of the Average Relative Error (ARE) metric, which is defined as the relative error of the estimated

flow size $\hat{x}(f_i)$ with respect to the real flow size $x(f_i)$, averaged over all measured flows:

$$\text{ARE} = \frac{1}{|F|} \sum_{i=1}^{|F|} \frac{\hat{x}(f_i) - x(f_i)}{x(f_i)} \tag{3.2}$$

Note that in the Count-Min sketch, by construction, $\hat{x}(f_i) \geq x(f_i)$. The relative errors are always computed at flow termination. The choice of ARE with respect to more application-specific metrics, (e.g., false positive rate for heavy-hitter detection), is due to its versatility. Indeed, ARE is agnostic to the definition of an elephant flow and general enough to be amenable for several applications.

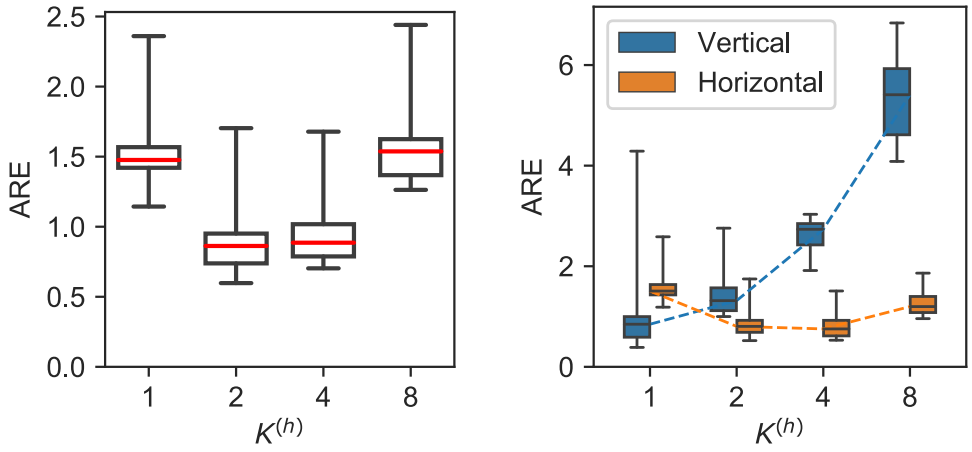
3.3.2 Simulation results

To highlight the impact of interfering traffic on the monitoring performance of sketch fragments we consider varying the number of fragments to update $K^{(h)}$ for the horizontal flow. This implies that, out of 8 fragments present along the path, only $K^{(h)}$ will be chosen *randomly* for each $f_i \in F^{(h)}$. For vertical flows instead, we fix $K^{(v)} = 1$ as those flows traverse only one fragment.

Fig. 3.3 show the ARE in terms of minimum, maximum, 25/50/75-th percentiles, and summarizes the main findings of our simulations. The bars depicted in Fig. 3.3a represent the overall ARE of aggregate flows, measured for varying $K^{(h)}$, while Fig. 3.3b presents a more detailed breakdown to distinguish between horizontal and vertical flows. While transitioning from $K^{(h)} = 1$ to $K^{(h)} = 2$, the monitoring accuracy significantly improves, further increasing $K^{(h)}$ to 4 only leads to a slight reduction in the ARE for horizontal flows. Interestingly enough, setting $K^{(h)} = 8$ increases the error for both groups of flows. Thus, in contrast with intuition, the behavior of ARE in function of $K^{(h)}$ does not appear to be monotonic, suggesting us that there exists an optimal value for $K^{(h)}$ in function of the topology and traffic patterns, as discussed in Sec. 3.2. Indeed, this behavior is due to the fact that, for $K^{(h)} = 8$, all horizontal flows are counted on all fragments present in the topology, thus increasing the pollution on all fragments, ultimately leading to reduced accuracy. On the contrary, the monitoring performance of vertical flows drops considerably while increasing $K^{(h)}$, with their ARE increasing by a factor of 4.85 when moving from $K^{(h)} = 1$ to $K^{(h)} = 8$. Yet, when taking into account the overall monitoring accuracy that includes both vertical and horizontal flows, it can be seen that employing $K^{(h)} = 2$ leads to better aggregate monitoring accuracy across all flows.

3.4 Related work

Sketch algorithms have been employed for various measurement tasks, such as top-k flow identification, traffic entropy, flow size distribution and cardinality estimation, as well as heavy-changer and heavy-hitter detection [117]. The diversity among



(a) Overall

(b) Horizontal and vertical flows

Figure 3.3: Average Relative Error evaluated for different number of fragment updates.

these tasks required generic data structures to support them concurrently at low complexity, together with efficient communication with the central controller. OpenSketch [115] proposed a software-defined measurement architecture, where a single unified hardware pipeline can be programmed to support a wide range of measurement tasks based on sketches. SCREAM [116] improves the accuracy of distributed sketch-based monitoring, by optimizing resource allocation across multiple switches and tasks, based on user requirements. ElasticSketch [35] combines a hash table with a CMS to separate elephant from mice flows and mitigate their collisions, thereby adapting to skewed flow size distributions. Additionally, they first propose a sketch compression technique to reduce communication overhead with the central aggregator. FCMSketch [124] is an elegant tree-like multi-stage counter scheme, which enables the implementation of sketches that support generic measurement tasks directly on PISA switches.

3.5 Discussion

In this chapter, we highlighted the effect of traffic patterns on the performance of disaggregated sketches. In particular, we show that blind fragment update policies which force a flow to update all the fragments along its path may not necessary lead to the best overall monitoring performance. Through numerical simulation and under a simplistic testbed scenario, we show that there should exist an optimal fragment update policy that is capable of selecting a subset of fragments to update among those available on the path of individual flows. Such policy must operate on the network traffic pattern and take into account the total traffic traversing individual fragments.

Further investigation of this behavior, alongside more complex experimental scenarios, is left as future work.

Chapter 4

Sketching Microservices Observability in Programmable NICs

Part of the work presented in this chapter has been published in:

- A. Cornacchia, T. A. Benson, M. Bilal, and M. Canini, "MicroView: Cloud-Native Observability with Temporal Precision", in *Proceedings of the CoNEXT Student Workshop*, ACM, 2023.

Cloud-native applications consist of thousands of single-concern, loosely-coupled microservices running on containerized platforms with many companies such as Uber, Netflix, and Twitter adopting this approach [126]–[131]. While this major shift from monolithic to distributed design introduces many benefits (e.g., flexibility, simplified maintenance, and efficient resource allocation), this shift also introduces new system challenges (e.g., resource orchestration and application debugging).

The sheer number of distributed components and the increased pace at which microservices are upgraded/rolled out complicates debugging in several ways. First, while the churn of code [132], [133] and increased number of components increases [134], [135] the likelihood of failures and performance changes, accurately and quickly debugging them requires collecting significantly more data [136]–[138], thus introducing a scalability challenge for monitoring. Second, the increased diversity in the type of unique microservices and the increase in layers introduced by the container ecosystem (e.g., service mesh, sidecars) results in a significant explosion in the number of unique types of failures. This naturally increases the range of data and analysis required for debugging, thus introducing an analysis challenge. Consequently, managing the health of distributed services in an automated fashion represents an everyday hurdle for service operators.

To address the scaling challenges, several [66] have proposed scaling out the monitoring infrastructure or adopting ad-hoc CPU bonding or scheduling strategies for the monitor processes, but these solutions require dedicating additional compute on the server for observability which would consequently increase long-term CapEx costs

and reduce the energy efficiency of the data center. Orthogonally, in practice operators [139], [140] configure aggressive sampling strategies, which sacrifice the quality of the monitoring data and detection accuracy. For example, metrics monitors [141], may adopt coarse-grained polling intervals (e.g., 30s), which degrade monitoring QoS KPI, such as accuracy and timeliness, for system (CPU, I/O, memory, power measurements, etc.) and application metrics (e-commerce, financial transactions, cloud-gaming, etc.) that have rapid variations in the order of milliseconds.

To address the analysis challenges, prior work on supervised [67], [142]–[144] techniques have been proposed. However, these require specialized labels which is impractical at the speed with which the microservices evolve. On the other hand, recently proposed unsupervised techniques [65], [145], [146] do not address the scaling challenges. More generally, these techniques (supervised and unsupervised) fail to dynamically adjust to significant workload changes which in turn change a microservice's performance envelope.

To close this gap, is this chapter we describe μView, a system that we design to enable better observability by preserving monitoring accuracy and timeliness, without sacrificing scalability or increasing overhead. The goal of μView is to provide a lightweight but general mechanism to locally analyze each service's metrics (e.g., application or system) at a fine temporal granularity, and generate useful signals about service performance. Figure 4.1 contrasts traditional observability architectures (using the work in [66] as an example) with our proposed system architecture. Our design builds on three unique insights:

First, a service's performance issues can be detected using locally available data. Note that while previous work highlights the need for global knowledge to attribute a problem to a specific service within a microservice, the detection is often based on locally available metrics [147]. For example, Hindsight [147] showed how local data can be used to assist distributed tracing to capture relevant requests. Second, while a service's local metrics are often generated at a fine-granularity, they are often analyzed at a coarse granularity because they are exported to a centralized observability data store (discussed in Sec. 4.1.1). Yet, analysis of this fine-granularity data can yield richer (faster, higher accuracy) insights. Third, the recent rise and adoption of IPUs [148], [149], provide a unique opportunity to process and analyze this richer fine-granularity data without imposing CPU overheads, interference, or bloat on the services. IPUs (alternately, DPUs) are a class of programmable NIC devices equipped with onboard processing units that can be programmed to execute offloaded tasks in data centers, decoupling data center infrastructure from business applications. These IPUs are becoming commodity hardware in data centers [148]–[151].

The central challenges to realizing μView are: (1) The design of general but lightweight and unsupervised metrics analysis techniques to efficiently perform local analysis on the IPU. This is particularly challenging because the diversity of metrics and performance problems is significantly large due to the highly distributed nature of the

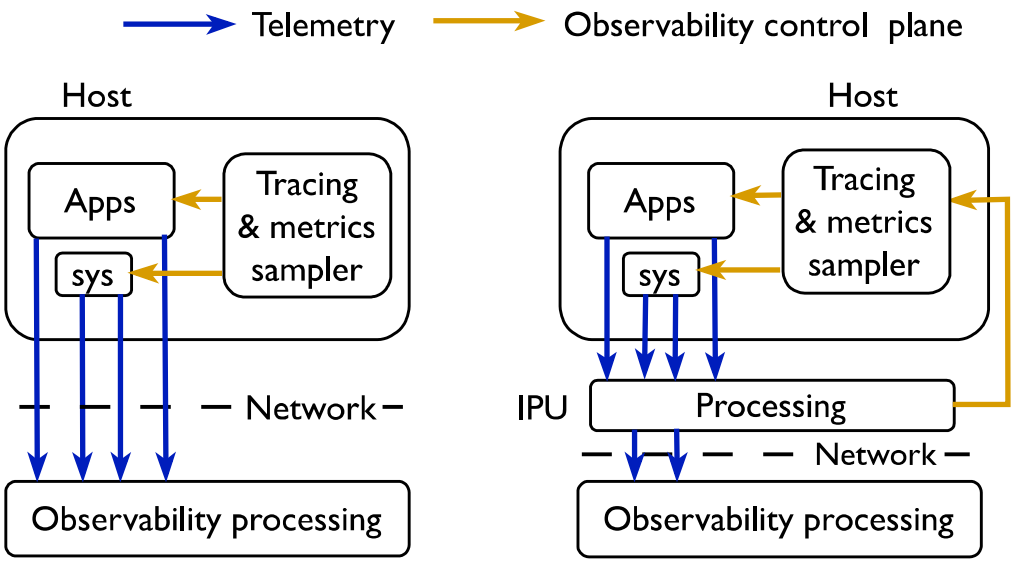


Figure 4.1: State-of-the-art observability architecture (depicted on the left) vs our proposed architecture (depicted on the right).

microservices paradigm and the increased number of layers in the container management frameworks. (2) The high code velocity of the microservices ecosystem implies that analysis techniques must evolve quickly in real-time. To address this challenge, μView leverages a lightweight streaming-based sketch algorithm that operates over multidimensional vectors of metrics in single-pass without requiring local storage of historical data. Our choice of sketch provides continual-learning for embedding new samples in its model, thus automatically adapting to new trends of metric values and relieving the operators from the need to handcraft cumbersome thresholds. (3) While processing on the IPU eliminates CPU processing overheads, as others have shown [66] data transfer if not addressed can impose significant CPU overheads. To address this last component, μView builds on the fact that IPUs support RDMA and introduces a protocol and mechanism designed for efficiently transferring data between the services and the IPU via RDMA.

Contributions. In summary, we contribute the following:

- FD-Sketch: we leverage an unsupervised, lightweight sketch technique that dynamically determines for each microservice the critical metrics and uses them to detect anomalies. We show that this technique can dynamically adjust to changes in workloads in an online fashion allowing it to reduce false positives and false negatives (Sec. 4.4).
- RDMA mechanism: we propose an efficient RDMA-based framework and accompanying system-design choices for coordinating the server-to-IPU exchange of metrics generated across various layers of a microservice deployment stack for localized processing (Sec. 4.2 and Sec. 4.5).

- Prototype/Implementation: we present a prototype implementation of μView (in Sec. 4.6) with several use cases (Sec. 4.7) and evaluate it on a production quality microservice in a testbed using failures injected via a production quality fault injector (Sec. 4.8). We showed that μView increases coverage by 5x while substantially saving costs by 50% over state-of-the-art approaches.

4.1 The observability data bloat

The observability of cloud-native applications across multi-cloud architectures poses a significant challenge due to the substantial volume of generated data. Typical microservices-based applications export several thousands of metrics from all layers of their components. Recent work has reported Netflix collecting about 2M metrics [60]. Whereas, Uber aggregates 500M metrics/s and stores the resulting 20M metrics/s globally [69]. Using AWS-managed Prometheus [70], ingesting 500M metrics/s would cost $21M/month, and storage costs with 150-day retention would be $210k/month. Even for smaller organizations, coping with the escalating observability data represents an increasing financial strain.

At the same time, exporting such a data volume negatively impacts the tail latency of user requests. A large-scale study on a production data center [66] observed that in traditional monitoring systems during data collection cycles, customer traffic suffers from jitter and high tail latency (up to 2×). Moreover, because even small improvements in CPU efficiency can save millions of dollars [152], data center operators typically try to maintain CPU utilization high [153]–[155], which results in low CPU headroom to accommodate both collection cycles and user requests.

Driven by the common-sense rationale that larger data volumes maximize the chance of having data that will be useful in the future, many organizations just collect and store all metrics. However, oftentimes this produces the unintended effect of creating noisy clutter in the environment and hampers the ability to search and gain insights across a plethora of data [156], [157]. This emphasizes the need for more cost-effective approaches and the need to narrow the focus of metrics data collection to informative and insightful data.

To effectively assist developers in localizing failures and SLO violations, the monitoring system should meet QoS demands for coverage and latency. However, traditional monitoring systems fail to satisfy both of these requirements for the following reasons. In terms of *coverage*, metrics collection systems such as Prometheus [141] are configured with coarse-grained polling intervals (e.g. 10-30s), which are not adequate for capturing variations that happen on the timescale of milliseconds, like network traffic fluctuations, spurious I/O faults, power spikes, etc. Similarly, distributed tracers like Dapper [61], Zipkin [63], or Jaeger [62] typically employ a relaxed sampling rate (e.g., below 1%), which results in low coverage as it is likely that a faulty trace might not

Intervals configuration		CPU Usage
Generation	**Ingestion**	
1 s	1 s	12-13%
1 s	30 s	3-4%
30 s	30 s	2%

Table 4.1: Average CPU consumption on a computing node in a k8s cluster, when collecting metrics at a (remote) Prometheus server with different configurations. Generation interval determines the frequency at which metrics are updated, controlled via housekeeping_interval in cAdvisor. Ingestion interval is the frequency used by Prometheus to scrape metrics from cAdvisor (Prometheus's scraping_interval).

be collected. Besides, trace collection does not take any advantage of metrics collection, because the sampling decision is independent of the state of the monitored service metrics. From the point of view of latency, the monitored data encounter several layers on their data-path before being available for queries, including filters/processors [158], network stacks, and storage devices. As a result, monitoring architectures introduce a non-negligible delay before the control plane gets the metrics.

4.1.1 The opportunity for *in-situ* monitoring

A microservice's metrics are generated independently by each layer of the microservice's stack: application, service mesh, and system (i.e., OS). Unfortunately, the metrics at each layer are generated using distinct methods. For example, while system-level metrics are often generated periodically at a fixed frequency, most application-level metrics are only generated when requests are processed. These differences imply that for certain metrics, tuning generation granularity is easy and granularity can be tuned to an arbitrary level. Whereas for others the finest granularity is bounded by the RPC processing rate.

Regardless of the method of generation, ingesting or uploading the metrics into an observability data store incurs significant performance overheads and costs. To quantify the performance overheads, in Table 4.1, we analyze the generation and ingestion overheads for the online boutique microservice [159] deployed with an Istio service mesh [58] on a testbed (details deferred to Sec. 4.8.1). From the table, we observe that most of the overheads are due to ingestion interval and not specifically generation. Thus, while today's observability frameworks can generate metrics at a finer granularity, they use coarse-grained metrics because of ingestion overheads.

Takeaway: This implies that observability frameworks can leverage fine-grained metrics, with marginal costs, for generating richer insights if they can avoid the ingestion overheads – for example, by performing processing locally on the node.

4.1.2 Potential use cases

Metrics reveal the internal state of applications and report the usage of system resources allocated to them. Unexpected variations of these signals likely indicate deviation from an application's common behaviors. Next, we describe a set of use cases that can be enabled by local processing of fine-grained metrics.

Accurate spike detection

Engineers spend significant effort to triage and resolve tail latency; however, such diagnosis usually requires correlating information across different sources, both spatially and temporally. Unfortunately, the coarse-grained interval at which system and application metrics are observed makes posterior time-correlation methods inaccurate and error-prone. In particular, coarse-grained intervals are unable to distinguish between a smooth increase or a sharp jump, which usually offers a crucial mean to pinpoint interesting points in time. Note that this is true both for counters and gauges, which are the two metric types available in Prometheus [141]. Thus, observability systems are often unable to detect, react, or cross-analyze the required data for these kinds of issues.

Coordinated cross container tracing

Even when metrics do indicate an uncommon application state, service operators would benefit from coordinating traces sampled across the different microservices processing the request. However, head-based sampling methods (which is common practice in production systems) blindly sample user requests and thus rely on luck to capture traces associated with this uncommon state, especially if the anomalous conditions last for just a short interval. Locally observing metrics at a timescale close to the rate of user requests in principle provides sufficient granularity to anticipate the realization of representative traces and timely trigger their collection. Triggering trace-based tail sampling using metrics (i.e., integrating μView into Hindsight [147]), allows tail sampling to begin when anomalies occur at the service before they may cascade into being reflected at the RPC request level.

Dynamic frequency sampling

Current monitoring systems adopt a fixed sampling interval to collect metrics. This relatively rigid configuration can make the collection process inefficient or inaccurate. First, different metrics potentially exhibit very different behaviors. For example, CPU usage typically fluctuates more than CPU limits, which changes only in response to reconfiguration. For slow varying metrics, an efficient solution would report new samples only when the metric changes. Second, for all metrics, it is desirable to obtain more frequent samples when an anomalous behavior is detected. The

rationale here is similar to Hindsight's argument for tracing: it is more desirable to have all edge-case traces with low overhead as opposed to sampling at low sampling percentages. Since monitoring tools from cloud providers (e.g., Amazon-managed Prometheus) charge users for the ingestion and storage of observability data, there is also a cost-benefit that comes with dynamic frequency sampling.

4.2 μView overview

Our design differs from prior approaches in that it adds a local stage for metrics processing at each node. Our architecture opens up the ability to locally monitor anomalous behaviors and deploy data hygiene algorithms before data ingestion and storage. With local metrics processing service operators can deploy strategies to filter out irrelevant data and gain online insights about relevant information, without cluttering storage with huge data volumes.

At a high level, we can decompose the architecture into a data plane and a control plane. We overview each in turn. Figure 4.2 shows a diagram of this architecture and the components that comprise it.

Data plane

The data plane implements local metrics processing at each node. It consists of an array of per-service Local Metrics Analysis Pipelines (LMAPs). Each LMAP is associated with a single microservice.[1] A LMAP receives as input a vector of metrics, referring to the corresponding microservice, and produces an output signal indicating whether the metrics vector contains anomalies.

For generality, μView does not dictate how metrics are to be monitored and LMAPs are oblivious to their sources. Metrics can be generated by several sources, including the application, the OS kernel or a service mesh component, like Istio [58]. Regardless of the source, all metrics processed by a specific LMAP must refer to a single microservice. As detailed later, the μView control plane handles the task of connecting the components that monitor and export metrics with the LMAP that consumes them. In line with existing practice, the set of per-service metrics is configured by service operators (e.g., DevOps engineers); we consider this as an initialization parameter for μView.

A LMAP consists of a set of preprocessing transformations and a sketch-based classifier. The preprocessing step converts the input *metrics* to *feature* vectors. The sketch-based classifier performs anomaly detection based on these features. Metrics

[1] We use the terminology service and microservice interchangeably, to indicate a self-contained application module, instantiated as one or more pod replicas in the cluster. Functionally, it is equivalent to the concept of service object in k8s.

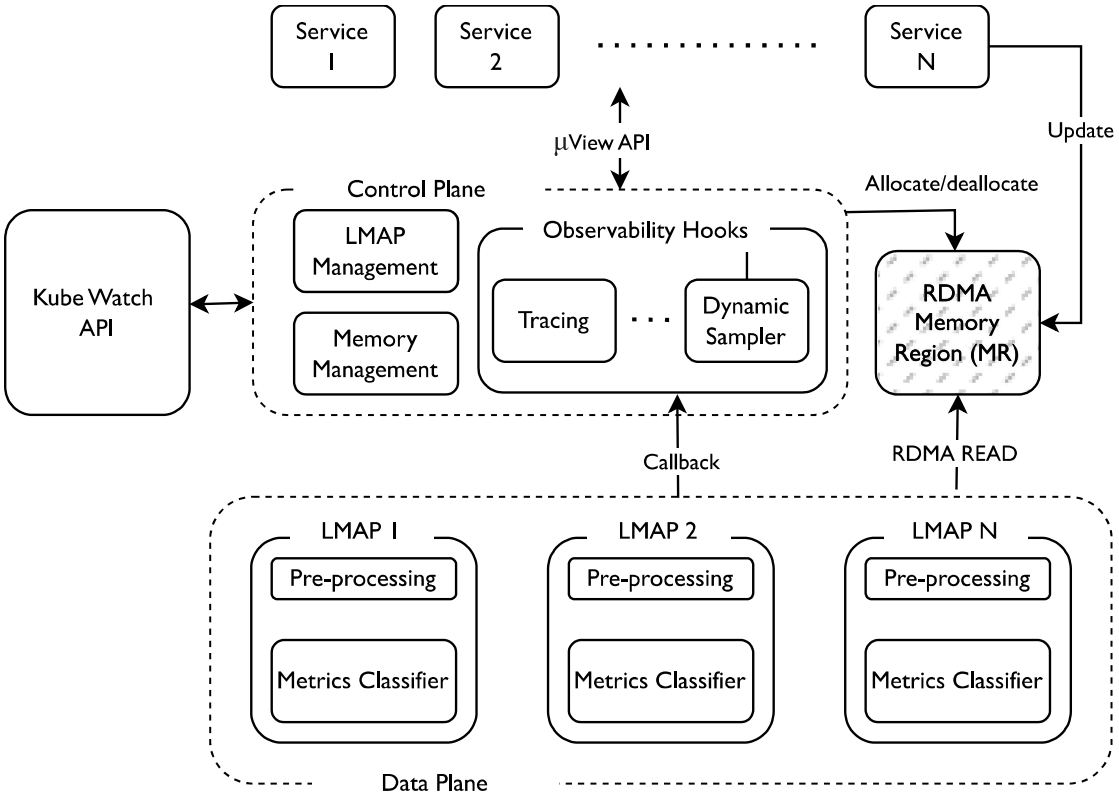


Figure 4.2: The architecture of μView.

produced by different replicas of the same service on the same node are typically aggregated into a single feature vector. Sec. 4.4 describes the metrics processing pipeline in detail. Finally, the outputs from each LMAP are used by the control plane, which can trigger actions in response.

The frequency at which metrics are ingested by the LMAPs is called *local scraping interval*. Service operators configure this parameter.

Control plane

The control plane orchestrates the interaction between the sources that generate metrics (e.g., application, OS kernel, etc.) and the data plane. At a glance, a control plane component executes on each compute node; this component (1) manages the lifecycle of the LMAPs, (2) handles how the data plane can access memory locations where metrics reside, and (3) starting from LMAP outputs, produces actionable insights for observability. For generality, μView does not dictate how insights are used directly. Our approach is general in that various observability hooks can be configured and μView merely invokes the hooks when appropriate. This allows integration of μView with existing observability tools, such as distributed tracers, loggers, and

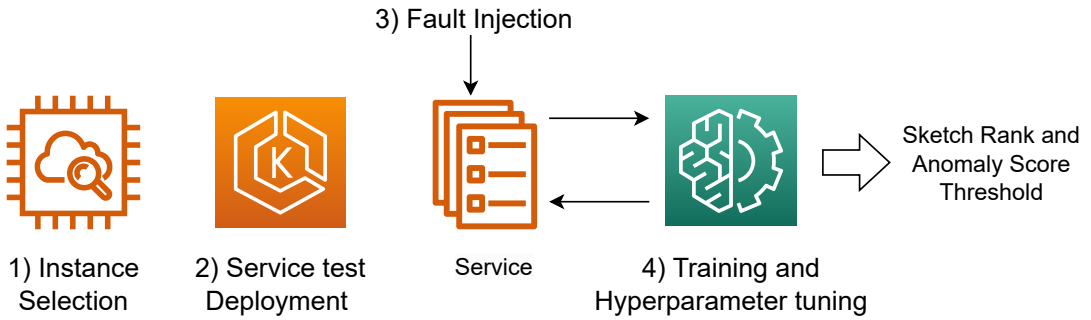


Figure 4.3: Offline configuration workflow.

metrics collectors. We listed several use case scenarios in Sec. 4.1.2; Sec. 4.7.1 illustrates a dynamic sampler hook that pushes metrics to a remote Prometheus collector based on data plane insights.

To adopt μView, microservices need to interface with the control plane via an API. We detail the *μView API* in Sec. 4.2.2. When the per-service metrics are specified ahead of time (i.e., they are stated in the service initialization configuration), we can generate automatic instrumentation to invoke the API for well-known metrics.[2] In other cases, the microservice application needs to be modified to insert an API invocation for each metric that it intends to export to μView.

IPU offloading

μView's design naturally lends itself to offloading data plane processing from the host CPU to emerging programmable IPUs [148], [151], which is beneficial to free up precious compute resources for the actual application logic. Namely, the IPU fetches metrics from memory locations registered by the control plane. The IPU also executes LMAPs and returns their outputs to the control plane. We elaborate on the advantages of this design choice in Sec. 4.5.

4.2.1 Offline configuration workflow

Before we proceed to a detailed discussion of μView's components, it is worth providing an overview of the offline configuration workflow for using our approach, which is illustrated in Figure 4.3. Following best practices [160], [161], we expect that service operators run automated tests to validate deployments before service instances enter production. We leverage this phase to train the LMAPs and tune their

[2]The instrumentation would override access for well-known metrics at service initialization time via a LD_PRELOAD mechanism.

Description	**API Call**
Manage LMAP	LMAPID `newLMAP`(Config, ServiceID) void `configLMAP`(LMAPID, Dict<ServiceID, List<MetricConfig>) void `deleteLMAP`(LMAPID)
Configure Metrics	MetricID `addMetric`(LMAPID, Metric, Type, AggType, Frequency) void `deleteMetric`(LMAPID, MetricID)
Add Hooks	HookID `registerHook`(List<LMAPID>, HookFn*)

Interface	**Declaration**
HookFn*	void `hook`(Feature, Output, AScore)

Table 4.2: μView API and hook interface.

hyperparameters. In particular, we use this phase to decide the rank k of the sketch-based classifier (explained in Sec. 4.4) and the anomaly score threshold γ. This is done for each LMAP, which corresponds to the number of microservices.

We start by collecting a dataset of metrics, which we assume to be free of anomalies. We complement this dataset with a set of metrics collected while synthetic faults are injected. We split the dataset into a training set (70%) and a testing set (30%).

The hyperparameters are chosen by grid-searching possible values while operating the following process. To pick the threshold, we consider the distribution of anomaly scores over the training dataset and set γ to a sufficiently large percentile (e.g., 90-99th). We compute the F1-score over the testing set. We repeat this process for each value of k and pick the one that maximizes the F1-score. We elaborate more details in Sec. 4.8.2.

As explained in Sec. 4.4, μView's sketch-based approach is adaptive to changes in the workload. Thus, we do not expect to retrain the LMAPs unless the workload changes drastically. In practice, we expect to retrain the LMAPs only when the application code is updated.

4.2.2 The μView API

μView exposes a management API and a hook interface. To facilitate LMAP metrics collection and management, μView exposes a configuration API (Table 4.2) which allows operators to register metrics of interest and metadata about the metrics required for the processing pipeline as well as initial configuration for the LMAP based on the offline training phase (including the sketch parameters). This interface also allows the operator to manage LMAPs.

To enable a general set of hooks that can take arbitrary actions based on the insights generated by an LMAP. μView presents a simple interface (Table 4.2) that all hooks must implement. In brief, all hooks are event-driven and must implement an

interface that implicitly registers them as callbacks when insights are generated by an LMAP.

4.3 Control plane functions

The control plane is responsible for the following functions.

LMAP management. The control plane registers with the cluster manager (e.g., k8s) to be notified about new pods and their status. Whenever the first replica of a new service is scheduled on a node, the control plane allocates the corresponding LMAP.

Multiple replicas of the same service on the same node share the same LMAP. When the last replica of a service is removed from a node, the control plane deallocates the corresponding LMAP.

Memory management. The control plane allocates memory regions for storing metrics exported by microservices. This is done to provide an efficient way for LMAPs to access metrics. Additionally, when the data plane is offloaded to an IPU (Sec. 4.5), the control plane optimizes memory allocation by grouping metrics into contiguous memory addresses. This allows LMAPs to batch multiple metrics into fewer memory accesses over RDMA, as also done in [66].

4.4 Local metrics analysis pipeline (LMAP)

In this section, we describe the challenges underlying the design of an LMAP and the components to address them. Recall, thanks to its proximity to the monitored services, the μView data-plane supports small local scraping intervals that can follow short-term variations of monitored metrics. In brief, an LMAP must quickly perform anomaly detection in real time while being sufficiently flexible and efficient to support an arbitrary set of operator-defined metrics streaming at arbitrarily configured frequency. To flexibly and efficiently support real-time processing, the LMAP requires a metrics processing pipeline to sanitize the data (Sec. 4.4.2) and a general technique to analyze/extract anomalies/insights from arbitrary metrics (Sec. 4.4.1).

4.4.1 Sketch-based metric classification

The high volume of continuously collected metrics imposes limitations on processing time, memory, and storage availability for the metrics classifier modules when they are co-located on servers running application containers.

Design goals. The metrics classifiers have to satisfy the following requirements.

1) Low overhead. The metrics classifiers should be fast and consume few CPU cycles, which are precious to the application's business logic. When deployed as a user-space module, it should not waste CPU cycles devoted to the application workload.

2) *Adaptive.* The anomaly detection module should be able to continuously learn and adapt to concept drifts in the underlying metrics data. This is particularly important in cloud-native environments, where services can be scaled horizontally, and metrics trends are highly dynamic and can change over time.
3) *Explainable.* The classifier should provide explainable outputs, which means that ideally we expect it to indicate which features triggered a detected anomaly.

Anomaly detection with subspace analysis

To satisfy all design goals simultaneously, we refer to a class of unsupervised anomaly detection techniques known as *subspace-based* anomaly detection [162]–[165], which have been shown to work well for anomaly detection in network traffic. Informally, subspace-based anomaly detection leverages subspace analysis (SA) [166] to generate a low-rank matrix approximation (e.g., PCA, SVD, etc.) of a dataset of non-anomalous data points. To decide whether or not a previously unseen data point is anomalous or not, it checks that the new data point has a "good" representation based on this low-rank matrix. Because these algorithms use compact matrix representation, they are fast and space-efficient.

More formally, assume we have a low-rank matrix U which can be used to *well represent* any data point in dataset M on a linear subspace. For a new input data point $\mathbf{x}$ (i.e., a *feature* vector), one can project $\mathbf{x}$ onto the low-rank matrix U and check whether the resulting embedding $U\mathbf{x}$ is close to the embeddings of the non-anomalous data points. Equivalently, since the input data point is known, one can check the reconstruction error $\|\mathbf{x} - UU^{\intercal}\mathbf{x}\|$. If the reconstruction error is large (i.e., above a certain threshold γ), then the data point is likely anomalous. The intuition behind this approach is that the low-rank matrix U captures the principal information of the dataset M in a compact form, and thus it can be used to detect anomalies in new data points whenever the information contained in U is not sufficient to represent them.

In μView, the matrix M corresponds to a dataset derived from a microservice's metrics. As mentioned (Sec. 4.2.1), the dataset M is collected offline and used to train the initial low-rank matrix U. At runtime, the low-rank matrix U is updated incrementally as new data points are collected. This enables μView to adapt to concept drifts in the underlying metrics data.

Sketch-based classifier

A straw man is to recompute U from scratch as the SVD decomposition on each incremental version of M, but this approach is very demanding in terms of storage and computation. In fact, the dataset with t data points, each with m features consists of a matrix $M \in \mathbb{R}^{t \times m}$. A second straw man is to rely on a separate server to compute the matrix U. However, this would require synchronizing the collection frequency of the metrics at the server with the local scraping interval, which imposes significant CPU

overhead on the nodes (c.f. Table 4.1) on top of higher network bandwidth consumption. Moreover, even accepting some degree of down-sampling, the SVD computation is known to be resource-demanding for large matrices [166].

The challenge we face is how to efficiently update the low-rank matrix approximation in a streaming setup, i.e., at time t obtain a matrix U_t using only information from time $t-1$.

Borrowing from [167], we use the Frequent Direction sketch algorithm (FD-Sketch). The algorithm builds on the idea of replacing the large matrix $M \in \mathbb{R}^{m\times t}$ with a significantly smaller sketch matrix $S \in \mathbb{R}^{m\times l}$, with $l \ll t$, such that $S^\top \mathbf{x} \approx M^\top \mathbf{x}$. That is, the computations performed on S give results with minimal loss in precision compared to those performed on M.

Therefore, an up-to-date low-rank approximation matrix $U_t \in \mathbb{R}^{m\times m}$ can be obtained at time t from the SVD of S_t. This only requires locally storing the sketch S_t, and updating it incrementally in a streaming setup, as originally proposed in [167]. For a formal analysis, the interested reader can refer to [168]. A survey on streaming subspace analysis methods can be found in [166].

Anomaly score. For each new input data point $\boldsymbol{x}_t$, μView produces a reconstruction error vector $\boldsymbol{\alpha}_t \in \mathbb{R}^m$, where each component $\boldsymbol{\alpha}_{t,i}$ quantifies how "anomalous" the metric $\boldsymbol{x}_{t,i}$ is.

$$\boldsymbol{\alpha}_t = \boldsymbol{x}_t - U_{(t-1)_k} U_{(t-1)_k}^\top \boldsymbol{x}_t \tag{4.1}$$

In Eq. (4.1), $U_{(t-1)_k}$ is the low-rank approximation matrix computed at time $t-1$ using the sketch algorithm. Moreover, the subscript k indicates that only the first $k < l < m$ columns of $U_{(t-1)} \in \mathbb{R}^{m\times m}$ are used to compute the reconstruction error. The intuition behind this is that the first k columns of U contain most of the information – this is by construction in SVD.

The vector $\boldsymbol{\alpha}_t$ is used to derive an *anomaly score*. Note that the larger the value of $\alpha_{t,i}$, the more likely the metric $x_{t,i}$ is anomalous. Therefore, the anomaly score is the norm of the reconstruction error vector, i.e., $\|\boldsymbol{\alpha}_t\|^2$. We say that there exists an anomaly at time t if $\|\boldsymbol{\alpha}_t\|^2 \geq \gamma$.

FD-Sketch meets μView design goals

We highlight the following aspects of the FD-Sketch algorithm that make it suitable for the requirements of μView.

One threshold to rule them all. μView relieves the service operator from the burden of configuring per-metric thresholds. Thanks to the FD-Sketch mechanism, μView relies on a single threshold γ to detect anomalies. At the same time, service operators can identify *critical metrics* by inspecting the top-j components by the absolute value of the reconstruction error vector $\boldsymbol{\alpha}_t$. In contrast to previous work [165], by considering the entire vector $\boldsymbol{\alpha}_i$ we open up to the possibility of turning a detector into a classifier. Namely, depending on which top-j metrics have the largest reconstruction

error, our approach can potentially identify different categories of failures. For example, if the top-j metrics are mostly related to network I/O, then the anomaly is likely due to a misbehaving network stack.

Thanks to the FD-Sketch algorithm, μView meets all the design goals simultaneously. First, FD-Sketch approximates a large SVD decomposition and only needs to maintain a relatively small matrix S, therefore it is lightweight and keeps the computation overhead bounded. Second, since FD-Sketch updates an approximate dataset matrix S_t in a streaming setup, wherein the S_t is recomputed based on new data points since time $t-1$, it remains aligned with recent data and provides adaptability. Finally, μView generates explainable outputs, as it provides a reconstruction error vector $\boldsymbol{\alpha}_t$ which can be used to identify critical metrics that have the largest impact on the reconstruction error.

4.4.2 Metrics processing pipeline

As part of the processing pipeline, raw metrics undergo a set of pre-processing transformations, before entering the sketch classification. The transformations are simple and account for the following aspects: (1) metrics that are cumulative need to be buffered and accumulated at each sample, (2) metrics may need to be aggregated across replicas of the same service,[3] and (3) metrics need to be mean-centered and rescaled to unit standard deviation.[4]

4.5 Offloading μView to an IPU

μView enables real-time processing of the metric stream from any microservice through a corresponding LMAP, which resides on the same node as the monitored microservice. However, μView's high-resolution metrics processing creates compute overheads. Additionally, metrics pre-processing described in Sec. 4.4.2 involves transformations such as data whitening and accumulation.[5] Increasing the collection frequency results in increased compute requirements for performing such operations.

Apart from CPU overhead, the μView data plane also needs to access memory where it stores sketches, which may induce cache pressure on the host and deteriorate memory access performance of the running services. Note that the μView memory footprint is proportional to the number of services being monitored, the number

[3] μView supports standard aggregation functions including `max`, `min`, `sum` and `mean`; it can be extended with custom aggregators.

[4] This is needed to avoid SA to overweight components with large norm.

[5] This pre-processing transformation depends on the classification algorithm being used, however, it is common to execute pre-processing steps of similar complexity for other inference algorithms.

of metrics per service, and the rank of the sketch approximation. For example, monitoring 1,000 metrics from 100 services with a sketch approximation $k = 10$ and using a double-precision floating-point format would allocate 8 MB of memory.

To alleviates CPU contentions and cache pressure on the host, freeing up resources for applications, we believe that offloading μView data plane to an IPU is highly beneficial.

4.5.1 Off-path offloading as a natural fit

Modern IPUs may provide *on-path* or *off-path* or both types of offloading capabilities. In the case of on-path capabilities, the offload functions run on the NIC cores, resulting in high efficiency but poor flexibility due to the low-level programming interfaces. In the case of off-path capabilities, a System-on-Chip (SoC) that is integrated with but separate to the NIC cores, is available as a computing resource. The SoC can host a full-fledged OS and run general-purpose code.

We regard the off-path SoC as an appealing offloading platform for the μView data plane. Modern IPUs typically feature more than a dozen SoC processors, aligning well with the task of metrics analysis, which naturally lends itself to embarrassingly parallel computation. For example, the NVIDIA BlueField-3 and the Intel IPU E2000 both boost 16 × ARM cores [151], [169]. Moreover, this type of compute complex allows executing high-level code runtimes and libraries, avoiding additional development complexity for the metrics analysis pipelines.

μView leverages RDMA for host-IPU communication, achieving host CPU bypass. In this way, μView supports high-frequency metric reading without incurring the overhead of system calls and memory copies. We choose RDMA instead of DMA because RDMA is a consolidated technology for high-performance networking in data centers, and has been recently demonstrated to provide higher throughput ($\approx 2\times$) than DMA for communication between the host and a BlueField-2 IPU [170].

4.5.2 RDMA communication

In RDMA, applications interact asynchronously with a Queue Pair (QP), consisting of a send queue and a receiving queue, which are managed by the RDMA NIC (RNIC). The RDMA QPs act as an asynchronous interface between RDMA applications and the RDMA stack to send and receive data. The RDMA application posts to the QP new Work Requests (WR) that instruct the RNIC with the communication primitive to be executed. RDMA supports *one-sided* and *two-sided* operations. In *one-sided* operations, e.g., `READ/WRITE` primitives[6], the sender directly reads(writes) data from(to) the receiver's memory. In one-sided operations, the receiver application is not aware of

[6]In the InfiniBand terminology, they are referred to as *verbs*

the communication. The receiver grants access to its memory by registering a memory region (MR) with the RNIC, and providing the sender with a *remote key (rkey)* before RDMA communication. Authorization to access memory based on the *rkey* is verified by the RNIC, thus the receiver CPU is not involved during RDMA one-sided communication.

In contrast, *two-sided* operations, i.e., `SEND/RECV` primitives, are similar to a typical socket-like communication, where the receiving application waits for data by posting a `RECV` WR, and the sender a corresponding `SEND` WR to send new data. For both one-sided and two-sided operations, the application polls a Completion Queue (CQ) to receive notification events, once the WRs are handled by the RNIC. We focus on the `READ` primitive to fetch metrics from the host to the IPU, as it allows us to avoid the host CPU involvement.

4.6 Implementation highlights

Next, we discuss how μView integrates metrics collection based on RDMA with LMAPs on a NVIDIA BlueField-2 IPU. μView leverages the parallelism offered by the IPU SoC by distributing the LMAPs across the available cores. μView spawns one agent per core, and each agent runs the LMAPs assigned to that core. Each agent maintains a TCP connection with the control plane. The control plane instantiates a new LMAP for each service and assigns it to an agent.

The agents execute a straightforward process: at every local scraping interval, they fetch metrics for the LMAPs they manage, feed metrics to the LMAPs, send back the outputs to the control plane, and wait for the next interval.

Reading metrics from the host to the IPU

The agents issue RDMA `READ`s to fetch metrics from host memory, avoiding the host's CPU involvement. The μView agents on the IPU adopt unidirectional one-sided RDMA communication, via `READ` s issued to the host. On the host-side, a new RDMA QP is allocated for each agent, and the QP identifier information is exchanged on the corresponding control channel with the IPU. Parallel QPs avoid head-of-line blocking across `READ` requests from different IPU agents, which could happen if the host-side allocated a single QP, because RDMA verbs are consumed in the same order they are posted. On the IPU-side, agents establish one-to-one RDMA connections with the host QPs, based on the received identifiers. As a design choice, every agent creates its own QP and shares a single CQ for the send queue and the receive queue, since RDMA communication is unidirectional and in μView IPU only the send queue is used for `READ` operations. On the BlueField2 SoC there are 8 ARM CPUs, while the upcoming BlueField3 will feature 16 processors: this number is significantly lower than the number of connections that would drop RDMA throughput because of cache contentions

and QP evictions [171].

The control plane manages the memory regions (MRs) where metrics reside. When a service calls the addMetric API (c.f. Table 4.2), the control plane allocates memory for the metric (its type determines the size) in a shared memory segment. This follows the same principles of ZERO [66], avoiding metrics being scattered across process or kernel space. The agents receive the configuration of metrics that each LMAP handles from the control plane, which includes the RDMA rkey and the offset within the MR for each monitored metric. Memory allocations(de-) and agent configuration happen only once when a new service is deployed, therefore the overhead is limited. The more dynamic serverless scenario is not considered in this work.

Tuning performance

To optimize memory access, the control plane groups metrics by service and allocates them contiguously in memory. This allows the agents to batch multiple metrics into fewer memory accesses (e.g., fewer RDMA READs). Moreover, we also can batch multiple RDMA WRs in a single system call, supported via the ibv_post_send primitive. To further optimize performance, we request a completion event only for the last READ of every scraping interval. Since READ requests are consumed in the order they arrive, the last completion event works as cumulative completion event for all the preceding requests. Finally, the memory management module ensures that an MR contains only metrics belonging to services that are allocated to the *same* agent (same IPU core). Further, we align the size of the MRs to the page size, i.e., 4KB, on the ARM architecture.

Since the IPU is meant to define a new trust boundary for infrastructure processing [148], [149], [151], we consider that an agent can process metrics of different services. A complete security analysis of μView is beyond the scope of this paper.

4.7 Observability hooks turn insights to actions

In this section we describe how μView can use the LMAP outputs to trigger actions in response to detected anomalies. We focus on two use cases: (1) distributed tracing and (2) metrics filtering.

4.7.1 Use case: distributed tracing

Metrics reveal the interval state of services and their containers at any point in time. Their unexpected variation potentially indicates an anomalous state, which negatively impacts user requests. As discussed in the Sec. 2.1 of the previous chapter, the continuous-time online analysis of performance metrics is pivotal to achieve

timeliness and accuracy. In μView we exploit the proximity of the LMAPs to the monitored metrics, and we try to early-catch from the locally observed metrics any preamble of future problematic executions. Building on this core idea, we describe how μView supports distributed tracing to sample informative requests.

At every *local* scraping interval, μView outputs a binary decision on whether the tracer should sample user requests. Later, it informs the external distributed tracing library that previously registered with the distributed tracing hook (Sec. 4.2.2) about its decision. Logically the μView sampling policy is based on (1) aggregating the output of all LMAPs of services involved in a user request, and (2) deciding to sample the request if at least one of them has classified its metrics as anomalous. At this point, this sampling decision is held until the subsequent classification cycle, when a new decision will be taken based on a new metrics reading.

To achieve this in practice, μView requires coordination across all the tracing hooks distributed across the cluster nodes. Specifically, the tracing hooks must agree on a distributed decision on whether to sample the requests until the next local scraping interval. This is because the microservices involved in a user request are distributed on multiple nodes, therefore individual tracing hooks cannot make a decision independently, as they only are aware of the output of local LMAPs. To minimize complexity, we chose to elect to the role of *leaders* the tracing hooks on the nodes that host at least one replica of the frontend service. The leader tracing hooks are responsible for triggering the tracer to sample user requests. Because all user requests are routed first through the frontend, leaders always sit at the beginning (head) of the execution path. Therefore, they can implement the sampling decision as soon as the first service in the execution path receives the user request. In this way, we don't introduce additional complexity with respect to a head-based sampling strategy. μView outputs one decision for different operations invoked on the frontend service. In this way, the anomalies that LMAPs detect on services not involved in the execution path of an operation do not play a role in sampling the requests of such an operation.

4.7.2 Use case: dynamic metrics sampling

The dynamic metric sampling mechanism allows μView reporting informative metrics variation events timely, and save the data volume generated for their collection. Our dynamic sampling mechanism builds on the core idea of using the anomaly score value as a measure of the informativeness of the collected metrics. For every local scraping interval, the sampler hook receives the per-service anomaly score $||\boldsymbol{\alpha}||^2$ from the LMAPs. Then the dynamic sampler hook takes a per-service decision about whether the current metrics should be sent to a remote monitor system. The dynamic sampler hook decides to send metrics when the anomaly score is above a threshold. In this case, it notifies external libraries via the registered callback. Dynamic metric sampling fundamentally implements a *push*-based mechanism for metric collection.

However, we envision it can be integrated within a baseline periodic pull-based metrics collection, running at a lower frequency. In fact, there is increasing demand from organizations for a more flexible monitoring infrastructure where co-existing pull-based and push-based metrics collection can be configured [172].

4.8 Evaluation

In this section, we evaluate the performance of μView. We first describe the experimental setup and motivate the spectrum of faults we injected. Then we present the results of a set of microbenchmarks aimed at tuning the LMAPs for individual pods. Finally, we present the benefits of μView observability hooks for the two use cases: distributed tracing and dynamic sampling.

4.8.1 Experimental setup

System implementation

We validated our design by implementing a μView prototype. We have implemented the LMAPs and observability hooks components in Python language, with 1278 LoC in total, The control-plane logic is written in 1180 LoC of C++ language, and uses `libibverb` and `librdmacm` for `RDMA` communication. We deployed a k8s v1.25.5 (k8s) cluster with Istio v1.16.1 service mesh layer. The cluster runs on 4 nodes (3 workers + k8s control-plane), each equipped with 8 Intel Xeon E3-1230v6 CPUs at 3.50GHz, 32 GB of RAM, and 100 Gbps network interfaces. We deployed Istio in its sidecar-based default configuration, which includes a co-located proxy container for each pod (service). However, we note that μView is agnostic to the Istio deployment strategy and is also compatible with its sidecarless configuration.

Benchmark application

We evaluated μView using Online-Boutique [159], a real-world microservice application which is maintained by Google. We chose Online-Boutique because it is a widely adopted cloud-native benchmark for evaluating fault localization [145], [173], metrics selection [174], resource allocation [175] and trace sampling [142], among others. It implements a Web-based e-commerce platform with 10-tier stateless polyglot services and a Redis cache. All microservices communicate with each other via gRPC calls. We set the minimum number of replicas to 1 for each microservice and generate user requests to the frontend service using Locust [176], a multi-threaded open-source load generator written in Python. Locust uses one thread per user and allows us to customize user actions (i.e., request endpoint) and the time between different actions. We distributed user actions non-uniformly across microservices to

Metric name	Metric source	Description
request_bytes_sum	Istio	Number of bytes in RPC requests among services
request_duration_milliseconds_count	Istio	RPC duration between interacting services
request_messages_total	Istio	Number of application messages exchanged by interacting services
response_bytes_sum	Istio	Number of bytes in RPC responses among services
blkio_device_usage_total	Resource	Number of bytes transferred to/from block devices
cpu_system_seconds_total	Resource	CPU utilizations in kernel mode
cpu_user_seconds_total	Resource	CPU utilizations in user mode
container_processes	Resource	Number of processes in a service container
oom_events_total	Resource	Number of out-of-memory events in a service container
redis_commands_duration_seconds_count	Application	Number of commands executed by Redis
redis_db_keys	Application	Number of keys stored in Redis
redis_commands_processed_total	Application	Number of call by command type HGET,HSET

Table 4.3: Examples of collected metrics.

emulate a realistic user interaction with an e-commerce website. Product browsing represents the most common activity, while checkout is less frequent than cart view.

Observability datasets

For our evaluation, we collect the training and test datasets consisting of metrics and traces by instrumenting the microservices with the OpenTelemetry SDK [158]. [7] For each microservice, we collected the following set of heterogeneous metrics. For service-level metrics, we considered Istio metrics and application metrics. Istio metrics are exported by the Envoy sidecar proxies and are related to requests and responses to services, such as `istio_requests_total`. Application metrics are generated by the user service running within a pod, and provide information about the state of the service instance. For example, Redis is instrumented to generate a set of metrics relevant to its performance, such as the number of keys it stores, the number of requests for each command (e.g., `HGET`, `HSET`), etc. Container-level metrics monitor the utilization of system resources for individual containers, such as CPU, memory, disk I/O, and network data transfers. We collect them from the k8s cAdvisor. Unless otherwise specified, we collected metrics emulating a local scraping interval of 1 second via Prometheus. [8] Some exemplars of collected metrics are reported in Table 4.2. We generated the training corpus in a controlled environment (Sec. 4.2.1), where we allocated vCPUs and memory limits of each microservice such that utilization is no larger than 10%, and verified that throttling did not occur. For all services except Redis, the dataset comprises 71 Istio metrics and 70 cAdvisor metrics.

[7] We use a single OpenTelemetry collector in *gateway-mode*, where the collector instance runs as a standalone service and acts as a proxy for metrics and traces between the services and Jaeger [62] (for traces) and Prometheus [141] (for metrics). We configure trace sampling through the OpenTelemetry collector.

[8] We also set the `housekeeping_interval` in cAdvisor consistently.

Fault injection

For synthetic fault injection, we used ChaosMesh [177], the industry's leading chaos testing platform. ChaosMesh supports various fault types and offers streamlined integration with k8s clusters.

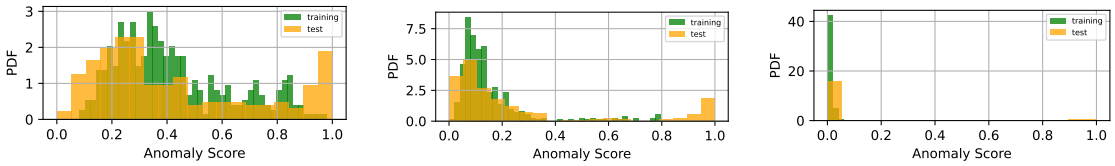
Resource contentions. In line with previous works [65], [178], we used ChaosMesh to inject resource-related faults, such as CPU spikes or high memory usage, that often arise in a production environment. Resource faults are common due to the presence of application bugs e.g., memory leaks or inefficiencies such as excessive heap memory usage or JVM garbage collector overhead. In addition, they can result from (cyclic) dependencies among co-located or interacting services [179].

4.8.2 Hyperparameter tuning

We perform a set of sensitivity analysis experiments to understand how the classification performance of the sketch changes in response to hyperparameter tuning. For this, we deploy a single pod in isolation and run the service for 1 hour. We train the sketch using roughly 1200 vectors of metrics generated during the first 20 minutes. The result of the data (for the remaining 40 mins) is used for testing. During the collection of test data, we inject short CPU stress anomalies at periodic intervals . Each anomaly lasts about 30 seconds.

Choosing anomaly score γ

The choice of the anomaly score is based on the probability distribution of the anomaly score after the training phase. Since we assume the training data contain a baseline of healthy metric samples (i.e., non-anomalous), a natural choice is to derive a threshold around the tail of the distribution. The effectiveness of this approach depends on the shape of the training score distribution. We first study how this changes for different values of the sketch reconstruction rank k. Figure 4.4 shows the results for different values of k. For small k, the sketch tends to underfit the training data, as shown by the tail of the training score distribution approaching 1 and by the small statistical distance between the training and test anomaly score distributions. Under these circumstances, the definition of the threshold γ is challenging. If γ is set to a high percentile the number of false negatives increases quickly, whereas smaller γ would generate several false positives. On the other hand, for $k = 20$ the training distribution is concentrated around 0, showing overfitting to the training data. In this case, the classifier doesn't generalize well to different data.



(a) k=5 (b) k=15 (c) k=20

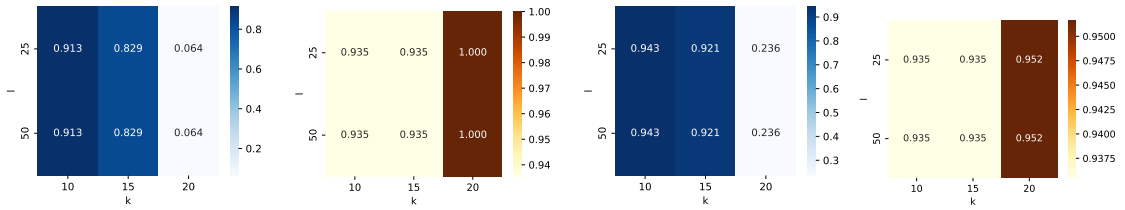
Figure 4.4: Anomaly score distribution for different number of columns k of the FD-Sketch. The plots on the left-side show a configuration where the FD-Sketch tends to underfit, whereas on the right-side the FD-Sketch overfits the training data. We choose the anomaly score threshold γ based on the training score distribution.

Detection performance

We vary the sketch reconstruction rank k, and the sketch size l and measure the precision and recall of the classifier for different anomaly detection thresholds derived from the q-percentile of the anomaly score distribution. We report the results in Figure 4.5. We observe that increasing k gives better recall but lower precision. This is because increasing k leads to skewed anomaly score distribution, which intuitively corresponds to higher sensitivity of the sketch to metrics variations and ultimately leads to overfitting. For example, for $k = 20$, a small noise in the metrics of the test data produces large reconstruction errors, which increases the number of false positives and reduces precision. This effect is mitigated by choosing a higher anomaly score threshold. On the other hand, we observe that the recall score benefits from more skewed anomaly score distribution, but is generally less sensitive to the parameter k. Overall, we observe that both precision and recall do not significantly change when increasing the sketch size from $l = 25$ to $l = 50$. We conclude that the choice of k and γ should be made based on the desired trade-off between precision and recall.

4.8.3 Sketch detector performance

In this section, we study the performance of the sketch detector under highly dynamic workloads with a set of microbenchmark experiments that consider individual pods in isolation. In particular, we answer the following question: *can the sketch discriminate a transient overload from persistent overload?* We define *transient* overload as a temporary overload condition due to an increase in traffic before the k8s Horizontal Pod Autoscaler (HPA) deploys additional replicas of a service to match the ingress workload. This is a challenging situation because the monitored metrics will not remain stationary and will drift. Thus, these experiments allow us to demonstrate the adaptability of the LMAPs. Theoretically, the sketch should be able to discriminate between a scenario where the cluster has spare resources to accommodate the



(a) Precision score, $q = 99$-th (b) Recall score, $q = 99$-th (c) Precision score, $q = 99.9$-th (d) Recall score, $q = 99.9 - th$

Figure 4.5: Impact of varying sketch parameters on *precision* (blue color map) and *recall* (orange color map) metrics. The anomaly detection threshold is set to the q-th percentile of the anomaly score distribution, i.e., $\gamma = F(q)$.

incoming load and one where the overload is *persistent* because the cluster has insufficient resources, i.e., scaling stops either because of the max replicas setting in HPA or the cluster is out of resources. In the former scenario, the sketch should not raise any alarm whereas in the latter the sketch should detect an anomalous state and raise alarms.

Since in the μView, every LMAP works independently, to answer this question we run a microbenchmark experiment on a single service. We analyze the *frontend* service and deploy its corresponding HPA that starts the deployment with a single replica. To emulate the case where the cluster cannot schedule new pods, we configure the kubelet's parameter `maxPods`, which limits the number of pods that can be scheduled on a node. We also make sure the HPA can saturate the node by ensuring `maxReplicas` > `maxPods`. We train the classifier under stationary load conditions, with a constant request arrival rate. The classifier parameters for this experiment were $l = 50$, $k = 12$, and $\alpha = 99.9$. While collecting the test dataset, we linearly increase the load by adding a new user every 30 seconds, starting from 11:35 as shown in Fig. 4.6a. The figure highlights the effect of a linearly increasing load on the CPU consumption. We plot the maximum CPU consumption across all pod replicas (after proper normalization and rescaling described in Sec. 4.4.2).

Takeaway-1: adaptation at runtime. The sketch can progressively incorporate unseen data as time evolves. During the first transient overload, i.e., before the first time the HPA rescales the workload, around 11:45am, the anomaly score rises with two spikes slightly below the threshold. After the HPA rescales the workload for the first time, the load is uniformly split across the available replicas. Subsequently, the anomaly score doesn't drop to zero but remains significantly lower than before the scaling, indicating that μView has adjusted its classifier to the new data distribution.

Takeaway-2: streamlined configuration effort. μView relieves platform engineers from the burden of setting per-metric thresholds. Recall, with μView we only tune a single anomaly score and not a per-metric configuration. Thanks to the sketch mechanism, μView relies on a single threshold to detect anomalies and raise alarms. For

example, the red curve in Fig. 4.6a highlights that μView detects an anomaly when the cluster is in a persistent overload state.

Takeaway-3: explainability of sketch outputs. We plot the top-3 metrics by magnitude of $\boldsymbol{\alpha}_t$ in Fig. 4.6b (Sec. 4.4.1). We see that the initial spikes before the first workload rescaling were due to metrics that directly relate to the number of open connections, such as `container_sockets`. Instead, the detection of persistent overload can be explained with the metric `kube_pod_status_phase`, since several pods failed to start due to the node having saturated its scheduling capacity. This points out the ability of the sketch to provide explainable outputs, which helps identify culprit metrics despite using a single threshold.

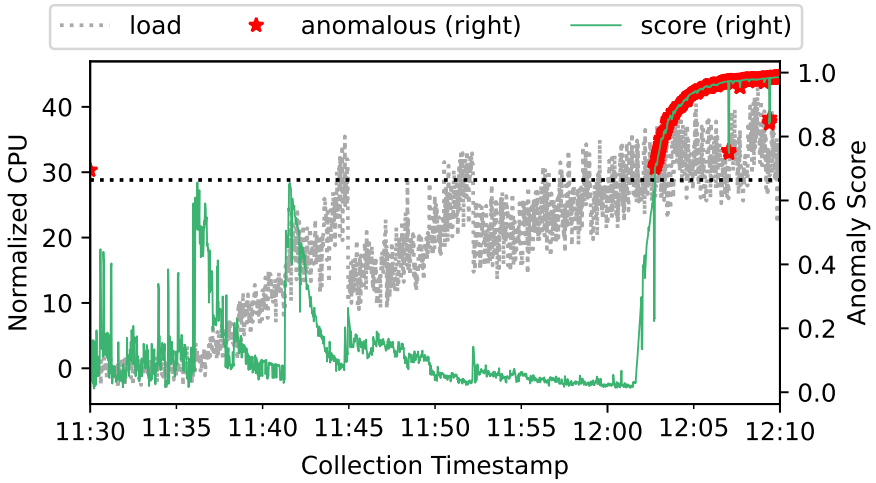
4.8.4 Case study #1: dynamic sampling

Next, we demonstrate how μView can reduce the data volume generated by metric collection while preserving accuracy and thus reducing ingestion and storage costs. For this experiment, we consider a single instance of a Redis microservice and we collect its metrics every second for 40 minutes. We build the training dataset for the sketch using the first 30 minutes and run the dynamic sampler on a test dataset M built on the remaining 10 minutes. Borrowing notation from Sec. 4.4.1, the dataset contains in row M_i the time-series of metric $i = 1,..,m$. Then, we apply the sketch dynamic sampler, as described in Sec. 4.7.2, and generate a filtered time-series $\tilde{M}_i$ with reduced data volume, for every metric time-series M_i. We evaluate the trade-off between accuracy and data volume reduction. As a measure of accuracy, for every metric M_i we consider the Normalized Cross-Correlation (NCC) as defined as:

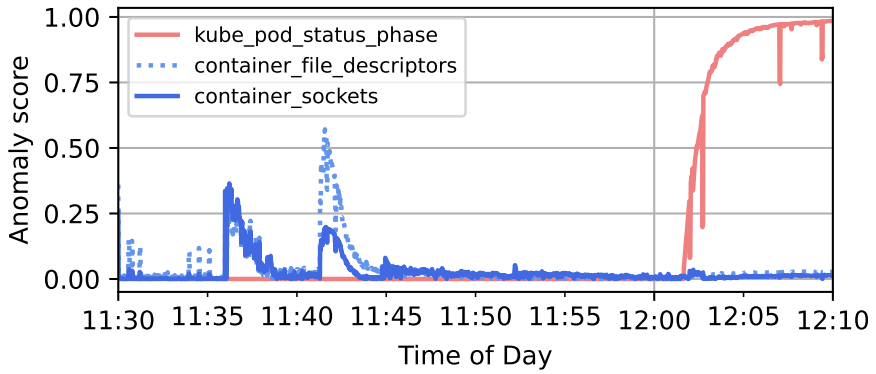
$$NCC_i(\tau = 0) = \frac{corr(M_i, \tilde{M}_i)}{corr(M_i, M_i)}$$

This is because the cross-correlation at a time lag $\tau = 0$ measures the similarity between the two time-series. We normalize it by the auto-correlation of the original time-series, so that the NCC gives a value between 0 and 1. The cross-correlation requires the time-series to have the same length. However, when the dynamic sampler hook drops samples the time-series $\tilde{M}_i$ will have some missing samples. We fill missing samples of $\tilde{M}_i$ by propagating the value of the last collected sample, until the next useful sample. This emulates the behavior of a centralized collector that at any time before the next available collection would only know the last sample. We compute the amount of data volume reduction as the ratio between the number of samples in the filtered metric $\tilde{M}_i$ and the number of samples in the unfiltered metric M_i.

We optimize the sketch hyperparameters according to the guidelines discussed in Sec. 4.8.2, resulting in $k = 15$ and $l = 25$. Figure 4.7 reports the results for different anomaly score thresholds. For dynamic sampling, the anomaly score threshold controls the aggressiveness of the sampler. When the threshold is large, metrics are reported only for significant variations. This results in poor similarity to the original



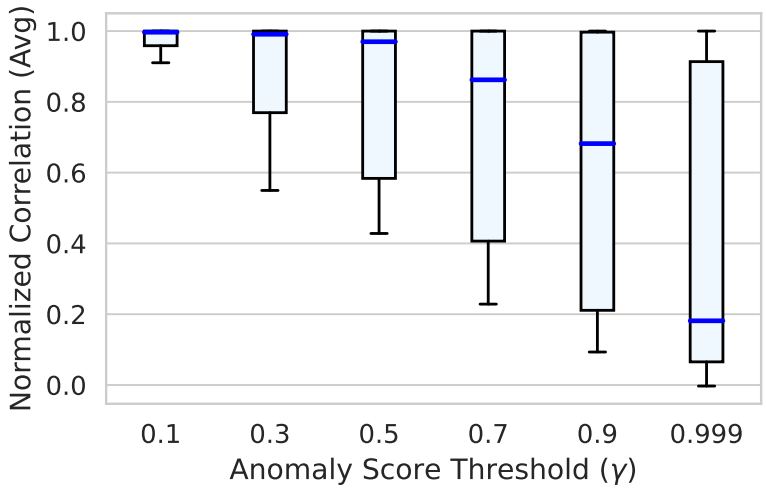
(a) Anomaly score of a `frontend` service. The horizontal line represents the anomaly score threshold.



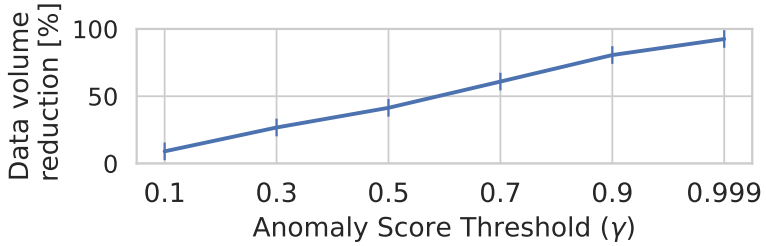
(b) Identification of the top-3 relevant metrics.

Figure 4.6: Evaluation under non-stationary load. The sketch can discriminate between short-term overload, i.e., before the k8s HPA rescales the workload and critical overload conditions.

series, but in a higher data volume reduction. Interestingly we observe that while the data volume reduces linearly (along with the linear increase in cost saving), the accuracy plateaus before dropping. This suggests that by tuning its sensitivity, μView offers a sweet spot that gives a good compromise between accuracy and cost overhead. At threshold 0.5, the dynamic sampler can reduce the ingested data volume by 50% with a less than 5% reduction in the accuracy.



(a) Similarity between filtered and unfiltered metric time-series.



(b) Data volume reduction with respect to unfiltered metric time-series

Figure 4.7: Effectiveness of the dynamic sampler hook with respect to different sensitivities, configurable by the anomaly score threshold.

4.8.5 Case study #2: distributed tracing

In this section, we evaluate the performance of the μView distributed tracing sampling policy (Sec. 4.7.1) in a cluster-wide deployment.

Methodology

We continuously generate user requests from 100 simultaneous users on the OnlineBoutique frontend. We run the experiment for 1 hour, while we collect both metrics and trace datasets. The metrics dataset is used to optimize the sketch hyperparameters (Sec. 4.2.1). After tuning each service's LMAP, we start using the μView distributed tracing sampling policy. on the trace dataset. We run the hyperparameter tuning workflow over the first 35 minutes and evaluate the sampling policy for the different failures over the remaining 25 minutes. Starting after 25 minutes, we inject short failures lasting 30 seconds on a randomly selected service every 2 minutes. This

is because from minute 25 to minute 35 we use the injected anomalies as a testing set for hyperparameter tuning, as explained in Sec. 4.2.1. We evaluate performance in terms of coverage, i.e., the percentage of collected symptomatic traces, and overhead, measured as a percentage of false positives, i.e., the percentage of non-symptomatic traces unnecessarily collected via the tracing hook policy. A trace is defined as symptomatic when one of the RPCs within the trace contains `HTTP/gRPC` error codes or when the trace latency exceeds a predefined value. Since different calls to different frontend operations will generally correspond to traces executing on different critical paths, the definition of this latency value has to depend on the frontend operation. Therefore, to evaluate μView we group traces by frontend operation and derive independent latency threshold values for each group. We use the 99^{th} percentile of the trace latency distribution within each group. Only the traces collected before introducing anomalies are considered for this computation.

μView anticipates symptomatic traces

We compare μView against head-based sampling[147], [180] and an offline *oracle* that employs 100% sampling probability for the entire duration of the injected anomalies. Figure 4.8 shows that tracing with μView achieves nearly total coverage, 5x better than sampling 20% of the request. For more than 70% performance boost, μView adds 5% overhead. This is because head sampling relies on luck to capture faulty traces, while μView is guided by metric signals.

GitLab Redis cache failure [181]. For this experiment, we inject application-level faults inspired by a real-world Redis incident (issue #1601 [181]). In #1601, GitLab experienced chronic latency issues with a Redis instance acting as an LRU cache. Every few minutes, the cache suffered from spikes of very high tail latency (e.g., 1 second), and simultaneously from bursts of key evictions. The problem had been occurring intermittently over two years, necessitating multiple rounds of hypothesis testing and verification before identifying the root cause. The diagnosis unveiled that when the Redis cache is oversubscribed and the Redis instance is close to CPU saturation, the system might enter the following feedback loop. Because of a shared memory pool handling both key storage and I/O buffers, Redis has to evict some keys to accommodate new requests whenever the memory usage reaches `maxmemory` threshold. Because of the single thread design of Redis, the key eviction routine pushes the CPU to saturation, dropping the throughput and increasing the backlog of requests, which in turn creates pressure on Redis memory.

We reproduced the symptom of this Redis issue by synthetically injecting eviction bursts in the Redis cache. For this experiment, we modified the OnlineBoutique application by adding a datastore service downstream to the Redis cache, reproducing a query aside scenario. In our application, if the user cart is not found in the Redis cache, the application has to access a (slower) downstream datastore to retrieve it. To protect the downstream service from overload, the Envoy proxy in Istio service mesh

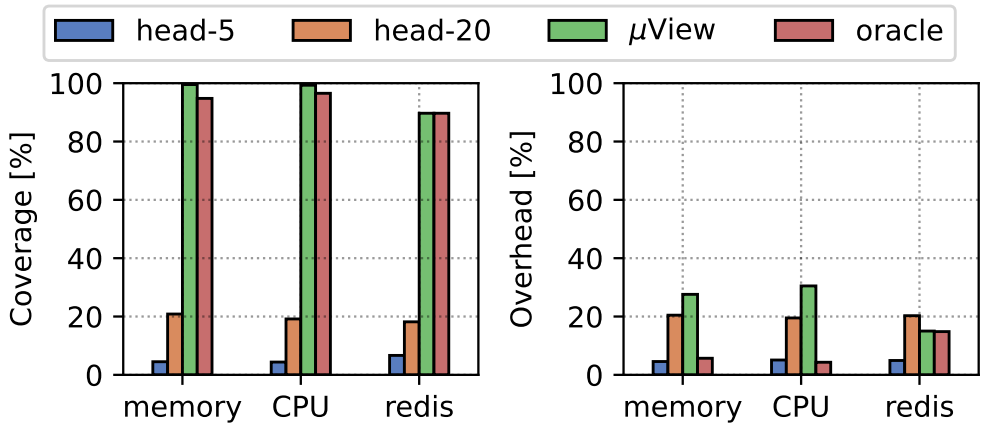


Figure 4.8: Performance-overhead trade-off of μView in assisting distributed tracing for different kind of injected anomalies.

had been configured to enforce local rate limiting between the Redis cache and the db service. The motivation for this design is that the cart associated with an active user session is reasonably assumed to hit the cache. This is because we renew the Redis expiration time for keys associated with active users with a timeout of a few minutes, and the Redis eviction policy is configured to `expiring-LRU`. However, during the cache incident, a significant amount of user requests missed the cache and had to retrieve their session data from the backend database, which in turn caused the rate limiter to drop requests. μView contributed to our troubleshooting efforts in two ways. First, because of its real-time metrics streaming capabilities, we could exactly pinpoint in time when the eviction burst took place. Based on this input we could restrict trace analysis to a specific time window. Second, over the selected time window, the Jaeger monitoring tool was able to capture a high percentage of requests ended with `HTTP 429` between the Redis cache and the database, which revealed the rate limiter as the root cause of request drops.

4.9 Related work

Troubleshooting cloud applications. There is significant work on troubleshooting microservices often focused on using sampled data (trace [61], [142], [147], [182]–[184], logs [185]–[187], metrics [60], [145]), few have considered non-centralized processing of data. Notably, Fay [188] and recently OpenTelemetry [189] proposed hierarchical approaches to reduce data sent to the central data store; however, they focus on providing simple aggregation and still impose CPU overheads on the servers. Instead, μView offloads processing to IPU freeing CPU cycles and provides a flexible and general sketch for anomaly detection.

Sketches for network monitoring. Sketches [29], [190]–[193] have long been explored as a fundamental primitive for network telemetry over sampling due to their lightweight and approximate nature. Unfortunately, due to network flow characteristics and hardware limitations, network telemetry focuses on counting sketches, e.g., count-min, whereas μView explores a distinctly different sketch optimized for online anomaly detection and μView focuses on optimizing metrics transfer via RDMA which network telemetry does not address.

Offloading to SmartNIC. Recent work has demonstrated the benefits of offloading to SmartNICs for network packet processing [194]–[196], accelerating key-value stores [197], distributed file systems and transactions [198]–[200], GPU-centric applications [201], and even microservices [202]. μView differs in its application of offloading for observability and introduces RDMA mechanisms to reduce overheads of data-sharing.

4.10 Discussion

In this chapter, we presented μView, a sketch-based system to improve observability that preserves monitoring accuracy while reducing cost and performance overheads. We analyzed the effectiveness of the system under dynamic conditions, demonstrating its ability to distinguish between transient and persistent overload situations, while also offering significant data volume reductions. μView reduces operational burden through the use of a catch-all anomaly score threshold, improves explainability by highlighting relevant metrics that contribute to anomalies, and enables substantial cost savings of up to 50% with negligible loss in measurement accuracy. μView can increase the coverage of faulty requests by 5x compared to OpenTelemetry sampling policies, while keeping the overhead low.

Chapter 5

Conclusion

In this thesis we took a step towards the realization of data center monitoring solutions with granular visibility and high accuracy, which is prominent in today's era of distributed systems and Anything-as-a-Service cloud model. DC operators need monitoring for informed decision-making to plan the DC evolution, and, as network provisioners, to timely respond to network incidents when they happen or witness with confidence about the innocence of their network else wise. Similarly, tenants strives for more and more monitoring intelligence, to better understand the performance of their applications and services. Monitoring is conspicuously complicated by the large scale disaggregated design of DC systems, which has brought to intensifying the capillarity and the volume of telemetry to cope with the increased exposure to failures and performance degradation. Unfortunately, this is hardly sustainable on the long term due to the huge overheads.

Network programmability, started with the SDN process with the goal of introducing additional flexibility and vendor-independence for network protocols, has resulted in the ubiquity of programmable data plane technologies both in the network switches and at the hosts NICs. These technologies have soon extended beyond mere packet processing, and nowadays serve as accelerators for diverse applications (e.g., distributed ML training [12]), thanks to its ability to support high-throughput computations. Monitoring is one of the applications that can benefit from this trend, as it requires processing data generated at high throughput and to continuously extract statistics and features, all of which can be done on-the-fly in network devices while data are transiting.

In our work, we have explored the potential of leveraging programmable data planes to address the DC monitoring challenges for two target categories: network traffic anomaly detection and cloud-native applications observability. All the included contributions are based on the rationale of moving monitoring functionalities close to where telemetry data are generated. Chapters 2 and 3 have focused on aspects of network traffic monitoring, and devised sketches to compute real-time traffic statistics

in P4 switches, while Chapter 4 has focused on the monitoring of cloud-native applications, demonstrating for the first time a system which uses sketches placed on a SmartNIC to reduce the observability data bloat. We believe that our work, which embraced the trend of partitioning and offloading functionalities between the servers CPUs and programmable network devices, is actually part of the broader inclination of data centers towards platform accelerators (IPUs, TPUs, GPUs, FPGAs, etc.), which today represents the key enabling factor to speedup performance.

In Chapter 2, we addressed the challenging task of cardinality estimation in traffic streams, *a.k.a.* counting unique flows. The main focus of our investigation has been extending state-of-the-art sketches to count flows in continuous-time and under a sliding window model. The outcomes of our research are two new sketches to be used in practical monitoring scenarios, where recent traffic statistics are what really matters to network management.

ST-HLL is an extension of the HLL sketch, and embeds a structural mechanism to forget outdated counted arrivals, which is based on the staggering of the internal structure of the sketch. The main strength of this sketch is that it does not add any operational complexity to vanilla HLL, while also keeping the same memory footprint. Moreover, we believe that the staggering approach is general enough to be revisited for other sketches sharing a similar architecture. In this chapter, we formulated an extensive statistical analysis to characterize the behavior of a register estimator under the staggering scheme, then we devised practical algorithmic solutions. We developed ST-HLL with simplicity and practicality in mind: it is the first timestamp-free sketch for cardinality estimation, which can be implemented just by using a single timeout mechanism available in commercial switches. Nevertheless, performance-wise it improves up to a factor $2x$ the estimation accuracy over alternative methods [85], for a given memory budget under real-world Internet traffic traces.

The second sketch, TS-PCSA, is an extension of the PCSA sketch, and supports sliding window counting through timestamp-tagged entries. The main novelty of this contribution is the simple yet effective strategy to associate a constant temporal offset to the sketch registers, which allows to keep the sketch as lightweight as previous techniques, while being up to 25% more accurate. We evaluated TS-PCSA under realistic settings, and we showed that it outperforms other solutions by 25%, given a fixed memory budget.

We concluded that despite the vast body of research around sketches for network traffic monitoring, when it comes to continuous-time operations only very few sketches can make cardinality estimation practical. Our work has shown how this goal is actually achievable, yet it has also emphasized that the amount of memory available on a single switch plays a central role for the performance of the estimator: if we could afford larger footprint TS-PCSA outperforming ST-HLL.

Following the outcome of Chapter 2, we directed our efforts towards finding mechanisms to distribute sketches across multiple switches and use the entire memory resources available network-wide. In Chapter 3, we looked into existing approaches to realize such abstraction, and we found that the current state-of-the-art for the frequency counting problem is not able to fully exploit the potential of distributed sketches. We showed the suboptimality of the approach via numerical simulations, demonstrating that traffic unbalancing may negatively affect the performance. Moreover, we highlighted the convexity of the problem, which suggested that a global optimization is possible and opened to new research avenues. Ultimately our findings proved that a new approach is needed to fully exploit the potential of distributed sketches, and it should account for the diverse traffic load condition expected on the sketch fragments. A very recent work [11] has followed-up on this direction, and proposed a new sketch that is able to cope with traffic-awareness, coherently with the prospects of our work.

In Chapter 2 and Chapter 3 we have looked to monitoring from the perspective of the DC, who is responsible for smooth network operations. The other side of the coin is the monitoring of the applications running on the DC, which is instead relevant to DC customers and the focus of Chapter 4. We have proposed a novel system, μView, to reduce the observability data bloat in cloud-native applications. The system is based on the use of sketches placed on an IPU (i.e., BlueField2 SmartNIC) to reduce the communication and storage costs of observability.

We believe our work in this field is especially relevant as part of a broader trend in data centers. Offloading the workloads from server's CPUs to external specialized accelerators, such as GPUs, IPUs, TPUs and FPGA is the current biggest hype in data centers and actually represents, after the end of the Moore's law, the key factor to boost performance. A pivotal challenge is choosing which functionalities to offload and where to place them, with the goal of optimizing for several indicators, such as latency, throughput, and energy consumption. Our work has identified in observability a good candidate to be offloaded, as it is largely comprised by routine tasks, including data pre-processing and filtering, that can be performed by lightweight algorithms and easily parallelized. At the same time, we have found SmartNICs as a natural fit for offloading observability functionalities, as they are designed with core numerosity in mind. Moreover, observability oftentimes only involve server-local measurements which can be easily accessed at high frequency via RDMA.

We validated μView on a production-grade microservice application in a self-managed on-premises Kubernetes cluster. Our results showed that thanks to the sketch mechanism acting as an intermediate filter, we can reduce the observability data bloat caused by metrics, with negligible loss in monitoring accuracy. We have also shown that μView can assist distributed tracing and increase the coverage of faulty requests with respect to industrial standards such as OpenTelemetry and Jaeger.

In conclusion, in this thesis we proposed a set of novel monitoring tools applicable to numerous elements of modern DCN infrastructure. We showed that estimating flow cardinality from high-speed traffic, while being able to answer to sliding window queries at arbitrarily points in time, can be achieved easily without introducing complexity in the network node. We showed that even if multiple physical sketches distributed across several nodes can contribute to a shared logical sketch and lead to better accuracy, doing so without optimizing the division of labor for the specific traffic patterns can be myopic and suboptimal in numerous scenarios. Finally, we showed that sketch-based approaches combined with off-path IPUs can reduce the observability data bloat in cloud-native applications, which is a major impairment, other than a financial pain, for enterprises adopting cloud-native.

Glossary

ARE

Average Relative Error. 60, 61

CMS

Count-Min Sketch. 57, 58

CNCF

Cloud-Native Computing Foundation. 12

CT

continuous-time. 22

DC

Data Center. 1, 95, 97

DCN

Data Center Network. 2, 98

DDoS

Distributed Denial of Service. 16, 17, 21, 52, 58

HLL

HyperLogLog. iii, 22–26, 28–38, 41, 43–47, 52, 53, 96

IPU

Infrastructure Processing Unit. ix, 4, 16, 66, 67, 73, 75, 78–81, 92, 96–98

k8s

Kubernetes. 13, 69, 71, 75, 83–86, 89

LMAP

Local Metrics Analysis Pipeline. 71–75, 78, 80–83, 86, 87, 90

LPFM

List of Possible Future Maxima. 28, 47

NICs

Network Interface Cards. ix, 2, 16, 65–68, 70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 95

PCSA

Probabilistic Counting with Stochastic Averaging. 18, 22–24, 37, 41, 96

PISA

Protocol Independent Switch Architecture. iii, 3, 4, 62

RDMA

Remote Direct Memory Access. 5, 97

SDN

Software Defined Networking. 2, 4, 7, 55, 95

ST-HLL

Staggered HyperLogLog. viii, 18, 22, 23, 28, 29, 31–37, 40, 41, 43–48, 50–53, 96

TNA

Tofino Native Architecture. 3, 4

TS-PCSA

TimeStamp-augmented PCSA. 18, 22, 23, 37–44, 48–51, 53, 96

Bibliography

[1] N. McKeown, T. Anderson, H. Balakrishnan, *et al.*, "OpenFlow: Enabling Innovation in Campus Networks", *SIGCOMM Computer Communication Review*, vol. 38, no. 2, 2008.

[2] Intel. (2021). Tofino Native Architecture – OpenTofino, [Online]. Available: https://github.com/barefootnetworks/Open-Tofino/blob/master/PUBLIC_Tofino-Native-Arch.pdf (visited on 02/27/2024).

[3] P. Bosshart, G. Gibb, H.-S. Kim, *et al.*, "Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN", in *Proceedings of the ACM SIGCOMM Conference*, Association for Computing Machinery, 2013.

[4] P. Bosshart, D. Daly, G. Gibb, *et al.*, "P4: Programming Protocol-independent Packet Processors", *SIGCOMM Computer Communication Review*, vol. 44, no. 3, 2014.

[5] Community. (2024). Behavioral Model version 2 (bmv2). The reference P4 software switch, [Online]. Available: https://github.com/p4lang/behavioral-model (visited on 02/27/2024).

[6] P. Vörös, D. Horpácsi, R. Kitlei, D. Leskó, M. Tejfel, and S. Laki, "T4P4S: A Target-independent Compiler for Protocol-independent Packet Processors", in *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, 2018.

[7] Community. (2023). P4-DPDK target, [Online]. Available: https://github.com/p4lang/p4-dpdk-target (visited on 02/27/2024).

[8] Intel. (2024). Intel Tofino Series, [Online]. Available: https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors/tofino.html (visited on 02/27/2024).

[9] Community. (2018). P4 on NetFPGA – public repository, [Online]. Available: https://github.com/NetFPGA/P4-NetFPGA-public/wiki (visited on 02/27/2024).

[10] F. Matus. (2020). Pensando – Distributed Services Architecture, [Online]. Available: https://hc32.hotchips.org/assets/program/conference/day2/HotChips2020_Networking_Pensando_v3.pdf (visited on 02/27/2024).

[11] L. Gu, Y. Tian, W. Chen, Z. Wei, C. Wang, and X. Zhang, "Per-Flow Network Measurement With Distributed Sketch", *IEEE/ACM Transactions on Networking*, vol. 32, no. 1, 2024.

[12] A. Sapio, M. Canini, C.-Y. Ho, *et al.*, "Scaling Distributed Machine Learning with In-Network Aggregation", in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, USENIX Association, 2021.

[13] A. Singh, J. Ong, A. Agarwal, *et al.*, "Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network", in *Proceedings of the ACM SIGCOMM Conference*, Association for Computing Machinery, 2015.

[14] M. A. Qureshi, J. Yan, Y. Cheng, *et al.*, "Fathom: Understanding Datacenter Application Network Performance", in *Proceedings of the ACM SIGCOMM Conference*, Association for Computing Machinery, 2023.

[15] K. Gao, C. Sun, S. Wang, *et al.*, "Buffer-based End-to-end Request Event Monitoring in the Cloud", in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, USENIX Association, 2022.

[16] Y. Zhou, C. Sun, H. H. Liu, *et al.*, "Flow Event Telemetry on Programmable Data Plane", in *Proceedings of the ACM SIGCOMM Conference*, Association for Computing Machinery, 2020.

[17] N. Teams. (2023). Nagios – Switch monitoring, [Online]. Available: https://www.nagios.com/solutions/switch-monitoring/ (visited on 02/27/2024).

[18] Y. Li, J. Gong, and M. Yu. (2021). Cisco NetFlow, [Online]. Available: https://www.ietf.org/rfc/rfc3954.txt (visited on 02/27/2024).

[19] P. Phaal, S. Panchen, and N. McKee, "InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks", RFC 3176, 2001. [Online]. Available: https://dl.acm.org/doi/pdf/10.17487/RFC3176.

[20] Y. Zhu, N. Kang, J. Cao, *et al.*, "Packet-Level Telemetry in Large Datacenter Networks", in *Proceedings of the ACM SIGCOMM Conference*, Association for Computing Machinery, 2015.

[21] V. Sekar, M. K. Reiter, W. Willinger, H. Zhang, R. R. Kompella, and D. G. Andersen, "CSAMP: a system for network-wide flow monitoring", in *5th USENIX Symposium on Networked Systems Design and Implementation (NSDI 08)*, USENIX Association, 2008.

[22] C. Kim, A. Sivaraman, N. Katta, A. Bas, A. Dixit, and L. J. Wobker. (2015). In-Band Network Telemetry via Programmable Dataplanes, [Online]. Available: https://anirudhsk.github.io/papers/int-demo.pdf (visited on 02/27/2024).

[23] R. Ben Basat, S. Ramanathan, Y. Li, G. Antichi, M. Yu, and M. Mitzenmacher, "PINT: Probabilistic In-band Network Telemetry", in *Proceedings of the ACM SIGCOMM Conference*, Association for Computing Machinery, 2020.

[24] F. Brockners, S. Bhandari, and T. Mizrahi, "Data Fields for In Situ Operations, Administration, and Maintenance (IOAM)", RFC 9197, 2022. [Online]. Available: https://www.rfc-editor.org/info/rfc9197.

[25] T. Mizrahi, G. Navon, G. Fioccola, M. Cociglio, M. Chen, and G. Mirsky, "AM-PM: Efficient Network Telemetry using Alternate Marking", *IEEE Network*, vol. 33, no. 4, 2019.

[26] J. Sonchack, O. Michel, A. J. Aviv, E. Keller, and J. M. Smith, "Scaling Hardware Accelerated Network Monitoring to Concurrent and Dynamic Queries With *Flow", in *USENIX Annual Technical Conference (ATC)*, USENIX Association, 2018.

[27] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown, "I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks", in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, USENIX Association, 2014.

[28] R. Teixeira, R. Harrison, A. Gupta, and J. Rexford, "PacketScope: Monitoring the Packet Lifecycle Inside a Switch", in *Proceedings of the Symposium on SDN Research*, Association for Computing Machinery, 2020.

[29] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger, "Sonata: Query-Driven Streaming Network Telemetry", in *Proceedings of the ACM SIGCOMM Conference*, Association for Computing Machinery, 2018.

[30] K. Gao, D. Li, and S. Wang, "Bandwidth-efficient Microburst Measurement in Large-scale Datacenter Networks", in *Proceedings of the 6th ACM APNet Conference*, Association for Computing Machinery, 2023.

[31] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, "Heavy-Hitter Detection Entirely in the Data Plane", in *Proceedings of the ACM Symposium on SDN Research (SOSR)*, Association for Computing Machinery, 2017.

[32] N. Ivkin, Z. Yu, V. Braverman, and X. Jin, "QPipe: quantiles sketch fully in the data plane", in *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, Association for Computing Machinery, 2019.

[33] X. Chen, S. L. Feibish, Y. Koral, *et al.*, "Fine-grained Queue Measurement in The Data Plane", in *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, Association for Computing Machinery, 2019.

[34] X. Chen, S. Landau-Feibish, M. Braverman, and J. Rexford, "BeauCoup: Answering Many Network Traffic Queries, One Memory Update at a Time", in *Proceedings of the ACM SIGCOMM Conference*, Association for Computing Machinery, 2020.

[35] T. Yang, J. Jiang, P. Liu, *et al.*, "Elastic Sketch: Adaptive and Fast Network-wide Measurements", in *Conference of the ACM Special Interest Group on Data Communication*, 2018.

[36] C. Community. (2013). NetFlow – white paper, [Online]. Available: https://community.cisco.com/t5/service-providers-knowledge-base/asr9k-netflow-white-paper/ta-p/3145878 (visited on 02/27/2024).

[37] Y. Li, R. Miao, C. Kim, and M. Yu, "FlowRadar: A Better NetFlow for Data Centers", in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, 2016.

[38] Cisco. (2018). Cisco ERSPAN, [Online]. Available: https://www.cisco.com/c/en/us/td/docs/switches/lan/catalyst9400/software/release/16-10/configuration_guide/nmgmt/b_1610_nmgmt_9400_cg/configuring_erspan.html (visited on 02/27/2024).

[39] P. A. W. Group. (2020). In-band Network Telemetry (INT) dataplane specifications v2.1, [Online]. Available: https://p4.org/p4-spec/docs/INT_v2_1.pdf (visited on 02/27/2024).

[40] J. Langlet, R. Ben Basat, G. Oliaro, M. Mitzenmacher, M. Yu, and G. Antichi, "Direct Telemetry Access", in *Proceedings of the ACM SIGCOMM Conference*, Association for Computing Machinery, 2023.

[41] C. Guo, L. Yuan, D. Xiang, *et al.*, "Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis", in *Proceedings of the ACM SIGCOMM Conference*, Association for Computing Machinery, 2015.

[42] P. Lapukhov and A. Adams. (2016). NetNORAD: troubleshooting networks via end-to-end probing, [Online]. Available: https://engineering.fb.com/2016/02/18/core-infra/netnorad-troubleshooting-networks-via-end-to-end-probing/ (visited on 02/27/2024).

[43] M. Ghasemi, T. Benson, and J. Rexford, "Dapper: Data Plane Performance Diagnosis of TCP", in *Proceedings of the Symposium on SDN Research (SOSR)*, Association for Computing Machinery, 2017.

[44] J. Zhu, K. Zhang, and Q. Huang, "A Sketch Algorithm to Monitor High Packet Delay in Network Traffic", in *Proceedings of the 5th ACM APNet Conference*, Association for Computing Machinery, 2022.

[45] Y. Zhao, K. Yang, Z. Liu, *et al.*, "LightGuardian: A Full-Visibility, Lightweight, In-band Telemetry System Using Sketchlets", in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, USENIX Association, 2021.

[46] Q. Huang, S. Sheng, X. Chen, *et al.*, "Toward Nearly-Zero-Error Sketching via Compressive Sensing", in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, USENIX Association, 2021.

[47] CNCF. (2024). CNCF – Cloud Native Computing Foundation, [Online]. Available: https://www.cncf.io/ (visited on 02/27/2024).

[48] F5. (2024). NGINX, [Online]. Available: https://www.nginx.com/ (visited on 02/27/2024).

[49] MongoDB. (2024). MongoDB, [Online]. Available: https://www.mongodb.com/ (visited on 02/27/2024).

[50] T. K. Authors. (2024). Kubernetes, [Online]. Available: https://kubernetes.io/ (visited on 02/27/2024).

[51] N. Enterprises. (2024). Nagios, [Online]. Available: https://www.nagios.org/ (visited on 02/27/2024).

[52] CNCF. (2024). CNCF Glossary – Observability, [Online]. Available: https://glossary.cncf.io/observability/ (visited on 02/27/2024).

[53] Honeycomb. (2024). Honeycomb – Observability for distributed services, [Online]. Available: https://www.honeycomb.io/ (visited on 02/27/2024).

[54] ServiceNow. (2024). ServiceNow (formerly Lightstep), [Online]. Available: https://www.servicenow.com/products/observability.html (visited on 02/27/2024).

[55] Datadog. (2024). Datadog, [Online]. Available: https://www.datadoghq.com (visited on 02/27/2024).

[56] D. LLC. (2024). Dynatrace, [Online]. Available: https://www.dynatrace.com/ (visited on 02/27/2024).

[57] A. S. Foundation. (2024). Apache skywalking, [Online]. Available: https://skywalking.apache.org/ (visited on 02/27/2024).

[58] Community. (2024). Istio, [Online]. Available: https://istio.io/ (visited on 02/27/2024).

[59] Isovalent. (2024). Cilium – Service Mesh, [Online]. Available: https://cilium.io/use-cases/service-mesh/ (visited on 02/27/2024).

[60] J. Thalheim, A. Rodrigues, I. E. Akkus, *et al.*, "Sieve: Actionable insights from monitored metrics in distributed systems", in *ACM/IFIP/USENIX Middleware Conference*, 2017.

[61] .

[62] Community. (2024). Jaeger, [Online]. Available: https://www.jaegertracing.io/ (visited on 02/27/2024).

[63] Community. (2024). Zipkin, [Online]. Available: https://zipkin.io/ (visited on 02/27/2024).

[64] Redis. (2024). Redis, [Online]. Available: https://redis.io/ (visited on 02/27/2024).

[65] Y. Gan, M. Liang, S. Dev, D. Lo, and C. Delimitrou, "Sage: practical and scalable ML-driven performance debugging in microservices", Association for Computing Machinery, 2021.

[66] Z. Wang, T. Ma, L. Kong, *et al.*, "Zero Overhead Monitoring for Cloud-native Infrastructure using RDMA", in *USENIX Annual Technical Conference (ATC 22)*, 2022.

[67] Y. Gan, Y. Zhang, K. Hu, *et al.*, "Seer: Leveraging Big Data to Navigate the Complexity of Performance Debugging in Cloud Microservices", in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Association for Computing Machinery, 2019.

[68] P. Flajolet and G. N. Martin, "Probabilistic counting algorithms for data base applications", *Journal of computer and system sciences*, vol. 31, no. 2, 1985.

[69] R. Skillington. (2018). M3: Uber's Open Source, Large-scale Metrics Platform for Prometheus, [Online]. Available: https://www.uber.com/en-PT/blog/m3/ (visited on 02/27/2024).

[70] A. W. Services. (2024). AWS – Prometheus, [Online]. Available: https://aws.amazon.com/it/prometheus/ (visited on 02/27/2024).

[71] A. Cornacchia, G. Bianchi, A. Bianco, and P. Giaccone, "Staggered HLL: Near-continuous-time cardinality estimation with no overhead", *Computer Communications*, vol. 193, 2022.

[72] A. Cornacchia, G. Bianchi, A. Bianco, and P. Giaccone, "Designing Probabilistic Flow Counting over Sliding Windows", in *2022 IEEE 11th IFIP International Conference on Performance Evaluation and Modeling in Wireless and Wired Networks (PEMWN)*, 2022.

[73] V. Bruschi, S. Pontarelli, J. Tollet, D. Barach, and G. Bianchi, "FlowFight: High performance–low memory top-k spreader detection", *Computer Networks*, vol. 196, 2021.

[74] Y. Chabchoub, R. Chiky, and B. Dogan, "How can sliding HyperLogLog and EWMA detect port scan attacks in IP traffic?", *EURASIP Journal on Information Security*, vol. 2014, no. 1, 2014.

[75] P. García-Teodoro, J. Díaz-Verdejo, G. Maciá-Fernández, and E. Vázquez, "Anomaly-based network intrusion detection: Techniques, systems and challenges", *Computers and Security*, vol. 28, no. 1, 2009.

[76] Y. Liu, W. Chen, and Y. Guan, "Identifying high-cardinality hosts from network-wide traffic measurements", *IEEE Transactions on Dependable and Secure Computing*, vol. 13, no. 5, 2015.

[77] H.-A. Kim and D. O'Hallaron, "Counting network flows in real time", in *IEEE Global Communications Conference (GLOBECOM)*, IEEE, 2003.

[78] M. Durand and P. Flajolet, "Loglog counting of large cardinalities", in *European Symposium on Algorithms (ESA)*, Springer, 2003.

[79] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier, "HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm", in *Conference on Analysis of Algorithms (AofA)*, 2007.

[80] F. Giroire, "Order Statistics and Estimating Cardinalities of Massive Data Sets", *Discrete Applied Mathematics*, vol. 157, no. 2, 2009.

[81] G. Bianchi, N. d'Heureuse, and S. Niccolini, "On-demand time-decaying bloom filters for telemarketer detection", *SIGCOMM Computer Communication Review*, vol. 41, no. 5, 2011.

[82] R. Ben-Basat, G. Einziger, R. Friedman, and Y. Kassner, "Heavy hitters in streams and sliding windows", in *IEEE International Conference on Computer Communications (INFOCOM)*, IEEE, 2016.

[83] R. B. Basat, G. Einziger, and R. Friedman, "Give me some slack: Efficient network measurements", *Theoretical Computer Science*, vol. 791, 2019.

[84] X. Gou, Y. Zhang, Z. Hu, *et al.*, "A Sketch Framework for Approximate Data Stream Processing in Sliding Windows", *IEEE Transactions on Knowledge and Data Engineering*, 2022.

[85] Y. Chabchoub and G. Heébrail, "Sliding HyperLogLog: Estimating cardinality in a data stream over a sliding window", in *International Conference on Data Mining Workshops*, 2010.

[86] E. Assaf, R. B. Basat, G. Einziger, and R. Friedman, "Pay for a sliding bloom filter and get counting, distinct elements, and entropy for free", in *IEEE International Conference on Computer Communications (INFOCOM)*, 2018.

[87] N. Ivkin, R. B. Basat, Z. Liu, G. Einziger, R. Friedman, and V. Braverman, "I know what you did last summer: Network monitoring using interval queries", *Measurement and Analysis of Computing Systems*, vol. 3, no. 3, 2019.

[88] K.-Y. Whang, B. T. Vander-Zanden, and H. M. Taylor, "A linear-time probabilistic counting algorithm for database applications", *ACM Transactions on Database Systems (TODS)*, vol. 15, no. 2, 1990.

[89] C. Estan, G. Varghese, and M. Fisk, "Bitmap Algorithms for Counting Active Flows on High-Speed Links", *IEEE/ACM Transactions on Networking*, vol. 14, no. 5, 2006.

[90] Z. Bar-Yossef, T. S. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan, "Counting Distinct Elements in a Data Stream", in *International Workshop on Randomization and Approximation Techniques*, Springer-Verlag, 2002.

[91] F. Giroire, "Réseaux, algorithmique et analyse combinatoire de grands ensembles", PhD thesis, Paris 6, 2006.

[92] C. Estan, G. Varghese, and M. Fisk, "Bitmap algorithms for counting active flows on high speed links", in *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, 2003.

[93] S. Heule, M. Nunkesser, and A. Hall, "Hyperloglog in practice: Algorithmic engineering of a state of the art cardinality estimation algorithm", in *Proceedings of the 16th International Conference on Extending Database Technology*, 2013.

[94] A. Metwally, D. Agrawal, and A. E. Abbadi, "Why go logarithmic if we can go linear? Towards effective distinct counting of search traffic", in *International conference on Advances in database technology (EDBT)*, 2008.

[95] H. Harmouch and F. Naumann, "Cardinality Estimation: An Experimental Survey", *Proceedings of the VLDB Endowment*, vol. 11, no. 4, 2017.

[96] E. Bisong, "Google BigQuery", in *Building Machine Learning and Deep Learning Models on Google Cloud Platform: A Comprehensive Guide for Beginners*. Apress, 2019.

[97] Microsoft. (2024). Kusto Query Language (KQL) overview, [Online]. Available: https://docs.microsoft.com/en-us/azure/data-explorer/kusto/query/ (visited on 02/27/2024).

[98] R. Sethi, M. Traverso, D. Sundstrom, *et al.*, "Presto: SQL on everything", in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, 2019.

[99] É. Fusy and F. Giroire, "Estimating the number of active flows in a data stream over a sliding window", in *Proceedings of the Meeting on Analytic Algorithmics and Combinatorics*, Society for Industrial and Applied Mathematics, 2007.

[100] R. Ben-Basat, X. Chen, G. Einziger, and O. Rottenstreich, "Efficient measurement on programmable switches using probabilistic recirculation", in *IEEE International Conference on Network Protocols (ICNP)*, IEEE, 2018.

[101] W. Szpankowski and V. Rego, "Yet another application of a binomial recurrence order statistics", *Computing*, vol. 43, no. 4, 1990.

[102] (). GitHub repository, [Online]. Available: https://github.com/alessandrocornacchia/Stag-HLL.git (visited on 02/27/2024).

[103] CAIDA.org. (2018). CAIDA 2018, [Online]. Available: https://www.caida.org/catalog/datasets/trace_stats/nyc-a/2018/equinix-nyc.dira.20180517-130000.utc.df.txt (visited on 02/27/2024).

[104] CAIDA.org. (2019). CAIDA 2019, [Online]. Available: https://www.caida.org/catalog/datasets/trace_stats/nyc-a/2019/equinix-nyc.dira.20190117-130000.utc.df.txt (visited on 02/27/2024).

[105] R. B. Basat, G. Einziger, I. Keslassy, A. Orda, S. Vargaftik, and E. Waisbard, "Memento: Making sliding windows efficient for heavy hitters", in *CoNEXT*, 2018.

[106] B. Turkovic, J. Oostenbrink, F. Kuipers, I. Keslassy, and A. Orda, "Sequential zeroing: Online heavy-hitter detection on programmable hardware", in *IFIP*, 2020.

[107] G. Einziger and R. Friedman, "Counting with tinytable: Every bit counts!", in *Proceedings of the 17th International Conference on Distributed Computing and Networking*, 2016.

[108] A. Shrivastava, A. C. Konig, and M. Bilenko, "Time adaptive sketches (ada-sketches) for summarizing data streams", in *SIGMOD*, 2016.

[109] G. Cormode, V. Shkapenyuk, D. Srivastava, and B. Xu, "Forward decay: A practical time decay model for streaming systems", in *International conference on data engineering*, 2009.

[110] Y. Qiu, K.-F. Hsu, J. Xing, and A. Chen, "A feasibility study on time-aware monitoring with commodity switches", in *Proceedings of the Workshop on Secure Programmable Network Infrastructure*, 2020.

[111] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications", *Journal of Algorithms*, vol. 55, no. 1, 2005.

[112] A. Cornacchia, G. Sviridov, P. Giaccone, and A. Bianco, "A Traffic-Aware Perspective on Network Disaggregated Sketches", in *2021 19th Mediterranean Communication and Computer Networking Conference (MedComNet)*, 2021.

[113] M. Trevisan, D. Giordano, I. Drago, M. M. Munafò, and M. Mellia, "Five Years at the Edge: Watching Internet From the ISP Network", *IEEE/ACM Transactions on Networking*, vol. 28, no. 2, 2020.

[114] C. Estan and G. Varghese, "New directions in traffic measurement and accounting", in *Proceedings of the Conference on Applications, technologies, architectures, and protocols for computer communications*, 2002.

[115] M. Yu, L. Jose, and R. Miao, "Software Defined Traffic Measurement with OpenSketch", in *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013.

[116] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, "Scream: Sketch resource allocation for software-defined measurement", in *ACM Conference on Emerging Networking Experiments and Technologies*, 2015.

[117] S. Li, L. Luo, and D. Guo, "Sketch for traffic measurement: design, optimization, application and implementation", *arXiv preprint arXiv:2012.07214*, 2020.

[118] V. Bruschi, R. B. Basat, Z. Liu, G. Antichi, G. Bianchi, and M. Mitzenmacher, "DISCOvering the heavy hitters with disaggregated sketches", in *International Conference on emerging Networking EXperiments and Technologies*, 2020.

[119] Q. Huang, P. P. Lee, and Y. Bao, "Sketchlearn: relieving user burdens in approximate measurement with automated statistical inference", in *Conference of the ACM Special Interest Group on Data Communication*, 2018.

[120] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild", in *Proceedings of the ACM/USENIX Internet Measurement Conference*, Association for Computing Machinery, 2010.

[121] In.

[122] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network", in *ACM Conference on Special Interest Group on Data Communication*, 2015.

[123] T. Yang, L. Wang, Y. Shen, *et al.*, "Empowering sketches with machine learning for network measurements", in *Workshop on Network Meets AI & ML*, 2018.

[124] C. H. Song, P. G. Kannan, B. K. H. Low, and M. C. Chan, "FCM-sketch: generic network measurements with data plane support", in *International Conference on emerging Networking EXperiments and Technologies*, 2020.

[125] A. Cornacchia, T. A. Benson, M. Bilal, and M. Canini, "MicroView: Cloud-Native Observability with Temporal Precision", in *Proceedings of the CoNEXT Student Workshop*, ACM, 2023.

[126] D. Merkel, "Docker: lightweight Linux containers for consistent development and deployment", *Linux Journal*, vol. 2014, no. 239, 2014.

[127] A. Sriraman and T. F. Wenisch, "μ Suite: a benchmark suite for microservices", in *IEEE International Symposium on Workload Characterization (IISWC)*, 2018.

[128] Y. Gan and C. Delimitrou, "The architectural implications of cloud microservices", *IEEE Computer Architecture Letters*, vol. 17, no. 2, 2018.

[129] J. Cloud. (2013). Decomposing Twitter: Adventures in Service-Oriented Architecture, [Online]. Available: https://www.infoq.com/presentations/twitter-soa/ (visited on 02/27/2024).

[130] A. Gluck. (2020). Introducing Domain-Oriented Microservice Architecture, [Online]. Available: https://www.uber.com/en-PT/blog/microservice-architecture/ (visited on 02/27/2024).

[131] E. Eiswerth. (2018). Growth Engineering at Netflix — Accelerating Innovation, [Online]. Available: https://netflixtechblog.com/growth-engineering-at-netflix-accelerating-innovation-90eb8e70ce59 (visited on 02/27/2024).

[132] S. Ananthanarayanan, M. S. Ardekani, D. Haenikel, *et al.*, "Keeping Master Green at Scale", in *Proceedings of the EuroSys Conference*, Association for Computing Machinery, 2019.

[133] B. Grubic, Y. Wang, T. Petrochko, *et al.*, "Conveyor: One-Tool-Fits-All Continuous Software Deployment at Meta", in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, USENIX Association, 2023.

[134] D. Huye, Y. Shkuro, and R. R. Sambasivan, "Lifting the veil on Meta's microservice architecture: Analyses of topology and request workflows", in *USENIX Annual Technical Conference (ATC)*, USENIX Association, 2023.

[135] H. Saokar, S. Demetriou, N. Magerko, *et al.*, "ServiceRouter: Hyperscale and Minimal Cost Service Mesh at Meta", in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, USENIX Association, 2023.

[136] B. Sigelman. (2018). Three Pillars of Observability with Zero Answers | Lightstep Blog, [Online]. Available: https://lightstep.com/blog/three-pillars-zero-answers-towards-new-scorecard-observability (visited on 02/27/2024).

[137] K. Lew and S. Narayanan. (2018). Lessons from Building Observability Tools at Netflix, [Online]. Available: https://netflixtechblog.com/lessons-from-building-observability-tools-at-netflix-7cfafed6ab17 (visited on 02/27/2024).

[138] J. Jackson. (2018). Debugging Microservices: Lessons from Google, Facebook, Lyft, [Online]. Available: https://thenewstack.io/debugging-microservices-lessons-from-google-facebook-lyft/ (visited on 02/27/2024).

[139] G. Miranda. (2023). A Sample of Sampling, and a Whole Lot of Observability at Scale, [Online]. Available: https://www.honeycomb.io/blog/sampling-observability-slack (visited on 02/27/2024).

[140] G. Leffler. (2021). The Hidden Cost of Sampling in Observability, [Online]. Available: https://www.splunk.com/en_us/blog/devops/the-hidden-cost-of-sampling-in-observability.html (visited on 02/27/2024).

[141] Prometheus. (2024). Prometheus: Monitoring system & time series database, [Online]. Available: https://prometheus.io/ (visited on 02/27/2024).

[142] Z. Huang, P. Chen, G. Yu, H. Chen, and Z. Zheng, "Sieve: Attention-based sampling of end-to-end trace data in distributed microservice systems", in *IEEE International Conference on Web Services (ICWS 21)*, 2021.

[143] X. Zhou, X. Peng, T. Xie, *et al.*, "Latent Error Prediction and Fault Localization for Microservice Applications by Learning from System Trace Logs", in *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, Association for Computing Machinery, 2019.

[144] D. Ardelean, A. Diwan, and C. Erdman, "Performance Analysis of Cloud Applications", in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, USENIX Association, 2018.

[145] L. Wu, J. Tordsson, E. Elmroth, and O. Kao, "MicroRCA: Root Cause Localization of Performance Issues in Microservices", in *IEEE/IFIP Network Operations and Management Symposium (NOMS 20)*, IEEE, 2020.

[146] M. Kim, R. Sumbaly, and S. Shah, "Root cause detection in a service-oriented architecture", in *Proceedings of the ACM SIGMETRICS Conference*, Association for Computing Machinery, 2013.

[147] L. Zhang, Z. Xie, V. Anand, Y. Vigfusson, and J. Mace, "The Benefit of Hindsight: Tracing Edge-Cases in Distributed Systems", in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023.

[148] Intel. (2024). Intel Infrastructure Processing Unit (Intel IPU) ASIC E2000, [Online]. Available: https://www.intel.com/content/www/us/en/products/details/network-io/ipu/e2000-asic.html (visited on 02/27/2024).

[149] N. Mehta. (2022). The next wave of Google Cloud infrastructure innovation: New C3 VM and Hyperdisk, [Online]. Available: https://cloud.google.com/blog/products/compute/introducing-c3-machines-with-googles-custom-intel-ipu (visited on 02/27/2024).

[150] D. Firestone, A. Putnam, S. Mundkur, *et al.*, "Azure Accelerated Networking:{SmartNICs} in the Public Cloud", in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018.

[151] NVIDIA. (2023). NVIDIA BlueField Networking Platform, [Online]. Available: https://www.nvidia.com/en-us/networking/products/data-processing-unit/ (visited on 02/27/2024).

[152] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg", in *Proceedings of the EuroSys Conference*, 2015.

[153] L. A. Barroso, J. Clidaras, and U. Hölzle, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition*. Morgan & Claypool Publishers, 2013.

[154] P. Ambati, I. Goiri, F. Frujeri, *et al.*, "Providing SLOs for Resource-Harvesting VMs in Cloud Platforms", in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020.

[155] Y. Zhang, Í. Goiri, G. I. Chaudhry, *et al.*, "Faster and Cheaper Serverless Computing on Harvested Resources", in *Proceedings of the 28th ACM SIGOPS Symposium on Operating Systems Principles (SOSP 21)*, 2021.

[156] A. Yigal. (2023). Three Observability Trends That Will Resonate in 2024 – And What To Do About Them, [Online]. Available: https://vmblog.com/archive/2023/11/22/logz-io-2024-predictions-three-observability-trends-that-will-resonate-in-2024-and-what-to-do-about-them.aspx (visited on 02/27/2024).

[157] T. Levy. (2023). There's No Value in Observability Bloat. Let's Focus on the Essentials, [Online]. Available: https://devops.com/theres-no-value-in-observability-bloat-lets-focus-on-the-essentials/ (visited on 02/27/2024).

[158] OpenTelemetry. (2023). OpenTelemetry, [Online]. Available: https://opentelemetry.io/ (visited on 02/27/2024).

[159] Community. (2024). Online Boutique, [Online]. Available: https://github.com/GoogleCloudPlatform/microservices-demo (visited on 02/27/2024).

[160] U. Ayachit. (2023). Exploring an Automated Testing Strategy for Infrastructure as Code, [Online]. Available: https://techcommunity.microsoft.com/t5/azure-high-performance-computing/exploring-an-automated-testing-strategy-for-infrastructure-as/ba-p/3971715 (visited on 02/27/2024).

[161] A. W. Services. (2023). EC2 instance types, [Online]. Available: https://aws.amazon.com/ec2/instance-types/ (visited on 02/27/2024).

[162] A. Lakhina, M. Crovella, and C. Diot, "Characterization of Network-Wide Anomalies in Traffic Flows", in *Proceedings of the ACM Internet Measurement Conference (IMC 04)*, Association for Computing Machinery, 2004.

[163] B. Schölkopf, J. Platt, and T. Hofmann, "In-Network PCA and Anomaly Detection", in *NIPS 2007*. IEEE, 2007.

[164] L. Huang, X. Nguyen, M. Garofalakis, *et al.*, "Communication-Efficient Online Detection of Network-Wide Anomalies", in *IEEE International Conference on Computer Communications (INFOCOM)*, IEEE, 2007.

[165] H. Huang and S. P. Kasiviswanathan, "Streaming Anomaly Detection Using Randomized Matrix Sketching", *VLDB Endowment*, 2015.

[166] A. Marchioni, L. Prono, M. Mangia, F. Pareschi, R. Rovatti, and G. Setti, "Streaming Algorithms for Subspace Analysis: Comparative Review and Implementation on IoT Devices", *IEEE Internet of Things Journal*, 2023.

[167] E. Liberty, "Simple and deterministic matrix sketching", in *Proceedings of the ACM SIGKDD Conference*, Association for Computing Machinery, 2013.

[168] M. Ghashami, E. Liberty, and J. M. Phillips, "Efficient Frequent Directions Algorithm for Sparse Matrices", in *Proceedings of the ACM SIGKDD Conference*, Association for Computing Machinery, 2016.

[169] N. Sundar, B. Burres, Y. Li, D. Minturn, B. Johnson, and N. Jain, "9.4 An In-depth Look at the Intel IPU E2000", in *IEEE International Solid-State Circuits Conference (ISSCC 23)*, 2023.

[170] X. Wei, R. Cheng, Y. Yang, R. Chen, and H. Chen, "Characterizing Off-path SmartNIC for Accelerating Distributed Systems", in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, 2023.

[171] Y. Chen, Y. Lu, and J. Shu, "Scalable RDMA RPC on Reliable Connection with Efficient Resource Sharing", in *Proceedings of the EuroSys Conference*, Association for Computing Machinery, 2019.

[172] S. Mushero. (2019). Push vs. Pull Monitoring Configs, [Online]. Available: https://steve-mushero.medium.com/push-vs-pull-configs-for-monitoring-c541eaf9e927 (visited on 02/27/2024).

[173] G. Yu, P. Chen, H. Chen, *et al.*, "Microrank: End-to-end latency issue localization with extended spectrum analysis in microservice environments", in *ACM Web Conference 2021*, 2021.

[174] J. Levin and T. A. Benson, "ViperProbe: Rethinking Microservice Observability with eBPF", in *IEEE CloudNet*, 2020.

[175] J. Park, B. Choi, C. Lee, and D. Han, "GRAF: A graph neural network based proactive resource allocation framework for SLO-oriented microservices", in *Proceedings of the 17th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT 21)*, Association for Computing Machinery, 2021.

[176] J. Heyman, C. Byström, J. Hamrén, and H. Heyman. (2024). Locust: a modern load testing framework, [Online]. Available: https://locust.io/ (visited on 02/27/2024).

[177] Community. (2024). ChaosMesh, [Online]. Available: https://chaos-mesh.org/ (visited on 02/27/2024).

[178] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer, "FIRM: An Intelligent Fine-grained Resource Management Framework for SLO-Oriented Microservices", in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, USENIX Association, 2020.

[179] V. Harsh, W. Zhou, S. Ashok, R. N. Mysore, B. Godfrey, and S. Banerjee, "Murphy: Performance Diagnosis of Distributed Cloud Applications", in *Proceedings of the ACM SIGCOMM Conference*, Association for Computing Machinery, 2023.

[180] O. T. Authors. (2023). OpenTelemtry — Sampling, [Online]. Available: https://opentelemetry.io/docs/concepts/sampling/ (visited on 02/27/2024).

[181] M. Smiley. (2023). GitLab Redis-cache failure, [Online]. Available: https://gitlab.com/gitlab-com/gl-infra/scalability/-/issues/1601#top (visited on 02/27/2024).

[182] R. Fonseca, G. Porter, R. H. Katz, and S. Shenker, "X-Trace: A Pervasive Network Tracing Framework", in *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07)*, USENIX Association, 2007.

[183] R. R. Sambasivan, A. X. Zheng, M. De Rosa, *et al.*, "Diagnosing Performance Changes by Comparing Request Flows", in *8th USENIX Conference on Networked Systems Design and Implementation*, USENIX Association, 2011.

[184] M. Toslali, E. Ates, A. Ellis, *et al.*, "Automating Instrumentation Choices for Performance Problems in Distributed Applications with VAIF", in *Proceedings of the ACM Symposium on Cloud Computing (SoCC 21)*, Association for Computing Machinery, 2021.

[185] A. Oliner, A. Ganapathi, and W. Xu, "Advances and Challenges in Log Analysis", *Communications of the ACM*, vol. 55, no. 2, 2012.

[186] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy, "SherLog: Error Diagnosis by Connecting Clues from Run-Time Logs", in *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, Association for Computing Machinery, 2010.

[187] F. Neves, N. Machado, and J. Pereira, "Falcon: A Practical Log-Based Analysis Tool for Distributed Systems", in *Proceedings of the 48th Annual IEEE/IFIP Conference on Dependable Systems and Networks (DSN)*, IEEE, 2018.

[188] Ú. Erlingsson, M. Peinado, S. Peter, and M. Budiu, "Fay: Extensible Distributed Tracing from Kernels to Clusters", in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP 11)*, Association for Computing Machinery, 2011.

[189] Community. (2024). OpenTelemetry Protocol with Apache Arrow, [Online]. Available: https://github.com/open-telemetry/otel-arrow (visited on 02/27/2024).

[190] S. Narayana, A. Sivaraman, V. Nathan, *et al.*, "Language-Directed Hardware Design for Network Performance Monitoring", in *Proceedings of the ACM SIGCOMM Conference*, Association for Computing Machinery, 2017.

[191] H. Namkung, Z. Liu, D. Kim, V. Sekar, and P. Steenkiste, "Sketchovsky: Enabling Ensembles of Sketches on Programmable Switches", in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, USENIX Association, 2023.

[192] Z. Liu, R. Ben-Basat, G. Einziger, *et al.*, "Nitrosketch: Robust and General Sketch-Based Monitoring in Software Switches", in *Proceedings of the ACM SIGCOMM Conference*, Association for Computing Machinery, 2019.

[193] Z. Liu, H. Namkung, G. Nikolaidis, *et al.*, "Jaqen: A High-Performance Switch-Native Approach for Detecting and Mitigating Volumetric DDoS Attacks with Programmable Switches", in *30th USENIX Security Symposium (USENIX Security 21)*, USENIX Association, 2021.

[194] M. S. Brunella, G. Belocchi, M. Bonola, *et al.*, "hXDP: Efficient Software Packet Processing on FPGA NICs", in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, USENIX Association, 2020.

[195] J. Xing, Y. Qiu, K.-F. Hsu, *et al.*, "Unleashing SmartNIC Packet Processing Performance in P4", in *Proceedings of the ACM SIGCOMM Conference*, Association for Computing Machinery, 2023.

[196] Y. Le, H. Chang, S. Mukherjee, *et al.*, "UNO: Uniflying Host and Smart NIC Offload for Flexible Packet Processing", in *Proceedings of the 2017 Symposium on Cloud Computing (SoCC 17)*, Association for Computing Machinery, 2017.

[197] M. Liu, T. Cui, H. Schuh, A. Krishnamurthy, S. Peter, and K. Gupta, "Offloading Distributed Applications onto SmartNICs Using iPipe", in *Proceedings of the ACM SIGCOMM Conference*, Association for Computing Machinery, 2019.

[198] J. Kim, I. Jang, W. Reda, *et al.*, "LineFS: Efficient smartnic offload of a distributed file system with pipeline parallelism", in *Proceedings of the 28th ACM SIGOPS Symposium on Operating Systems Principles (SOSP 21)*, Association for Computing Machinery, 2021.

[199] H. N. Schuh, W. Liang, M. Liu, J. Nelson, and A. Krishnamurthy, "Xenic: SmartNIC-Accelerated Distributed Transactions", in *Proceedings of the 28th ACM SIGOPS Symposium on Operating Systems Principles (SOSP 21)*, Association for Computing Machinery, 2021.

[200] J. Li, Y. Lu, Q. Wang, J. Lin, Z. Yang, and J. Shu, "AlNiCo: SmartNIC-accelerated Contention-aware Request Scheduling for Transaction Processing", in *USENIX Annual Technical Conference (ATC)*, USENIX Association, 2022.

[201] Z. Wang, H. Huang, J. Zhang, F. Wu, and G. Alonso, "FpgaNIC: An FPGA-based Versatile 100Gb SmartNIC for GPUs", in *USENIX Annual Technical Conference (ATC)*, USENIX Association, 2022.

[202] M. Liu, S. Peter, A. Krishnamurthy, and P. M. Phothilimthana, "E3: Energy-Efficient Microservices on SmartNIC-Accelerated Servers", in *USENIX Annual Technical Conference (ATC)*, USENIX Association, 2019.

This Ph.D. thesis has been typeset by means of the TEX-system facilities. The typesetting engine was pdfLATEX. The document class was `toptesi`, by Claudio Beccari, with option `tipotesi=scudo`. This class is available in every up-to-date and complete TEX-system installation.