

UNIVERSITY OF CALGARY

Proactive Traffic Sampling with Dynamic Flow Rates in Software-Defined Networks

by

Soroosh Esmailian

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

GRADUATE PROGRAM IN COMPUTER SCIENCE

CALGARY, ALBERTA

DECEMBER, 2024

© Soroosh Esmailian 2024

Abstract

As modern networks continue to grow in scale and speed, traffic sampling has become an indispensable tool in network management. While there exists a plethora of sampling systems, they generally assume flow rates are stable and predictable over a sampling period. Consequently, when deployed in networks with dynamic flow rates, some flows may be missed or under-sampled, while others are over-sampled.

This thesis presents the design and evaluation of **dSamp**, a network-wide sampling system capable of handling dynamic flow rates in Software-Defined Networks (SDNs). The key idea in **dSamp** is to consider flow rate fluctuations when deciding on which network switches and at what rate to sample each flow. We formulate the network-wide sampling with dynamic flow rates as a robust optimization problem. Our proactive approach leverages statistical information about flow rates to cope with fluctuations in flow rates. Since our initial formulation is an Integer Second Order Cone Program (ISOCP), which is infeasible for both small-scale and large-scale network instances, we shift our focus to developing an efficient approximate Integer Linear Program (ILP) called **APX**, which can compute sampling allocations even for large-scale networks. To show the efficacy of **dSamp** for network monitoring, we have implemented **APX** and several existing solutions in ns-3 and conducted extensive experiments using both model-driven and trace-driven simulations.

Our model-driven results indicate that **APX** outperforms the approaches in [50] and [21] by up to 10%. Similarly, our trace-driven results show that **APX** surpasses these works by up to 6.37%. Unlike [50] and [21], which require fine-tuning in model-driven simulations for use in trace-driven simulations, **APX** works across all simulations without such a requirement.

Acknowledgements

I would like to extend my heartfelt thanks to Professor Majid Ghaderi for his invaluable guidance and expertise throughout my research journey. His pivotal role was crucial in enabling me to conduct research at this level and successfully complete my thesis. Furthermore, I am deeply appreciative of the support provided by Mahdi Dolati and Sogand Sadraghighi. Their insights and encouragement were key to the fruition of this research.

I cannot overlook the enduring support and love from my family. Their faith in me and the sacrifices they made to support my aspirations have left me eternally grateful.

Lastly, I wish to express my gratitude to all those who played a part, directly or indirectly, in this endeavor. Your contributions, whether large or small, have been an essential component of this fulfilling journey.

Table of Contents

Abstract	ii
Acknowledgements	iii
Table of Contents	v
List of Figures	vi
List of Tables	vii
List of Symbols, Abbreviations, and Nomenclature	viii
1 Introduction	1
1.1 Motivation	1
1.2 Objective	4
1.3 Contribution	6
1.4 Organization	7
2 Background and Related Works	8
2.1 Mathematical Preliminaries	8
2.1.1 Optimization	8
2.1.2 Chance-Constrained Optimization	11
2.2 Background	12
2.2.1 Software-Defined Networking (SDN)	12
2.2.2 Network Monitoring	15
2.2.3 Network TAP Devices	17
2.2.4 Traffic Mirroring	17
2.2.5 Traffic Sampling	18
2.3 Related Works	19
2.3.1 Per-Port Sampling Solutions	21
2.3.2 Deterministic Per-Flow Sampling Solutions	21
2.3.3 Adaptive Per-Flow Sampling Solutions	22
2.4 Summary	24

3	Sampling System	27
3.1	System Design	27
3.1.1	System Workflow	28
3.1.2	Sampling Service	29
3.1.3	Orchestrator	29
3.1.4	Traffic Estimator	30
3.1.5	Optimizer	30
3.1.6	Discussion	30
3.2	Sampling Optimization	31
3.2.1	Problem Formulation	31
3.2.2	Linearization	35
3.2.3	Approximation	39
4	Evaluations	41
4.1	Implementation	41
4.1.1	Why ns-3?	41
4.1.2	Workflow	42
4.1.3	Network Components	44
4.1.4	Functions	45
4.2	Benchmarks	53
4.2.1	APX Scalability Analysis	57
4.2.2	APX Comparison with ILP and ISOCP	59
4.2.3	APX Load Distribution Analysis	62
4.2.4	APX Sensitivity to Rate Distribution	63
4.2.5	Effect of Epoch Length	65
4.2.6	Effect of Rate Variability	66
4.2.7	Effect of Capacity Violation Probability (δ)	67
4.2.8	Model-Driven Simulations	67
4.2.9	Trace-Driven Simulations	70
5	Conclusion and Future Works	77
5.1	Conclusion	77
5.2	Thesis Summary	77
5.3	Future Works	78
	Bibliography	81

List of Figures

1.1	A small network with two switches and four flows.	4
2.1	Main components of an OpenFlow switch.	13
3.1	High-level architecture of dSamp	28
4.1	Model-Driven ns-3 Workflow.	43
4.2	Abilene topology used in our trace-driven and model-driven simulations. Link speeds are 10 Gbps.	54
4.3	Average shortest path lengths for scale-free networks generated using the approach in [26] with varying numbers of switches.	57
4.4	Measured runtime of APX	57
4.5	Performance comparison between APX and ILP	58
4.6	Performance comparison between APX , ILP and ISOCP under settings of low variation, mixed variation, and high variation in flow rates.	60
4.7	Effect of rate distribution on APX	62
4.8	Histogram of different distributions.	62
4.9	Simulations for testing different epoch lengths on APX	65
4.10	Effect of variability in flow rates.	66
4.11	Effect of capacity violation probability.	67
4.12	Properties of model-driven traffic.	68
4.13	Model-driven ns-3 simulations results.	70
4.14	Properties of trace-driven traffic.	71
4.15	Simulating traffic between host 1 and host 3 over five epochs using a 25-second sample from the ISP trace [5].	72
4.16	Trace-driven ns-3 simulation results.	73
4.17	Trace-driven ns-3 simulation results under different delay scenarios.	75

List of Tables

2.1	Summary of the traffic sampling solutions.	25
3.1	Table of Notations.	32
4.1	Default simulation parameters.	55
4.2	Runtime comparison between APX and ILP.	58
4.3	Low variation setting (CoV = 0.2).	59
4.4	Mixed variation setting (CoV = [0.2, 0.5, 2]).	59
4.5	High variation setting (CoV = 2).	59

List of Symbols, Abbreviations, and Nomenclature

Symbol	Definition
CLT	Central Limit Theorem
CoV	Coefficient of Variation
DC	Data Center
ECMP	Equal Cost Multi-Path Routing
ILP	Integer Linear Programming or Program
ISOCP	Integer Second Order Cone Programming or Program
ISP	Internet Service Provider
LP	Linear Programming or Program
MAC	Media Access Control
MILP	Mixed-Integer Linear Programming or Program
MIP	Mixed-Integer Program or Programming
MIPGap	Mixed-Integer Programming Gap
NAT	Network Address Translation

OD	Origin-Destination
pps	Packets Per Second
SDN	Software-Defined Networks or Networking
SNMP	Simple Network Management Protocol
SOCP	Second-Order Cone Programming or Program
TCAM	Ternary Content-Addressable Memory
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol
VM	Virtual Machine

Chapter 1

Introduction

1.1 Motivation

Network monitoring offers a view of the network and illustrates network behavior, which serves as a foundation for many network management tasks such as anomaly detection [14], customer accounting [24] and traffic engineering [63]. Traditional network monitoring has focused on coarse-grained measurements, where the content of packets is disregarded. For instance, the Simple Network Management Protocol (SNMP) [40] can be used to collect information such as the number of packets and bytes transmitted on each device. These types of coarse-grained, low-frequency statistics are suitable only for certain network management tasks, like traffic engineering, and are inadequate for tasks requiring fine-grained information [73]. Consequently, detailed network measurements have become essential, leading network monitoring solutions to analyze network behavior at the packet-level granularity. To address this need, many network operators employ monitoring solutions based on packet sampling, where network switches sample only a subset of the packets passing through them [21].

Sampling can be performed on a per-port or per-flow basis. In per-port sampling solutions, such as NetFlow [20] and sFlow [8], a sampling rate is specified for each switch input port. Consequently, flows with higher packet rates are more likely to be sampled compared to those with lower rates. Therefore, these solutions are not suitable for network management tasks that require a consistent per-flow sampling rate, such as anomaly detection [30]. Per-flow sampling solutions such as [54], address this bias towards high-rate flows by specifying a target sampling rate for each flow. However, in these solutions, switches sample flows independently, leading to redundant flow samples and inefficient use of switch resources.

Centralized sampling solutions can enhance the flow monitoring capabilities of networks and prevent redundant sampling [50]. Thus, recent sampling solutions [21, 47] have adopted centralized sampling, where a centralized controller coordinates the monitoring responsibilities across different switches within the network. To determine the per-flow sampling rates on each switch, these works typically assume that the flow rates are deterministic, in the sense that they are 1) fixed over time, and 2) known a priori. However, in reality, flow rates are dynamic and fluctuate over time. Consequently, these solutions may result in under-sampling or over-sampling when the actual traffic rate of a flow deviates from the assumed rate. This limitation renders such solutions less effective for many network management tasks that require a minimum per-flow sampling rate, such as anomaly detection, since it reduces flow visibility. We define flow visibility as the percentage of flows that meet a specific target sampling rate. Higher visibility can increase the performance of various network management tasks.

High flow visibility is one of the objectives that today's monitoring solutions must achieve. Beyond flow visibility, a monitoring system is also expected to provide **scalabil-**

ity and **low overhead**. To achieve high flow visibility, a monitoring solution such as [22] could dynamically *react* to flow rate fluctuations. However, these *reactive* approaches significantly increase computation and communication overhead, as the sampling allocations must be frequently recalculated and installed on the switches, thereby violating the low overhead requirement. In this context, sampling allocations specify the set of switches and the corresponding sampling rates for each flow. A *proactive* approach, on the other hand, can provision the right amount of sampling resources in advance, avoiding the need to recompute the sampling schedule. Recent proactive approaches [37, 23, 21], however, either prepare for worst-case fluctuations or do not consider instantaneous fluctuations within a measurement interval, resulting in suboptimal use of sampling resources. Another challenge for a monitoring solution is ensuring scalability, allowing it to function effectively even when deployed in networks with high traffic volumes, such as those found in large-scale Internet Service Providers (ISPs) and data centers (DCs).

Addressing the above objectives is challenging, and many network monitoring solutions struggle to fulfill all three requirements. This motivates the adoption of **dSamp**, a packet-level monitoring system designed to effectively handle dynamic flow rates in Software-Defined Networks (SDNs). In this work, we focus on developing a *proactive* sampling system that utilizes statistical flow rate information to compute robust sampling schedules. These schedules can cope with flow rate fluctuations in networks, thereby avoiding frequent recalculations and also considering instantaneous flow rate fluctuations.

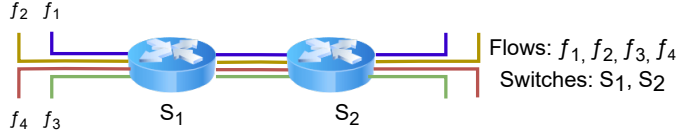


Figure 1.1: A small network with two switches and four flows.

1.2 Objective

In this thesis, our primary objective is to design a packet-level monitoring system that can address the dynamic nature of flow rates in SDNs while being scalable and providing low overhead.

Recent sampling solutions [21, 47] have adopted coordinated sampling, where a centralized controller coordinates monitoring responsibilities across different switches. These solutions offer higher flow visibility compared to those lacking network-wide coordination by avoiding redundant sampling. Nonetheless, as previously mentioned, these solutions fail to account for fluctuations in flow rates, resulting in decreased flow visibility.

To show the effect of considering flow rate fluctuations in an example, consider the small network depicted in Fig. 1.1. The network consists of two switches denoted by S_1 and S_2 , each having a sampling capacity of 3 packets per second (pps). There are four traffic flows denoted by f_1, f_2, f_3 and f_4 that are required to be sampled at the sampling rate of 0.1. The rates of f_1 and f_2 follow a Normal distribution with mean 5 pps and standard deviation 10 pps, while the rates of f_3 and f_4 follow a Normal distribution with mean 14 pps and standard deviation 1 pps. A deterministic sampling solution that ignores rate fluctuations assigns flows f_1 and f_3 to switch S_1 , and flows f_2 and f_4 to switch S_2 for sampling. This assignment ensures equal sampling load distribution (with respect to the mean flow rates) between the switches, which is a common objective in coordinated sampling solutions [50, 47]. Let δ_i denote the probability that capacity of switch i is violated (*i.e.*, its sampling capacity

is not sufficient to sample its flows at the specified target sampling rate). We have:

$$\delta_1 = \delta_2 = \mathbb{P} \{0.1 \times \mathcal{N}(14 + 5, 1 + 100) > 3\} = 0.14.$$

Next, consider a sampling solution that considers variability in flow rates, such as the one proposed in this thesis. It may decide on the following assignment, where flows f_1 and f_2 are assigned to switch S_1 and f_3 and f_4 are assigned to switch S_2 . In this case, we obtain that:

$$\delta_1 = \mathbb{P} \{0.1 \times \mathcal{N}(5 + 5, 100 + 100) > 3\} = 0.08,$$

$$\delta_2 = \mathbb{P} \{0.1 \times \mathcal{N}(14 + 14, 1 + 1) > 3\} = 0.08,$$

which represents almost 43% reduction in the switch capacity violation probability by simply considering fluctuations in flow rates. Any reduction in the switch capacity violation probability translates to more precise flow sampling and, consequently, enhanced visibility of network flows.

Our objective is to design a sampling system that accommodates the dynamic nature of flow rates while being scalable for large networks. Our key idea involves taking flow rate fluctuations into account during the decision-making process. This approach contrasts with existing works that assume knowledge of the exact flow rates. While it is not possible to know the exact rate of flows in advance, statistical information, such as mean and variance, can often be derived from historical measurements or service level agreements. Given this, we will explore how flow rate fluctuations can be modeled based on such information and how these models can be integrated into the decision-making process.

1.3 Contribution

This thesis presents several key contributions in the design and evaluation of **dSamp**, a proactive traffic sampling system for SDNs that addresses the dynamic nature of flow rates:

Customized Problem Formulation. We model the network-wide traffic sampling problem with dynamic flow rates in SDNs as an Integer Second Order Cone Program (ISOCP). This formulation captures critical considerations such as flow rate fluctuations, switch capacity constraints, and the need for network-wide coordinated sampling. Specifically, we incorporate statistical flow rate information, such as the mean and variance of flow rates, which can often be derived from historical measurements or service level agreements. This approach is more robust compared to relying on instantaneous flow rates.

Efficient Approximation Algorithm. To overcome the computational infeasibility of solving the ISOCP formulation for both small-scale and large-scale networks, we introduce two reformulations. The first reformulation, called **ILP**, is a linear transformation of the original problem; however, it is only suitable for small-scale networks. Therefore, we move toward an approximation method called **APX**, which has been proven to be feasible for large-scale networks.

ns-3 Simulations and Benchmarks. We present extensive ns-3, model-driven, and trace-driven simulations to study **dSamp**'s micro and macro behavior. Microbenchmarks compare ISOCP, ILP, and APX under various conditions, while macrobenchmarks include sets of model-driven and trace-driven simulations. We perform model-driven simulations to compare APX with existing sampling solutions from [50] and [21], and we fine-tune the approach from [50] for use in trace-driven simulations. Subsequently, we conduct extensive trace-driven simulations to compare **dSamp** with other sampling systems, utilizing real traffic traces collected from an ISP. Our model-driven results indicate that **dSamp** surpasses the approaches in [50] and [21] by up to 10%. Additionally, our trace-driven results show that **dSamp** outperforms these methods by up to 6.37%. Through trace-driven simulations, we also analyze

the impact of introducing a delay at the start of each epoch.

1.4 Organization

The rest of this thesis is organized as follows:

- Chapter 2 presents the necessary mathematical preliminaries, relevant background on network traffic monitoring, and a literature review of previous studies in network traffic sampling.
- Chapter 3 presents the design of **dSamp**, a network-wide sampling system capable of handling flow rate fluctuations in SDNs, suitable for use in OpenFlow networks. We begin with an ISOCP formulation, which proves to be infeasible for both small-scale and large-scale scenarios. Given this, a linear transformation of the ISOCP formulation called ILP is proposed; however, it remains infeasible for large-scale scenarios. To overcome this limitation, an approximation method called APX is introduced.
- Chapter 4 benchmarks ISOCP, ILP and APX algorithms. We begin with microbenchmarks, which compare ISOCP, ILP, and APX under various conditions. Next, we move to macrobenchmarks, utilizing both model-driven and trace-driven scenarios to validate APX's superior flow visibility compared to other sampling solutions. Additionally, we analyze the impact of introducing a delay at the start of each epoch.
- Finally, Chapter 5 concludes the thesis and presents some possible directions for future research.

Chapter 2

Background and Related Works

In this chapter, we cover the main mathematical preliminaries, used throughout the thesis in Section 2.1. We then review some background information related to network monitoring in Section 2.2. Section 2.3 provides a discussion on different network traffic sampling solutions. Finally, Section 2.4 summarizes these solutions.

2.1 Mathematical Preliminaries

This section covers the main mathematical preliminaries used throughout this thesis.

2.1.1 Optimization

Optimization is the process by which the optimal solution to a problem, or optimum, is produced. The word optimum has come from Latin word *optimus* meaning best [27]. Optimization in science involves systematically finding the best solution to a problem by selecting the most effective option from a set of available alternatives. This process typically requires maximizing or minimizing some objective (e.g., efficiency, cost, speed) under specific constraints. It's a critical tool in various scientific fields for enhancing outcomes, improving processes, and solving complex problems efficiently. Through optimization, scientists and

researchers can determine the optimal conditions or parameters for experiments, designs, and systems, leading to advancements and innovations across disciplines.

Consider a general optimization problem with a minimization objective, which can be formulated as follows [55]:

$$\text{minimize } f(x) \tag{2.1}$$

$$\text{subject to } g_i(x) \leq b_i, \quad \forall i \in \mathcal{I} \tag{2.2}$$

$$g_i(x) = c_i, \quad \forall i \in \mathcal{E} \tag{2.3}$$

$$x \in \mathcal{X} \tag{2.4}$$

This describes a general optimization problem aimed at minimizing the objective function $f(x)$, while satisfying inequality constraints $g_i(x) \leq b_i$ and equality constraints $g_i(x) = c_i$, with the decision variable x being within the feasible set \mathcal{X} . Depending on the nature of the objective function $f(x)$, the constraints $g_i(x)$, and the nature of the set \mathcal{X} , various types of optimization problems may arise:

1. **Linear Programming (LP):** In Linear Programming, the objective function is linear, and the constraint functions are affine, meaning they are composed of a linear function and a constant:

$$\min_x f(x) = c^T x = \sum_{j=1}^n c_j x_j \quad c \in \mathbb{R}^n \tag{2.5}$$

$$\text{subject to } g_i(x) = a_i^T x - b_i \leq 0 \quad i \in \mathcal{I} \cup \mathcal{E}, a_i \in \mathbb{R}^n, b_i \in \mathbb{R} \tag{2.6}$$

$$x \in \mathcal{X} \quad \mathcal{X} = \{x \in \mathbb{R}^n \mid x_j \geq 0, j = 1, 2, \dots, n\} \tag{2.7}$$

2. **Nonlinear Programming (NLP):** Some function(s) $f(x), g_i(x)$ ($i \in \mathcal{I} \cup \mathcal{E}$) are nonlinear.

3. **Continuous Optimization:** $f(x), g_i(x)$ ($i \in \mathcal{I} \cup \mathcal{E}$) are continuous on an open set containing \mathcal{X} ; \mathcal{X} is closed and convex.
4. **Integer Programming (IP):** In integer programming, some or all of the decision variables are required to take integer values. The objective function and constraints in an IP problem can be either linear or nonlinear, *i.e.*, $\mathcal{X} \subset \{0, 1\}^n$ (binary) or $\mathcal{X} \subset \mathbb{Z}^n$ (integer). Integer Linear Programming (ILP) is a specific case of IP where the objective function and constraints are linear, and all decision variables are restricted to integer values. When an ILP is extended to include both integer and continuous decision variables, it becomes a Mixed-Integer Linear Programming (MILP) problem. Both ILP and MILP are NP-hard, reflecting their inherent computational difficulty.
5. **Unconstrained Optimization:** $\mathcal{I} \cup \mathcal{E} = \emptyset$; $\mathcal{X} = \mathbb{R}^n$.
6. **Constrained Optimization:** $\mathcal{I} \cup \mathcal{E} \neq \emptyset$ and/or $\mathcal{X} \subset \mathbb{R}^n$.
7. **Quadratic Programming (QP):** Quadratic Programming (QP) is an optimization problem where the objective function is quadratic, and the constraints are linear. The quadratic objective function $f(x)$ can be expressed in the form of $\frac{1}{2}x^T Qx + c^T x$.
8. **Second-order Cone Programming (SOCP):** The constraints in SOCP can be more complex than LP or QP, involving a second-order cone (also known as a Lorentz cone or a quadratic cone [38]). A second-order cone constraint takes the form of $\|A_i x + b_i\|_2 \leq c_i^T x + d_i$ and can replace the linear constraints $g_i(x) \leq b_i$. If the decision variables are restricted to integer values, the problem becomes an Integer Second-Order Cone Program (ISOCP), which is more complex than an SOCP.

In the above optimization problems we did not consider optimization under uncertainty, that is, where some of f, g_i ($i \in \mathcal{I} \cup \mathcal{E}$) are only known probabilistically. We will now discuss chance-constrained optimization, which aims to address optimization problems under uncertainty.

2.1.2 Chance-Constrained Optimization

Chance-constrained optimization is a type of mathematical programming that deals with optimization problems under uncertainty. The core idea is to ensure that constraints are satisfied not always, but with a certain probability. This is particularly useful in situations where exact predictions are challenging, and some level of risk is acceptable. However, many chance-constrained problems involve non-convex feasible regions, making them difficult to solve using standard optimization techniques [42]. Non-convex problems are challenging because they may have multiple local minima, making it computationally intensive to find the global minimum. A typical chance-constrained optimization problem can be formulated as follows:

$$\min_{x,t} = \{t : \text{Prob}_{(c,A,b) \sim \mathcal{P}}(c^T x \leq t, Ax \leq b) \geq 1 - \epsilon\}, \quad (2.8)$$

In the above equation, x is the decision variable, the uncertain parameters are the coefficients c , the matrix A , and the vector b . These parameters are typically modeled as random variables with a joint probability distribution ρ . The goal is to minimize t such that the probability of satisfying the constraints $c^T x \leq t$ and $Ax \leq b$ is at least $1 - \epsilon$. ϵ is a small positive number representing the acceptable probability of constraint violation. The uncertainty in the problem comes from the fact that c , A , and b are not fixed but rather follow a distribution. One effective way to address the computational challenges of chance-constrained problems is by converting them into *Second-Order Cone Programs (SOCPs)*. This is possible when the underlying uncertainty is normally distributed and the constraints are affine. The key idea is to approximate the chance constraint using a convex deterministic equivalent that can be formulated as a SOCP, thereby making the problem more tractable in both theory and practice [17].

2.2 Background

2.2.1 Software-Defined Networking (SDN)

Software-Defined Networking (SDN) was developed in response to the challenges and limitations of traditional IP networks, which are inherently complex and difficult to manage [33]. Traditional networks often require manual configuration of individual devices, making it hard to adapt to changes, enforce policies, and manage the network efficiently. Moreover, traditional networks are vertically integrated, meaning the control plane (which decides how to handle network traffic) and the data plane (which forwards traffic) are tightly coupled within the same devices, limiting flexibility and hindering innovation.

SDN addresses these challenges by decoupling the control plane from the data plane, allowing the control logic to be centralized in a software-based controller, resulting in improved network performance in terms of network management, control and data handling.

The benefits of SDNs have led to the widespread adoption of SDN in various industries, including large-scale data centers and cloud service providers, where it has been proven to significantly improve operational efficiency and reduce costs.

Openflow. OpenFlow is a powerful protocol that enables the decoupling of control and data planes in networks, providing centralized control, dynamic traffic management, and facilitating innovation through the flexibility to test new network technologies. Its deployment in various environments, from campus networks to virtualized settings, demonstrates its adaptability and potential to shape the future of networking [31].

The OpenFlow architecture consists of a controller, an OpenFlow switch, and a secure channel. The controller centrally manages the network to implement the functions of the control layer. The OpenFlow switch handles forwarding at the data layer and communicates with the controller through a secure channel to receive forwarding rules and report its status. The secure channel established between the controller and the OpenFlow switch enables the controller to manage the switch and receive feedback from it.

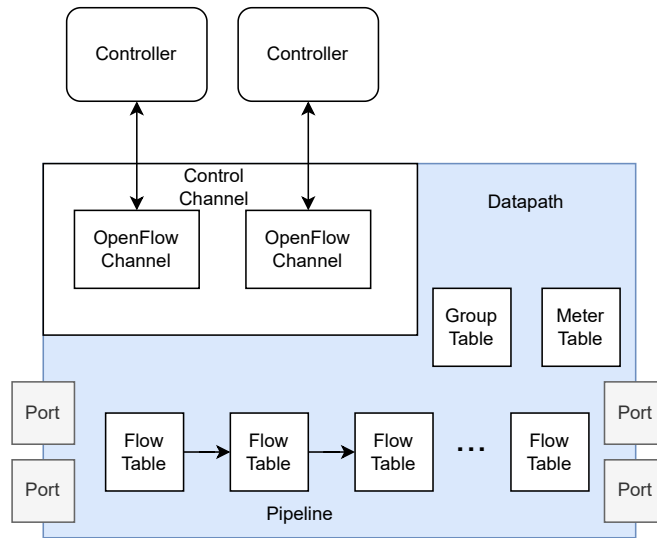


Figure 2.1: Main components of an OpenFlow switch.

An OpenFlow logical switch consists of several components, as shown in Fig. 2.1, including one or more flow tables and a group table, which handle packet lookups and forwarding, as well as one or more OpenFlow secure channels to an external controller. Below is a brief explanation of each component [7]:

- *Flow tables*: These tables are central to the operation of an OpenFlow switch. Each flow table contains a set of flow entries, where each entry includes match fields, counters, and a set of instructions to apply to matching packets. Packets entering the switch are matched against flow entries in these tables. If a match is found, the corresponding actions are executed.
- *Group table*: This table allows more complex actions on packets, such as load balancing, multipath forwarding, and failover. A group entry in this table can point to multiple actions, and depending on the group type, packets can be processed in various ways, such as being sent to multiple outputs or being forwarded based on group membership.
- *OpenFlow secure channel*: This component serves as the communication interface between the OpenFlow switch and the external controller. Through this channel, the

controller can manage the switch by adding, updating, or deleting flow entries in the flow tables, both reactively (in response to packet events) and proactively (in advance of expected traffic).

- *Controller*: The controller is an external entity that manages the OpenFlow switch through the OpenFlow protocol. It defines network behavior by programming the flow tables and other components within the switch.
- *Pipeline*: The pipeline refers to the sequence in which packets are processed through the flow tables within the switch. After a packet matches an entry in a flow table, it might be processed further by additional flow tables in the pipeline, depending on the instructions associated with the flow entry.
- *Datapath*: This is the combination of flow tables and the group table that the packet goes through. It represents the path that data takes through the switch, which is defined by the entries in the flow and group tables.

The OpenFlow secure channel is usually encrypted using Transport Layer Security (TLS), but may be run directly over Transmission Control Protocol (TCP) in plain text in OpenFlow 1.1 and later versions. The following OpenFlow messages are transmitted over the channel:

- *Controller-to-Switch message*: is sent by the controller to the OpenFlow switch to manage or obtain the OpenFlow switch status. Messages such as Statistics Request (*OFPMP_FLOW*), Aggregate Statistics Request (*OFPMP_AGGREGATE*), and Port Statistics Request (*OFPMP_PORT_STATS*) are all categorized as *controller-to-switch* messages which will allow controller to capture the statistics of flows stored on each switch [7].
- *Asynchronous message*: is sent by the OpenFlow switch to the controller to update network events or status changes to the controller. One example is the (*OFP_PACKET_IN*) message, which is triggered by the (*Packet_In*) event. This event occurs when a switch

encounters a packet that doesn't match any existing flow entries in its flow table. Instead of discarding the packet, the switch generates an (*OFPP_PACKET_IN*) message and sends it to the controller for further instructions.

- *Symmetric message*: is sent without solicitation by either the OpenFlow switch or the controller. Examples include (*OFPT_ECHO_REQUEST*) and (*OFPT_ECHO_REPLY*), which help verify that the secure channel between the controller and switch remains functional.

2.2.2 Network Monitoring

Network traffic monitoring involves the gathering and examination of data on network activity, providing network operators with insights into the dynamics of a network. Network monitoring can serve as a basis for a range of network management activities, including traffic engineering [11], traffic matrix estimation [62], and anomaly detection [35]

In general, a network monitoring system consists of a network management station, e.g., collector servers and analyzer applications, and multiple agents. Network devices such as routers, switches, and middleboxes like firewalls, load balancers, and network address translations (NATs) act as monitoring agents. These agents gather network monitoring data, such as packets and statistics, and transmit it to the network management station for comprehensive analysis and processing. In general the monitoring workflow includes five main stages [63]:

1. Data collection: Begins with the measurement phase, where data is gathered from network devices.
2. Data processing: The collected data is then aggregated and preprocessed to prepare it for analysis.
3. Data Transmission: Preprocessed data is sent to a central analysis point, usually the SDN controller or a network management station.

4. *Data Analysis*: The transmitted data is analyzed to provide insights into network performance, detect issues, and support decision-making.
5. *Data Visualization*: The results of the analysis are visualized and presented to network operators, facilitating efficient network management and troubleshooting.

The network monitoring process is inherently iterative, where each phase feeds into the next, creating a continuous loop of improvement. As data is collected, processed, and analyzed, insights gained can prompt adjustments and optimizations, which are then monitored again in the next cycle. Although network monitoring presents challenges, such as increased communication overhead, it also offers several benefits, which can be categorized as follows:

- *Improved security*: Real-time network monitoring helps detect and respond to security threats quickly, enhancing overall network protection [72].
- *Visibility into the network*: The high-level view of the network provided by network monitoring systems is valuable for traffic engineering and identifying the root causes of issues that could negatively impact network performance [68].
- *Performance optimization*: By analyzing traffic patterns and network behavior, administrators can identify bottlenecks and optimize network performance [25].
- *Problem detection and resolution*: Network monitoring systems enable rapid identification and resolution of network issues, which is essential for minimizing downtime and maintaining the stability and availability of the network. [56].
- *Capacity planning*: A critical aspect of network monitoring is optimizing infrastructure and ensuring smooth operations as an organization grows. Capacity planning, particularly through traffic prediction, is discussed in [25], where it is noted that traffic prediction helps estimate the future status of network links to consider deploying additional capacity in the network.

Since our goal is to develop a packet-level sampling system capable of handling dynamic flow rates, this thesis focuses on packet-level monitoring. The following sections provide a detailed exploration of various packet-level monitoring systems.

2.2.3 Network TAP Devices

Traditionally, monitoring network traffic at the packet level is accomplished using hardware TAP devices [64]. A hardware TAP is physically connected to a network link, typically between two network devices such as switches or routers, and it copies all the traffic that passes through it. The copied traffic can then be sent to an analyzer for further processing.

Hardware TAPs are intrusive because they require physical installation on the network, which can disrupt operations. Therefore, the optimal time to deploy a TAP is during the infrastructure's construction phase. Additionally, they lack flexibility, making them less suitable for dynamic environments like Software-Defined Networking (SDN) where resource allocation needs to be programmable. Moreover, hardware TAPs are not well-suited for virtualized network environments. They struggle to handle the dynamic nature of virtual networks, where the monitoring requirements can change rapidly. Virtual TAPs or software-based solutions are often required in such cases [46].

2.2.4 Traffic Mirroring

Traffic mirroring is a network feature that allows the duplication of network traffic from one or more ports to a designated destination port for monitoring and analysis purposes. This technique is widely used for performing tasks such as detecting intrusions, monitoring network performance [4]. Network monitoring protocols such as SNMP, Netflow and sFlow propose methods for traffic monitoring using switches or routers [64]. The port used to output the duplicated traffic is usually called the mirror port. In port mirroring, a switch duplicates its passing traffic to a designated mirroring port. Port mirroring, however, could suffer from *loss* due to port over-subscription (POS) [44]. In POS, traffic from every port on

a switch is copied to its mirroring port. If the total rate of traffic at the switch exceeds the capacity of the mirroring port, the overload traffic is dropped. Additionally, while simple and easy to deploy, POS is highly inefficient as it creates a static mirroring configuration independent of the distribution of traffic and the routing layout of the network [46].

Achieving more efficient mirroring requires adjusting the mirroring configurations based on the network's traffic distribution and the routing layout. However, most legacy switches do not support such functionality.

The Cisco Catalyst 6500 series switches [1] are examples of legacy switches that support dynamic port mirroring configurations, allowing administrators to change mirroring settings on-the-fly. Dynamic port mirroring can also be achieved using programmable switches like OpenFlow. These switches support flow mirroring in addition to port mirroring. In port mirroring, a switch replicates the traffic passing through it to a designated mirroring port. Although this method is simple and widely supported, even on legacy switches, it results in inefficient network traffic monitoring due to the port-level granularity of the mirroring process [48]. Flow mirroring, by contrast, operates at a flow-level granularity, offering more precise monitoring. However, implementing flow mirroring on OpenFlow switches is costly, as it requires the installation of additional forwarding rules, leading to increased Ternary Content-Addressable Memory (TCAM) usage on the switches [53].

2.2.5 Traffic Sampling

Capturing every packet in the network using hardware taps or traffic mirroring can provides high flow visibility. However, it imposes high communication and processing overhead in the network. Many network operators employ monitoring solutions based on packet sampling, where network switches sample only a subset of the packets that pass through them [36, 21]. However, sampling is a resource-intensive operation for network switches, given their restricted processing capabilities. A typical switch can only sample a tiny fraction of the packets it encounters before its performance starts to degrade [59].

Sampling can be performed on a per-port or per-flow basis. In per-port sampling solutions, such as NetFlow [20] and sFlow [8], a sampling rate is specified for each switch input port. As a result, flows with higher packet rates are more likely to be sampled compared to flows with lower rates. Consequently, these solutions are inadequate for network management tasks that require a minimum per-flow sampling rate, e.g., anomaly detection [14]. Per-flow sampling addresses this bias toward high-rate flows by specifying a target sampling rate for each flow. Sampling solutions based on per-flow sampling are proposed in [54] and [45].

2.3 Related Works

Network monitoring can be broadly divided into two categories: physical network monitoring and virtualized network monitoring. Physical network monitoring focuses on overseeing a single network, while virtualized network monitoring is designed to manage multiple virtual networks (VNs) coexisting on a shared physical network infrastructure. In this thesis, we will focus solely on physical network monitoring.

In physical network monitoring, many applications require measurements at the per-flow level, where a flow is defined as a sequence of packets that share a common property, such as a source IP address, destination IP address, port number, or a combination of these properties. A flow may be terminated by a timeout criterion, ensuring that the inter-packet time within the flow does not exceed a certain threshold, or by a protocol-based criterion, such as a TCP FIN packet [24]. For some network monitoring applications, such as traffic engineering [63], flow statistics like the average traffic rate, the number of flows, or the packet count over a given time period are sufficient. Legacy switches can collect these statistics through protocols such as SNMP [40]; however, OpenFlow switches use a different approach.

OpenFlow switches can measure per-flow statistics through flow entries. A flow entry is a fundamental component of the flow table within a network switch. Flow entries control

how packets are processed and forwarded by the switch. The switch examines the packet and extracts relevant fields, such as source/destination IP addresses, MAC (Media Access Control) addresses, and other header information. These fields are then compared against the match fields defined in the flow entries within the flow table. If a matching flow entry is found, the statistics for that flow will be updated. Otherwise, the switch typically uses the default action. Depending on the switch configuration, this might involve dropping the packet, sending it to the controller for further processing (a *Packet-In* event), or forwarding it based on a lower-priority, more general rule. The OpenFlow protocol [7] defines two mechanisms for collecting per-flow statistics: *pull-based* and *push-based*.

In the pull-based mechanism, the SDN controller polls the network devices (e.g., switches) periodically, for example, every few seconds to every few minutes, to gather flow statistics. This can include data such as packet counts, byte counts, flow durations, and other metrics. A controller can choose whether to collect per-flow statistics or per-switch statistics. Per-flow statistic collection provides fine-grained monitoring; however, it increases communication overhead compared to per-switch statistics. Works such as [57, 58, 60, 66, 16] have proposed methods to reduce this overhead.

In the push-based mechanism for statistics collection, network devices actively send statistical data to the SDN controller rather than the controller polling the devices for data. By using a push-based model, the need for constant polling from the SDN controller is reduced, thereby decreasing the communication overhead between the controller and the network devices. Works such as [67, 41, 67, 43] use this mechanism.

However, all these traffic statistics cannot provide the fine-grained flow measurement required by tasks such as traffic classification [69]. Therefore, it is critical for a monitoring system to provide packet-level measurement. Packet-level traces can be captured using hardware tap devices [2] or by using the port mirroring features available on legacy switches and SDN-based switches.

As the scale and speed of modern networks continue to increase, more scalable and

efficient solutions are needed for packet-level traffic monitoring. Therefore, many recent solutions adapt packet sampling [47, 21] where a subset of packets is captured by switches. In the following section, we will explore different traffic sampling solutions.

2.3.1 Per-Port Sampling Solutions

NetFlow [20] and sFlow [8] are widely used for per-port sampling. In per-port sampling, a switch samples a fraction of packets that pass through one or more of its ports regardless of which flow these packets belong to. Several works try to improve the performance of NetFlow and sFlow by more intelligently distributing sampling load on network switches. For instance, the works [65] and [46] sampling solutions that aim at minimizing the sampling load on network switches by coordinating sampling across switches. However, since per-port sampling solutions are based on a fixed sampling rate for each switch port, they are biased toward flows that send at a higher rate, which results in low accuracy for many management tasks that require a minimum per-flow sampling rate [28].

2.3.2 Deterministic Per-Flow Sampling Solutions

Works in this category consider per-flow sampling but with the assumption that flow rates remain constant during the sampling period and do not fluctuate. For instance, [13] enables real-time traffic classification in high-speed networks by sampling a fixed number of packets at each switch during the early stages of each flow. Other works, such as [29], focus on improving traffic estimates from the sampled packets. However, these methods lack coordination between switches. When switches independently sample flows, it can result in redundant packet sampling across multiple switches. Given the limited sampling capacity of switches, this redundancy results in a lower effective sampling rate per flow, reducing overall flow visibility in the network.

In response to this issue, more recent works leverage a centralized controller to coordinate sampling responsibilities across switches. The works [21], [50], [51] and [15] distribute

sampling load among different switches. These works, however, assume that the sampling controller knows the set of flows and their sending rates in advance, which is not realistic. The work [47] considers coordinated sampling in an online setting, where the controller does not have any information about the flows that will arrive in the system for sampling in the future. They also separate sampling decisions for short and long flows, by having a fixed sampling rate for short flows and only performing coordinated sampling for long flows.

All the aforementioned solutions assume fixed flow rates during the sampling period, which may lead to under-sampling, resulting in lower accuracy in some network management tasks such as anomaly detection.

2.3.3 Adaptive Per-Flow Sampling Solutions

While other network dynamics, such as changes in routing paths or network topologies, may also occur, we focus on adaptive sampling solutions that specifically address fluctuations in flow rates and are designed to effectively manage these variations. These works can be categorized in two categories: Reactive and Proactive.

Reactive Solutions

These solutions account for flow rate fluctuations by dynamically reacting to them. Works such as [22], [19], [58], and [61] dynamically adjust the polling period according to fluctuations in flow rates. When a significant change in traffic patterns is detected, [22] reduces the sampling interval to gather more data. On the other hand, if the traffic remains consistent, the interval is lengthened to avoid excessive sampling. In Payless [19], controllers adjust the polling period based on the size of flow rates. Cemon [58] uses a sliding window queue that stores recent data, specifically the differences between the values of two adjacent measurements. The controller adjusts the polling period based on the new information and data stored in the queue. Rada-rate, a technique proposed by [61], uses a modified SWT technique that can outperform both Payless and Cemon. However, all these approaches

impose high overhead because they dynamically react to flow rate fluctuations and need to install sampling allocations more frequently compared to proactive solutions.

Proactive Solutions

These works include studies that consider flow rate fluctuations and provision the right amount of resources in advance to cope with these fluctuations.

The authors in [23] aim to reduce communication overhead by assigning different sampling probabilities to flows based on their sizes. They estimate flow sizes during the current measurement interval and assign sampling probabilities for the next interval accordingly. iSTAMP [39] uses flow statistics collected during each sampling period to adaptively adjust which flows should be monitored more closely in subsequent sampling periods. OpenMeasure [37] attempts to predict future flow sizes based on previously collected data and track the most informative flows in the next interval. While these methods consider variations in flow sizes across different measurement intervals, they do not account for instantaneous changes in flow rates within the same measurement interval.

Works such as [50] and [21] assume by default that flow rates remain fixed during the sampling period; however, they also provide methods to ensure robustness against worst-case flow rate fluctuations. For example, cSamp [50] uses a post-processing mechanism to make the solution robust against estimation errors in flow rates. Specifically, the sampling solution computed for fixed flow rates is scaled to account for the maximum deviation based on the assumed estimation error. A somewhat similar approach is proposed in [21], where sampling capacities are scaled by a fixed factor to leave headroom for potential flow rate fluctuations.

All mentioned proactive solutions either do not account for instantaneous changes in flow rates or prepare for a worst-case scenario. In contrast, our approach takes a probabilistic perspective, directly bounding the probability that any switch will be overloaded due to fluctuations in flow rates.

2.4 Summary

In this chapter, we presented an overview of the mathematical preliminaries used in this thesis. Following that, we reviewed some background information related to network monitoring. Finally, we described some of the related works with a focus on traffic sampling solutions. Table 2.1 provides a high-level summary of the traffic sampling solutions covered in this chapter.

Table 2.1: Summary of the traffic sampling solutions.

Reference	Adaptive	Objective	Method	
Chowdhury <i>et al.</i> [19]	✓	Adjust the polling period based on flow sizes.	Per-flow sampling	traffic
Su <i>et al.</i> [58]	✓	Adjust the polling period based on the new information and the difference between the values of two adjacent measurements.	Per-flow sampling	traffic
Du <i>et al.</i> [23]	✓	Assigning different sampling rates for each flow based on their size to improve communication overhead.	Per-flow sampling	traffic
Malboubi <i>et al.</i> [39]	✓	Adaptively adjust which flows to monitor more closely in subsequent sampling period based on collected flow statistics.	Per-flow sampling	traffic
Cohen <i>et al.</i> [21]	✓	Develop an effective online sampling solution that optimizes the sampling process in real-time.	Per-flow sampling	traffic
Dogman <i>et al.</i> [22]	✓	Adjusts the sampling interval based on the statistical characteristics of network traffic.	Per-flow sampling	traffic
Tang <i>et al.</i> [61]	✓	Dynamically adjust the polling period optimizing the balance between measurement accuracy and overhead.	Per-flow sampling	traffic
Sekar <i>et al.</i> [50]	✓	Maximizing the overall captured traffic through coordinated sampling across switches.	Per-flow sampling	traffic
Shirali <i>et al.</i> [54]	✓	Enable packet sampling in OpenFlow switches.	Per-flow sampling	traffic
Sadrhaghghi <i>et al.</i> [47]	✓	Separating sampling decisions for long and short flows to enhance scalability.	Per-flow sampling	traffic
Liu <i>et al.</i> [37]	✓	Dynamically update measurement rules to track and measure the most informative flows.	Per-flow sampling	traffic

Reference	Adaptive	Objective	Method
Li <i>et al.</i> [36]	✓	Offer complete visibility into all network flows with low memory and bandwidth overhead.	Per-flow traffic sampling
Hohn <i>et al.</i> [29]	✓	Recover accurate information about the original traffic after sampling.	Per-flow traffic sampling
Ha <i>et al.</i> [28]	-	Maximize the detection of malicious traffic by adjusting the sampling rates at various network switches.	Per-switch traffic sampling
Xu <i>et al.</i> [65]	-	Balance the measurement load among switches.	Per-port traffic sampling
Chang <i>et al.</i> [15]	-	Distribute traffic measurement tasks evenly across network monitors.	Per-flow traffic sampling
Sekar <i>et al.</i> [51]	-	Evaluate the performance of [50] across different monitoring applications.	Per-flow traffic sampling
Canini <i>et al.</i> [13]	-	Sample few packets of each flow for high speed traffic classification.	Per-flow traffic sampling
Sadrhaghighi <i>et al.</i> [46]	-	Minimize sampling load on the switches by coordinating sampling across switches.	Per-port traffic sampling

Chapter 3

Sampling System

Our approach is based on network-wide optimization of sampling resources with respect to the routing layout and dynamic flow rates. The key idea is that, from a traffic monitoring perspective, it does not matter if only one switch in a flow path samples a flow, as long as the flow is fully sampled with a high probability. This insight leads us to design a system that maximizes the number of flows that are sampled (and can be sampled with high probability) from the set of flows selected for sampling.

To this end, we present the design and evaluation of **dSamp**, a software-defined sampling system capable of handling dynamic flow rates in SDNs. Our design can provide high *flow visibility*, *i.e.*, the percentage of flows that were fully sampled by the sampling system. While **dSamp** was simulated with ns-3 custom switches and a custom controller, it can be used in OpenFlow networks, whether the switches are physical or virtual.

3.1 System Design

The high-level architecture of **dSamp** is depicted in Fig. 3.1. The main component of **dSamp** is the orchestrator, which is implemented on top of the SDN controller. The design of **dSamp** also includes an optimizer module that works over sampling epochs. The sampling queries that are sent to the sampling service during the current sampling epoch are batched

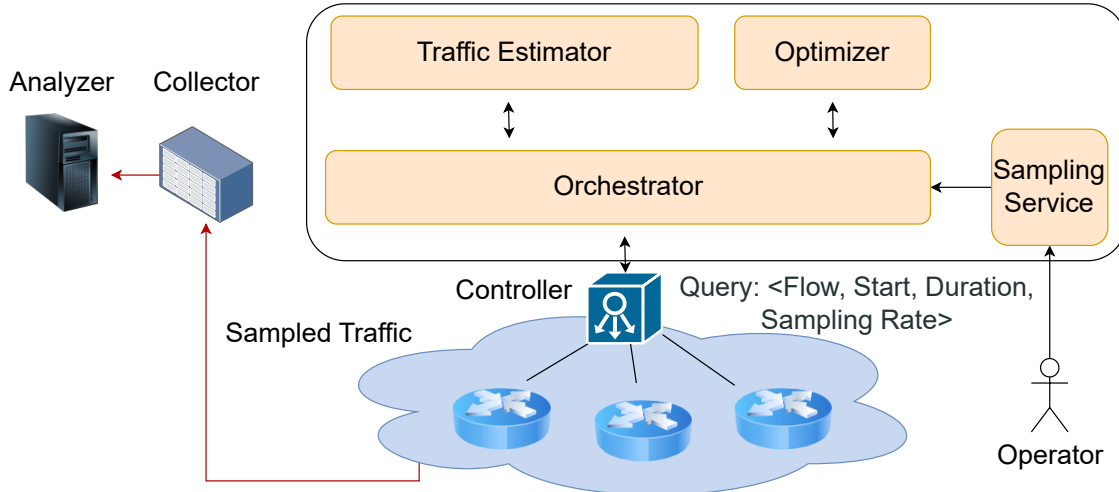


Figure 3.1: High-level architecture of dSamp.

together and then queried by the orchestrator. Afterwards, the orchestrator processes them in a single operation at the beginning of the following sampling epoch. The epoch-based structure enables **dSamp** to adapt its sampling decisions to the dynamic set of flows at the start of each epoch. In our design, a flow is specified by a source-destination IP address pair. Thus, a flow may represent the aggregated traffic between an Origin-Destination (OD) pair in an ISP network or a pair of virtual machines (VMs) in a data center network. The length of the sampling epoch is a design parameter that can be tuned to achieve different trade-offs. In particular, a shorter epoch allows the system to adapt more quickly to changes in network traffic, while a longer epoch helps reduce the control overhead. In the following, we provide a high-level overview of the system’s operational workflow, followed by a detailed examination of the individual components within the architecture.

3.1.1 System Workflow

There are two phases in **dSamp** operation, namely the setup phase and the sampling phase. In the setup phase, the network operator adjusts various system parameters such as the sampling epoch length. Once the setup phase is completed, the sampling phase starts. As stated before, **dSamp** works over sampling epochs. At the beginning of each epoch, the

orchestrator queries the controller to collect statistical data about flows and their respective paths. Additionally, it queries the sampling service to identify the flows to be sampled. It then invokes traffic estimator and optimizer modules to compute a sampling schedule for the selected flows. A sampling schedule determines which switch in the network will sample each flow. Finally, the orchestrator communicates with the controller to install the newly computed sampling schedule on the relevant switches. The destination of all sampled traffic is the collector server (essentially a load balancer in front of storage servers), which is linked to analysis applications that carry out the required management task.

3.1.2 Sampling Service

The sampling service receives sampling queries from the users (e.g., network operators and tenants). At the end of each epoch, the orchestrator collects all the sampling queries from the sampling service. Queries are defined by a tuple $\langle \text{flow}, \text{start}, \text{duration}, \text{sampling rate} \rangle$. The field “start” denotes the start time of sampling, while “duration” indicates the duration of sampling for the specified flow. The field “sampling rate” determines the desired sampling rate.

3.1.3 Orchestrator

The orchestrator is the main component of **dSamp** and includes two modules, namely traffic estimator and optimizer. All communications between **dSamp** and network switches go through the orchestrator. At the start of each epoch, the orchestrator collects all sampling queries from the sampling service. Additionally, it obtains flow statistics and their paths from the controller for the recently ended epoch. The orchestrator passes this information to the traffic estimator and, in turn, receives the mean and variance for each flow’s rate. Using the mean and variance of each flow’s rate, as well as their paths, the orchestrator leverages the optimizer module to create a sampling schedule for each flow. This schedule is then installed on network switches via the controller.

3.1.4 Traffic Estimator

To calculate an optimal sampling schedule for flows with non-deterministic rates, **dSamp** requires the mean and variance estimates of the flow rate. To this extent, various approaches can be utilized, *e.g.*, [71] employs an autoregressive model, while [10] uses machine learning to estimate flow rates. In our design, we applied a simple estimator based on past traffic statistics. Specifically, the traffic estimator queries the controller to collect statistics about flow rates over the past epochs and then estimates the mean and variance of flow rates for the upcoming epoch based on the collected statistics.

3.1.5 Optimizer

The optimizer is responsible for computing a sampling schedule while considering flow rate fluctuations. The goal is to optimize the use of switch sampling resources in order to maximize the number of fully sampled flows, *i.e.*, flows that are sampled at their specified target rate. To achieve this goal, **dSamp** limits sampling each flow to a single switch along its path. Hence, depending on the available switch capacity, a flow either could be sampled on one switch along its path or will not be sampled at all. This will satisfy two goals: 1) avoiding duplicate sampled packets, and 2) improving flow visibility, as **dSamp** only samples flows for which there is a high confidence of fully sampling them. This way, **dSamp** gives higher priority to fully sampling flows (for improved flow visibility) rather than sampling many flows but only partially. This design is advantageous for tasks requiring a specific minimum sampling rate for each individual flow. The primary focus of the following section is the design of efficient algorithms for the optimizer.

3.1.6 Discussion

While the design of **dSamp** primarily targets ISPs, it could also be altered to be used in DCs. In the DC setting, the sampling query specifies a list of VM pairs belonging to a tenant.

Similar to the case of ISP deployment, the system reconfigures switches only at the beginning of each sampling epoch. This strategy is consistent with the original design, minimizing control overhead associated with collecting flow statistics and reconfiguring switches.

One limitation of the optimizer is the potential for overloading TCAM (Ternary Content Addressable Memory) space. In a worst-case scenario, where the algorithm assigns a single switch to sample all flows, this could lead to overloading. TCAM can typically handle up to a few thousand flow entries [52]. If an operator chooses to measure thousands of flows, `dSamp` may exceed the available TCAM space.

3.2 Sampling Optimization

Our goal in this section is to design an *efficient* algorithm for computing the network-wide sampling allocations with dynamic flow rates. The algorithm must be efficient and solvable within a reasonable timeframe for realistic-sized networks. A reasonable timeframe for the optimizer is one that allows the problem to be solved within a relatively short period, ideally within milliseconds, to minimize delays. We define realistic-sized networks as small-scale networks (10 to 100 switches), medium-scale networks (100 to 300 switches), and large-scale networks (ranging from 300 to a few thousand switches).

We show that dynamic flow rates can be incorporated in the problem formulation, but they lead to second-order constraints that are computationally intensive to solve, even for small network scenarios. Consequently, we focus on developing an approximate formulation that can be utilized in large-scale networks, while achieving performance that is close to that of the exact formulation. The important notions used in this section are shown in Table 3.1.

3.2.1 Problem Formulation

Let \mathcal{F} denote the set of *flows* to be sampled in the network. We define \mathcal{S} to be the set of all switches in the network and \mathcal{S}_f to be the set of switches on the path of flow $f \in \mathcal{F}$. For

Table 3.1: Table of Notations.

Notation	Description
r_f	Sampling load of flow f
μ_f	Mean sampling load for flow f
σ_f^2	Variance of sampling load for flow f
$x_{f,s}$	Sampling decision for flow f on switch s
B_s	Sampling capacity of switch s
\mathcal{F}	Set of flows to be sampled
\mathcal{S}	Set of switches in the network
\mathcal{F}_s	Set of flows that pass through switch s
\mathcal{S}_f	Set of switches on the path of flow f
δ	Probability of violating a switch sampling capacity
α	Pre-specific target sampling rate

notation convenience, we also define \mathcal{F}_s to be a subset of flows in \mathcal{F} that pass through switch $s \in \mathcal{S}$. The network orchestrator requires each flow in \mathcal{F} to be sampled at a target sampling rate (to achieve a network monitoring objective). Consequently, the sampling load of each flow, defined as the rate of sampled traffic transmitted to the collector, is computed from its rate and its target sampling rate. Each switch has a limited capacity to send sampled packets to the collector. We use B_s to denote the transmission capacity of switch $s \in \mathcal{S}$ for sending the sampled packets. We consider a scenario where the flow rates are dynamic and fluctuate over time. Thus, we use the random variable r_f to show the sampling load of flow f . This random variable models the fluctuation of the flow's rate, and any realization of r_f is valid.

To formulate the sampling allocation problem, we define binary decision variable $x_{f,s}$, which is set to one if flow f is sampled on switch s , and to zero otherwise. To avoid duplications, where a packet is sampled redundantly on multiple switches, we require that each flow be sampled either on one switch or nowhere at all if the available sampling capacity

is not sufficient to handle the resulting sampling load. Consequently, we introduce the following constraint to ensure that each flow is sampled by at most one switch in the network:

$$\sum_{s \in \mathcal{S}_f} x_{f,s} \leq 1, \quad \forall f \in \mathcal{F}. \quad (3.1)$$

Given the stochastic nature of the flow rates, we use the following probabilistic constraints to capture the limited sampling capacity for each switch:

$$\mathbb{P}\left\{ \sum_{f \in \mathcal{F}_s} r_f \cdot x_{f,s} \leq B_s \right\} \geq 1 - \delta, \quad s \in \mathcal{S}, \quad (3.2)$$

where, $0 < \delta < 1$ is a small probability with which the network orchestrator accepts the violation of the sampling capacity of a switch. Note that the violation of the sampling capacity results in the loss of a subset of sampled packets, which translates into a lower realized sampling rate for the affected flows on that switch. Although a lower value for δ reduces the capacity violation probability, it also leads to rejecting more flows (*i.e.*, setting more $x_{f,s}$ to zero) to ensure that the available capacity is sufficient to absorb any fluctuation in the rate of those flows that are accepted for sampling (*i.e.*, flows whose corresponding $x_{f,s}$ is set to one).

Incorporating (3.2) directly into an optimization program results in a stochastic optimization, which is generally non-trivial to solve efficiently unless the exact distribution of variables r_f is known, and can be expressed via simple closed-form expressions, none of which are true in real-world settings. Instead, a common approach in the literature [49, 21, 47] is to consider a deterministic version of (3.2), as follows:

$$\sum_{f \in \mathcal{F}_s} r_f \cdot x_{f,s} \leq B_s, \quad s \in \mathcal{S}, \quad (3.3)$$

which is a linear constraint, but it entirely ignores the flow rate dynamics. To consider flow rate dynamics, several approaches can be used to make 3.2 deterministic:

1. Apply the Central Limit Theorem (CLT) [34].
2. Approximate the chance-constrained equation by concentration bounds such as Chernoff bound [18].
3. Apply the Γ -Robust optimization approach proposed by [12].

Since the target sampling rates are small in network-wide sampling, then each switch will have sufficient capacity to sample multiple flows. In such a setting, we can apply the CLT to provide a tight estimation for the tail probability in (3.2). Using this approach, we are not required to assume that only a specific number of flow rates can deviate from their mean values simultaneously, as is the case with the Γ -Robust method.

Given that flow rates r_f fluctuate over time, these fluctuations can be modeled as random variables with known means μ_f and variances σ_f^2 . According to the CLT, the sum of a large number of independent and identically distributed random variables tends to be normally distributed, regardless of the original distribution of the variables.

Thus, for a sufficiently large number of flows, the total load L_s on a switch s , defined as:

$$L_s = \sum_{f \in \mathcal{F}_s} r_f \cdot x_{f,s} \tag{3.4}$$

can be approximated by a Normal distribution:

$$L_s \sim \mathcal{N} \left(\sum_{f \in \mathcal{F}_s} \mu_f \cdot x_{f,s}, \sum_{f \in \mathcal{F}_s} \sigma_f^2 \cdot x_{f,s}^2 \right) \tag{3.5}$$

This approximation allows us to estimate the probability that the total sampled load on a switch exceeds its capacity B_s using the properties of the Normal distribution [17], leading to the following constraint:

$$\sum_{f \in \mathcal{F}_s} \mu_f \cdot x_{f,s} + z_\delta \sqrt{\sum_{f \in \mathcal{F}_s} \sigma_f^2 \cdot x_{f,s}} \leq B_s, \quad (3.6)$$

where, z_δ denotes the $(1 - \delta)$ quantile of the standard Normal distribution. Putting this all together, the network-wide sampling allocation problem with dynamic flow rates can be defined as the following integer second order cone program:

Problem 1: ISOCP (Network-Wide Sampling Allocation Problem)

$$\text{maximize} \quad \sum_{f \in \mathcal{F}} \sum_{s \in \mathcal{S}} x_{f,s}$$

subject to:

$$\sum_{s \in \mathcal{S}_f} x_{f,s} \leq 1, \quad \forall f \in \mathcal{F} \quad (3.7a)$$

$$\sum_{f \in \mathcal{F}_s} \mu_f \cdot x_{f,s} + z_\delta \sqrt{\sum_{f \in \mathcal{F}_s} \sigma_f^2 \cdot x_{f,s}} \leq B_s. \quad \forall s \in \mathcal{S} \quad (3.7b)$$

3.2.2 Linearization

Given the conic constraints (3.6) in the ISOCP formulation, it is generally infeasible to solve with off-the-shelf solvers in a reasonable timeframe for sampling in realistic-sized networks. To address the challenge posed by the conic constraints, it is possible to transform these constraints into linear ones by introducing additional auxiliary integer variables in the formulation. Our numerical experiments indicated that this transformation significantly improves the computation time when employing off-the-shelf solvers such as Gurobi [3]. To this end, we start by rewriting constraints (3.6) as follows where both sides of the inequality are squared:

$$z_\delta^2 \sum_{f \in \mathcal{F}_s} \sigma_f^2 \cdot x_{f,s} \leq (B_s - \sum_{f \in \mathcal{F}_s} \mu_f \cdot x_{f,s})^2. \quad (3.8)$$

To ensure the original constraints are enforced after this transformation, it is necessary that both sides of the inequality are non-negative. Clearly, the left-hand side is always positive, while the right-hand side is unrestricted. To resolve this, we also include the following constraints in the formulation:

$$B_s - \sum_{f \in \mathcal{F}_s} \mu_f \cdot x_{f,s} \geq 0. \quad (3.9)$$

Then, we expand the right-hand side of (3.8) to obtain:

$$B_s^2 - 2B_s \sum_{f \in \mathcal{F}_s} \mu_f \cdot x_{f,s} + \left(\sum_{f \in \mathcal{F}_s} \mu_f \cdot x_{f,s} \right)^2, \quad (3.10)$$

where, the last term can be further expanded to yield the following expression:

$$\sum_{f \in \mathcal{F}_s} \mu_f^2 \cdot x_{f,s}^2 + 2 \sum_{f, f' \in \mathcal{F}_s} \mu_f \cdot x_{f,s} \cdot \mu_{f'} \cdot x_{f',s}. \quad (3.11)$$

Now, we introduce auxiliary variables $w_{f,f',s}$ to model the multiplication of binary variables $x_{f,s}$ and $x_{f',s}$. The following constraints ensure that $w_{f,f',s}$ is only equal to one when both $x_{f,s}$ and $x_{f',s}$ are equal to one:

$$w_{f,f',s} \leq x_{f,s}, \quad (3.12)$$

$$w_{f,f',s} \leq x_{f',s}, \quad (3.13)$$

$$w_{f,f',s} \geq x_{f',s} + x_{f,s} - 1. \quad (3.14)$$

Using $w_{f,f',s}$, we can re-write (3.8) as follows:

$$\begin{aligned} z_\delta^2 \sum_{f \in \mathcal{F}_s} \sigma_f^2 \cdot x_{f,s} \leq B_s^2 - 2B_s \sum_{f \in \mathcal{F}_s} \mu_f \cdot x_{f,s} \\ + \sum_{f \in \mathcal{F}_s} \mu_f^2 \cdot x_{f,s} + 2 \sum_{f, f' \in \mathcal{F}_s} \mu_f \mu_{f'} \cdot w_{f,f',s}, \end{aligned} \quad (3.15)$$

which has a linear structure. Therefore, the original ISOCP can be transformed into the following linear program:

Problem 2: ILP (Linear Transformation of ISOCP)

$$\text{maximize } \sum_{f \in \mathcal{F}} \sum_{s \in \mathcal{S}} x_{f,s}$$

subject to:

$$\sum_{s \in \mathcal{S}_f} x_{f,s} \leq 1 \quad \forall f \in \mathcal{F}, \quad (3.16a)$$

$$B_s - \sum_{f \in \mathcal{F}_s} \mu_f \cdot x_{f,s} \geq 0, \quad \forall s \in \mathcal{S} \quad (3.16b)$$

$$w_{f,f',s} \leq x_{f,s}, \quad \forall f, f' \in \mathcal{F}_s, \forall s \in \mathcal{S} \quad (3.16c)$$

$$w_{f,f',s} \leq x_{f',s}, \quad \forall f, f' \in \mathcal{F}_s, \forall s \in \mathcal{S} \quad (3.16d)$$

$$w_{f,f',s} \geq x_{f,s} + x_{f',s} - 1, \quad \forall f, f' \in \mathcal{F}_s, \forall s \in \mathcal{S} \quad (3.16e)$$

$$\begin{aligned} z_\delta^2 \sum_{f \in \mathcal{F}_s} \sigma_f^2 \cdot x_{f,s} &\leq B_s^2 - 2B_s \sum_{f \in \mathcal{F}_s} \mu_f \cdot x_{f,s} + \sum_{f \in \mathcal{F}_s} \mu_f^2 \cdot x_{f,s} \\ &+ 2 \sum_{f, f' \in \mathcal{F}_s} \mu_f \mu_{f'} \cdot w_{f,f',s}. \end{aligned} \quad \forall s \in \mathcal{S} \quad (3.16f)$$

Theorem 3.1. *ILP requires at most $O(|\mathcal{F}|^2 \mathcal{S})$ more decision variables compared with ISOCP.*

Proof. ILP requires decision variables $x_{f,s}$ for constraints (3.1) and decision variables $w_{f,f',s}$ for the linearization process. The number of former variables is equal to the number of variables in ISOCP. However, the latter variables are new and there are at most $O(|\mathcal{F}|^2 \mathcal{S})$ of them. \square

Theorem 3.2. *Any feasible solution $\{x_{f,s}^*\}$ for ISOCP is also a feasible solution for ILP.*

Proof. Let $\{x_{f,s}^*\}$ be a feasible solution to ISOCP. We construct corresponding variables $\{w_{f,f',s}^*\}$ for ILP by setting:

$$w_{f,f',s}^* = x_{f,s}^* \cdot x_{f',s}^*, \quad \forall f, f' \in \mathcal{F}_s, \forall s \in \mathcal{S}.$$

We need to verify that $\{x_{f,s}^*, w_{f,f',s}^*\}$ satisfy all the constraints of ILP.

- **Constraint 3.1:** This constraint is identical in both problems and is satisfied by $x_{f,s}^*$.
- **Constraint 3.9:** In ISOCP, the left-hand side of the conic constraint (3.6) is less than or equal to B_s , so:

$$B_s - \sum_{f \in \mathcal{F}_s} \mu_f \cdot x_{f,s}^* \geq 0.$$

- **Constraints (3.12 - 3.14):** For the constructed $w_{f,f',s}^* = x_{f,s}^* x_{f',s}^*$, these constraints are satisfied:

- If $x_{f,s}^* = x_{f',s}^* = 1$, then $w_{f,f',s}^* = 1$, and the constraints become:

$$w_{f,f',s}^* \leq x_{f,s}^* \implies 1 \leq 1,$$

$$w_{f,f',s}^* \leq x_{f',s}^* \implies 1 \leq 1,$$

$$w_{f,f',s}^* \geq x_{f,s}^* + x_{f',s}^* - 1 \implies 1 \geq 1 + 1 - 1 \implies 1 \geq 1.$$

- If either $x_{f,s}^* = 0$ or $x_{f',s}^* = 0$, then $w_{f,f',s}^* = 0$, and the constraints become:

$$w_{f,f',s}^* \leq x_{f,s}^* \implies 0 \leq 0 \text{ or } 0 \leq 1,$$

$$w_{f,f',s}^* \leq x_{f',s}^* \implies 0 \leq 0 \text{ or } 0 \leq 1,$$

$$w_{f,f',s}^* \geq x_{f,s}^* + x_{f',s}^* - 1 \implies 0 \geq 1 - 1 \implies 0 \geq 0.$$

- If $x_{f,s}^* = x_{f',s}^* = 0$, then $w_{f,f',s}^* = 0$, and the constraints become:

$$w_{f,f',s}^* \leq x_{f,s}^* \implies 0 \leq 0,$$

$$w_{f,f',s}^* \leq x_{f',s}^* \implies 0 \leq 0,$$

$$w_{f,f',s}^* \geq x_{f,s}^* + x_{f',s}^* - 1 \implies 0 \geq 0 + 0 - 1 \implies 0 \geq -1.$$

All of these inequalities hold true.

- **Constraint 3.15:** The conic constraint (3.6) in ISOCP ensures that:

$$z_\delta^2 \sum_{f \in \mathcal{F}_s} \sigma_f^2 \cdot x_{f,s}^* \leq \left(B_s - \sum_{f \in \mathcal{F}_s} \mu_f \cdot x_{f,s}^* \right)^2.$$

Expanding the right-hand side and using $w_{f,f',s}^* = x_{f,s}^* x_{f',s}^*$, we have:

$$z_\delta^2 \sum_{f \in \mathcal{F}_s} \sigma_f^2 \cdot x_{f,s}^* \leq B_s^2 - 2B_s \sum_{f \in \mathcal{F}_s} \mu_f x_{f,s}^* + \sum_{f \in \mathcal{F}_s} \mu_f^2 x_{f,s}^* + 2 \sum_{f, f' \in \mathcal{F}_s} \mu_f \mu_{f'} w_{f,f',s}^*.$$

which is exactly constraint (3.15) of ILP.

Thus, any feasible solution to ISOCP can be extended to a feasible solution for ILP. □

3.2.3 Approximation

While experimenting with ILP, we found that even this formulation is computationally too expensive, and can be solved only for small-sized problem instances using Gurobi. Thus, we turned our attention to developing an approximation formulation that can be applied to large-scale problem instances. To this end, notice that the following inequality is always satisfied:

$$\sqrt{\sum_{f \in \mathcal{F}_s} \sigma_f^2 \cdot x_{f,s}} \leq \sum_{f \in \mathcal{F}_s} \sigma_f \cdot x_{f,s}. \quad (3.17)$$

Using this inequality, we can rewrite (3.2) as follows:

$$\sum_{f \in \mathcal{F}_s} \mu_f \cdot x_{f,s} + z_\delta \sum_{f \in \mathcal{F}_s} \sigma_f \cdot x_{f,s} \leq B_s, \quad (3.18)$$

which is linear. By substituting the above linear constraints in the original ISOCP, we obtain the following formulation:

Problem 3: APX (Approximation of ISOCP)

$$\text{maximize } \sum_{f \in \mathcal{F}} \sum_{s \in \mathcal{S}} x_{f,s}$$

subject to:

$$\sum_{s \in \mathcal{S}_f} x_{f,s} \leq 1, \quad \forall f \in F \quad (3.19a)$$

$$\sum_{f \in \mathcal{F}_s} \mu_f \cdot x_{f,s} + z_\delta \sum_{f \in \bar{\mathcal{F}}_s} \sigma_f \cdot x_{f,s} \leq B_s. \quad \forall s \in \mathcal{S} \quad (3.19b)$$

which is a linear program with the *same number of decision variables* as in the original formulation.

Chapter 4

Evaluations

In this chapter, we evaluate the performance of **dSamp** through extensive ns-3 simulations. In Section 4.1, we detail the process of implementing the simulations in ns-3. Subsequently, in Section 4.2, we present a set of benchmarks to evaluate the algorithms used in **dSamp**. **dSamp** is useful for various network management tasks (e.g., anomaly detection [70]) and is particularly effective in scenarios where not sampling a flow is better than using low-rate sampling, such as in detecting DDoS attacks [21].

4.1 Implementation

4.1.1 Why ns-3?

ns-3, or Network Simulator 3, is a discrete-event network simulator primarily used for research and educational purposes [6]. It is an open-source project that provides a flexible environment for networking research and is widely adopted in both academia and industry. ns-3 offers a modular architecture that allows researchers to construct and simulate complex network scenarios by combining different modules. It supports various network protocols, including IPv4, IPv6, TCP, and User Datagram Protocol (UDP), and various routing protocols, making it suitable for both wired and wireless network simulations. In addition, ns-3

provides extensive tracing and logging capabilities, which are essential for analyzing network behavior.

While there are several network simulators like Mininet, OMNeT++, Estinet, and Flow Simulator, we chose ns-3 due to its scalability, low memory usage, and the capability for packet-level analysis, which are crucial for SDN simulations, especially in large-scale environments. Notably, ns-3 performs better than Mininet in large-scale benchmarks [32].

4.1.2 Workflow

There are two primary workflows in ns-3 simulations: model-driven and trace-driven.

Model-Driven. In a model-driven workflow, the ns-3 controller first reads a config file that specifies switch sampling capacities, the chosen algorithm (*algorithm_name*), the epoch number being simulated (*epoch_index*), and the (*run_number*) which is used as a seed for traffic generation. The controller (`controller.h`) then populates the vector for storing sampling allocations using the `FillMap()` function. It reads these allocations from a folder named *Algorithms_Model/{algorithm_name}*, where *algorithm_name* is specified in the config file. Following this, the `StartFlow()` function is initiated based on the flow information file stored in *flow_direction_{epoch_index}.txt*, where *epoch_index* indicates that these flows are for a specific epoch, as previously set in the config file.

Flow information includes the mean, Coefficient of Variation (CoV), source switch, and destination switch. A Python program called `FlowGenerator.py` generates the flow information for each epoch once and stores it in a file. Based on this flow information and a fixed topology provided to the Python optimizer, the optimizer generates the sampling allocations. This approach allows sampling allocations for each epoch to be determined and stored in separate files based on the *epoch_index*, enabling ns-3 to apply the sampling allocations and run the traffic generator for each epoch by simply reading these files. This process eliminates the need for ongoing communication with the Python optimizer, thereby reducing coding complexity. Finally, the `Measure()` function will write flow measures to a

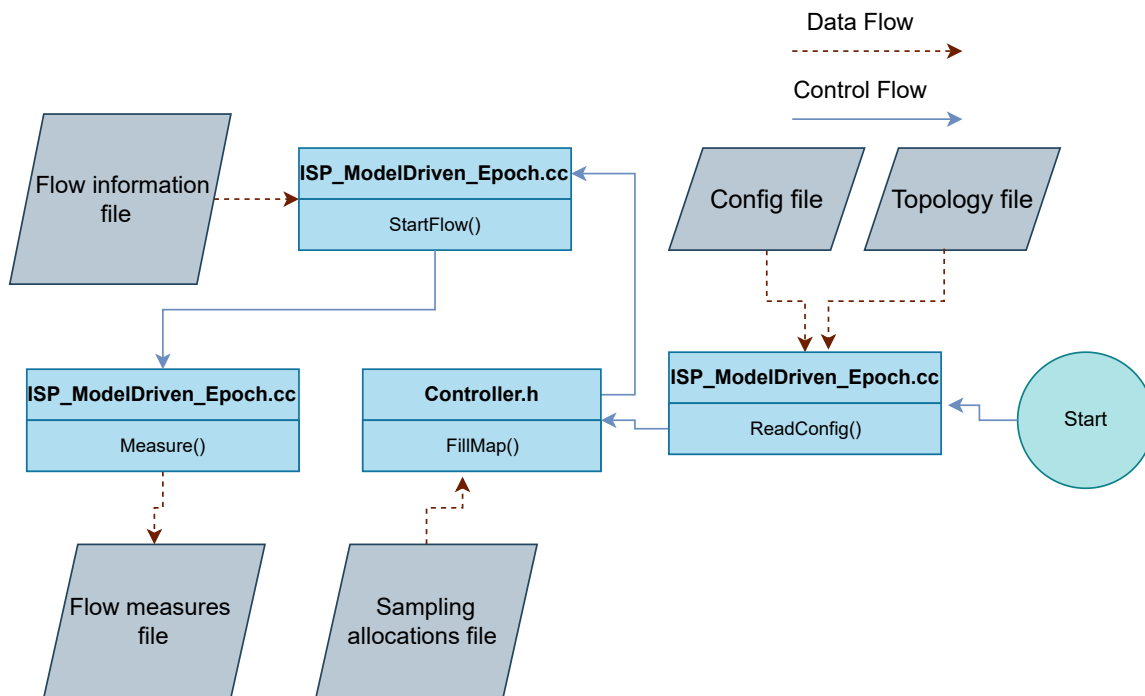


Figure 4.1: Model-Driven ns-3 Workflow.

file for later analysis. The whole workflow is illustrated in 4.1.

Trace-Driven. In a trace-driven setup, the workflow is mostly the same as in the model-driven workflow, except that instead of using the CoV and mean rates in ns-3, the actual rates are taken from a file named *Rates.txt*, which can be sourced from real-world ISP traces or data center traces. These flow rates are expected to change every 100ms for each flow. Similar to the model-driven scenario, this approach allows sampling allocations for every epoch to be determined and stored in separate files, enabling the ns-3 simulator to apply the sampling allocations and flow rates for each epoch by simply reading these files.

To make the simulation more realistic, a delay can be introduced at the beginning of each epoch to account for the time required for the controller to send the flow statistics pull request, the optimizer to calculate the sampling allocations, and the controller to receive the response. In the subsection on trace-driven simulations (refer to 4.2.9), we will demonstrate the impact of adding this delay.

```

1  NodeContainer switches, hosts;
2  NetDeviceContainer linkDevices;
3  switches.Create(10);
4  switches.Create(10);
5  linkDevices.Add(p2p.Install(NodeContainer(hosts.Get(0), switches.Get
    (0))

```

Listing 4.1: Node creation

4.1.3 Network Components

Hosts and Switches

We leverage the `NodeContainer` class for initializing switches. The `NodeContainer` class is a utility provided by ns-3 to store and manage a collection of nodes. Nodes in ns-3 represent network devices such as computers, routers, switches, etc., which participate in the simulation. By using the `Create()` function in the `NodeContainer` class, we can create several nodes. For example, in the following code snippet 4.1, line 2 initializes a container to store network interfaces. Lines 3-4 create 10 hosts and 10 switches. Line 5 creates a point-to-point link between the first host and the first switch, installs network devices on both nodes, and adds these devices to the `linkDevices` container for later use.

Controller

We wrote a custom controller class to act as an OpenFlow controller. The controller will retrieve the sampling allocations from `{algorithm_name}-{epoch_index}.txt`, located in the folder `Algorithms_Model/{algorithm_name}`. These allocations vary based on `epoch_index` and `algorithm_name`, which are defined in the config file.

These allocations specify which switch should sample each flow and the corresponding probability. Code snippet 4.2 presents the definition of sampling allocations with objects. Specifically, `Flow` consists of four nested pairs that determine each flow. Here, a flow is identified by a 5-tuple `[srcPort, dstPort, srcIP, dstIP, Protocol]`. However, by only considering `srcIP` and `dstIP` for flow sampling, we can focus on OD pairs. The `SwitchConfig` vector

```

1 typedef pair<int, int> SdPort;
2 typedef pair<Ipv4Address, Ipv4Address> SdIp;
3 typedef pair<std::string, SdPort> ProtocolSdPorts;
4 typedef pair<SdIp, ProtocolSdPorts> Flow;
5 vector<std::map<Flow, long double>> SwitchConfig;

```

Listing 4.2: Sampling Allocations Vector

```

1     for (int j = 0; j < host_len; j++) {
2         internet.Install(switches.Get(j));
3         internet.Install(hosts.Get(j));
4     }
5     NS_LOG_INFO("Assigning IP addresses.");
6     Ipv4AddressHelper ipv4;
7     Ipv4InterfaceContainer interfaces;
8 and   int index = 0;
9       int first = 0;
10      int second = 0;
11      for (int j = 0; j < int(linkDevices.GetN()); j += 2) {
12          index = j / 2;
13          first = index % 256;
14          second = index / 256;
15          std::string base = "10." + std::to_string(second) + "." + std
16              ::to_string(first) + ".0";
17          ipv4.SetBase(base.c_str(), "255.255.255.0");
18          interfaces.Add(ipv4.Assign(NetDeviceContainer(linkDevices.Get(
19              j), linkDevices.Get(j + 1))));
20      }
21      Ipv4GlobalRoutingHelper::PopulateRoutingTables();
22      std::cout << "Global routing is done " << std::endl;

```

Listing 4.3: Routing

maps the flow to the sampling probability, specifying the target sampling rate of a flow.

4.1.4 Functions

Routing

After creating point-to-point connections between switches and hosts based on the defined network topology, routing is established as illustrated in code snippet 4.3.

First in lines 1-4, we install the internet stack (e.g., TCP/IP protocols) on each switch and host, enabling them to communicate using standard networking protocol.

In lines 5-18, unique subnets are assigned to each pair of connected network devices. Each connection between a host and a switch (or between switches) involves two network interfaces within the same subnet. Specifically, the object `Ipv4InterfaceContainer` is used, which holds a collection of IP address assignments. These assignments represent the IP addresses of network interfaces that are connected via point-to-point links.

Finally, in lines 19-20, we automatically set up global routing tables across all nodes (hosts and switches) in the network. Routing is performed automatically using Dijkstra's algorithm to populate global routing tables for all switches and hosts in the network. This allows the network to forward packets between different switches.

Traffic Generation

As previously mentioned, traffic will be generated based on a truncated normal distribution with a specific mean and variance. In our simulations, we will simulate traffic using only UDP packets. Code snippet 4.4 shows the traffic generation function for the model-driven simulations (`StartFlow()`). This function will start each flow based on its source IP and destination IP. In line 2, the seed number will be set based on the seed set by user in the config file (*run_number*). The `PacketSinkHelper` at line 4 is set up for each receiving host, ensuring that the packets sent by the On-Off Application (`onOffAP`) are not dropped and are received at the destination.

To generate traffic from a source IP to a destination IP, the On-Off Application will be used. The instance is created at line 14, followed by setting different configurations such as `PacketSize`, `OnTime`, and `OffTime`. We assume that the On-Off Application is always on, therefore the `OffTime` is set to 0. Line 9 creates an object `x`, for which the mean and variance should be set, and it will generate flow rates based on these values. This object will produce a specific number each time the `getValue()` function is called. We will use an if condition to reject negative values and regenerate them until a positive number is produced, ensuring a truncated normal distribution. It is noteworthy to mention that the packet size

```

1      static void StartFlow(Ipv4InterfaceContainer interfaces, uint16_t
2          port, NodeContainer hosts, float start, float end, int
3          flow_source, int flow_dest, double flow_mean, double variance) {
4          RngSeedManager::SetSeed(port);
5          RngSeedManager::SetRun(run_number);
6          PacketSinkHelper sinkHelper("ns3::UdpSocketFactory",
7              InetSocketAddress(interfaces.GetAddress(flow_dest*2), port)
8          );
9          ApplicationContainer sinkApp = sinkHelper.Install(hosts.Get(
10             flow_dest));
11          sinkApp.Start(Seconds(start));
12          sinkApp.Stop(Seconds(end+0.1));
13          float step = 0.1;
14          Ptr <NormalRandomVariable> x = CreateObject<
15             NormalRandomVariable>();
16          cout<<" Mean = " << flow_mean<<endl;
17          x->SetAttribute("Mean", DoubleValue(flow_mean));
18          x->SetAttribute("Variance", DoubleValue(variance));
19          double value = 0;
20          OnOffHelper onOffAP("ns3::UdpSocketFactory", InetSocketAddress
21             (interfaces.GetAddress(flow_dest*2), port));
22          onOffAP.SetAttribute("PacketSize", UintegerValue(1024 - 30));
23          onOffAP.SetAttribute("OffTime",StringValue("ns3::
24             ConstantRandomVariable[Constant=0]"));
25          onOffAP.SetAttribute("OnTime",StringValue("ns3::
26             ConstantRandomVariable[Constant=4]"));
27          for (float s =start ; s<end ;s = s+step){
28              value = x->GetValue();
29              if (value <= 0) {
30                  cout<<"Negative value"<<endl;
31                  while (value <= 0){
32                      value = x->GetValue();
33                  }
34                  total_Negatives++;
35              }
36          onOffAP.SetAttribute("DataRate", StringValue((to_string(
37             value) + "KBps")));
38          ApplicationContainer clientApps;
39          clientApps = onOffAP.Install(hosts.Get(flow_source));
40          clientApps.Start(Seconds(s));
41          clientApps.Stop(Seconds(s+step));
42      }

```

Listing 4.4: Model-Driven Flow Generation

```

1  string Rate_file = "Rates.txt";
2  ifstream MyReadFile(Rate_file);
3  NS_LOG_INFO("Reading rate file");
4  int counter = 0;
5  vector<string> out;
6  while (getline(MyReadFile, myText)) {
7      if (counter == line)
8          {
9              const char delim = ' ';
10             tokenize(myText, delim, out);
11             break ;
12         }
13     counter +=1;
14 }
15 float step = 0.1;
16 counter = (index_epoch) * end_time * int(1 / step);
17 cout<<counter<<endl;
18 double value = 0;
19 for (float s =start ; s<end ;s = s+step){
20     value = stod(out[counter])/(1024*8);
21     onOffAP.SetAttribute("DataRate", StringValue (to_string(value) +
22         "KBps"));
23     ApplicationContainer clientApps;
24     clientApps = onOffAP.Install(hosts.Get(flow_source));
25     clientApps.Start(Seconds(s));
26     clientApps.Stop(Seconds(s+step));
27     counter++;
}

```

Listing 4.5: Trace-Driven Flow Generation

we defined is 1024 bytes (1 KB). As seen in line 15, we decrease the size by 30 bytes because the On-Off Application considers the packet size as only the payload, excluding the header.

Traffic generation in the trace-driven scenario is similar to the model-driven approach, except that in the trace-driven scenario, all flow rates are stored in a text file named *Rates.txt*. This file contains the rates for each flow, which change every 100 milliseconds. Given that each epoch is 5 seconds by default, there will be 50 numbers for each flow in each epoch (10 intervals per second for 5 seconds). Code snippet 4.5 shows the code for trace-driven traffic generation. Rates are read and stored in the `value` variable every 100 milliseconds for each flow.

```
1 Config::Connect("/NodeList/*/DeviceList*/$ns3::  
    PointToPointNetDevice/PhyRxEnd", MakeBoundCallback(&Receive,  
    global_capacity));
```

Listing 4.6: Packet Reception Callback for Point-to-Point Devices

Flow Sampling

Line 1 in code snippet 4.6 is used to connect a callback function to a specific trace source, which allows the simulator to trigger a user-defined function (callback) when a particular event occurs. Trace sources will notify when certain events occur during the simulation, such as packets being sent or received.

`Config::Connect` is a method used in ns-3 to connect a trace source to a callback function. This method links an event (specified by a path) to a function that will be called whenever the event happens. We specified this method to listen for the `PhyRxEnd` event (the end of packet reception) on all nodes that have `PointToPointNetDevice` interfaces.

To implement the sampling function in ns-3, we defined `Receive` function which is triggered whenever a switch receives a packet. The function first matches the packet with the corresponding entry in the sampling allocations vector, then samples it based on a probability, if there is sufficient sampling capacity available in the switch. Code snippet 4.7 shows the process of matching a packet with the sampling allocations vector. Upon receiving a packet, the function first checks which switch received the packet based on the `context` string. Next, it verifies if there is an existing entry in the sampling allocations vector for the packet. If no entry exists, the packet will not be considered for sampling. Otherwise, the packet will undergo further analysis.

Code snippet 4.8 shows the process for further analysis. In line 4, the `random_generator` function generates a number between 1 and 1000 for every UDP packet, using different seeds based on the packet's corresponding flow and the switch that received it. If this number is less than the product of the sampling probability and 1000, the function will further check

```

1     PppHeader pppHeader;
2     copy->RemoveHeader(pppHeader);
3     Ipv4Header ipHeader;
4     copy->RemoveHeader(ipHeader);
5     UdpHeader udpHeader;
6     copy->RemoveHeader(udpHeader);
7     map<IP_UdpPorts , long double>::iterator it;
8     double percent = 0;
9     const char delim = '/';
10    int flag = 0;
11    vector<string> temp;
12    tokenize(context, delim, temp);
13    string device = temp[2];
14    if (stoi(device) < total_switches)
15    {
16        return;
17    }
18    else
19    {
20        device = to_string(stoi(device) - total_switches);
21    }
22    it = SwitchConfig.at(stoi(device)).find(IP_UdpPorts(SD_IP(ipHeader
23    .GetSource(), ipHeader.GetDestination()), UdpPorts("udp" , SD_Port
24    (4000, int(udpHeader.GetDestinationPort()))));
25    if (it != SwitchConfig.at(stoi(device)).end())
26    {
27        percent = it->second;
28        flag = 1 ;
29    }
30    if (flag == 0) {
31        cout<<"Packet cannot be matched with any entry"<<endl;
32        return;
33    }

```

Listing 4.7: Packet Matching

whether the sampling capacity of the switch has been reached. If the capacity limit has been reached, the packet will not be sampled. However, if the switch still has available sampling capacity, the packet will be sampled, the statistics related to the switch and flow will be updated (lines 9, 13, 17, and 20), and the packet will be saved to a Pcap file (lines 12 and 18) specific to that switch.

The sampling capacity of a switch is determined by the `switch_buffer` vector, which stores the arrival time of each sampled packet at that switch. In lines 7-14, whenever a new packet arrives, if the sampling capacity of the switch is full and the time difference

```

1      Time time_arrived = Simulator::Now();
2      double current_time = time_arrived.GetSeconds();
3      int flow_index = it->first.second.second - 4000;
4      int r = random_generator.at(flow_index* total_switches + stoi(
5          device)).operator()();
6      if (r <= percent * 1000) {
7          int device_num = stoi(device) ;
8          if (int(switch_buffer.at(device_num).size())== capacity and
9              current_time - switch_buffer.at(device_num).at(0) > 1)
10             {
11                 total_sampled.at(device_num)+=1;
12                 switch_buffer.at(device_num).erase(switch_buffer.at(
13                     device_num).begin());
14                 switch_buffer.at(device_num).push_back(current_time);
15                 files.at(device_num)-> Write(time_arrived, p);
16                 flow_samples.at(flow_index)++;
17             }
18         else if (int(switch_buffer.at(device_num).size()) < capacity)
19             {
20                 total_sampled.at(device_num) += 1;
21                 files.at(device_num)-> Write(time_arrived, p);
22                 switch_buffer.at(device_num).push_back(current_time);
23                 flow_samples.at(flow_index)++;
24             }
25         else{
26             rejected.at(device_num)++;
27             cout<<"Packet discarded device " << device<<endl;
28         }
29     }
30 }

```

Listing 4.8: Packet Sampling

between the latest sampled packet's arrival and the current packet's arrival is greater than 1 second, the packet should be sampled, and the latest sampled packet's arrival time should be updated. Lines 15-21 allow a packet to be sampled whenever the sampling capacity of the switch is not full.

Three statistics are collected in `Receive` function. The number of packets sampled by each switch is tracked by (`total_sampled`), while (`rejected`) tracks the number of packets not sampled by each switch due to capacity constraints. (`flow_samples`) tracks the number of sampled packets for each flow.

```

1      static void SinkTx(string context,Ptr<const Packet> p)
2      {
3          const char delim = '/';
4          vector<string> temp;
5          tokenize(context, delim, temp);
6          string device = temp[4];
7          string source = temp[2];
8          int sum = 0;
9          int num_temp = (stoi(device) - sources_index.at(stoi(source)))
10         /(end_time*10);
11         for (int i=0;i<int(flows_dir.size()); i++){
12             vector <int> info = flows_dir.at(i);
13             if (stoi(source) == info[0]){
14                 break;
15             }
16             sum = sum +info [2];
17         }
18         total_sent.at(num_temp+sum) ++;
19     }

```

Listing 4.9: Packet Transmission Sinking

Flow Measuring

The `SinkTx` function (4.9) can track which host sent each packet by analyzing the first argument passed to it (`context`). Similar to the `Receive` function, `SinkTx` is a callback function, but its trace source is different, originating from all On-Off applications. Therefore, whenever a host sends a packet, this function is called and increments the `total_sent` count for the flow to which the packet belongs.

The `Measure()` function (4.10) is scheduled to execute at the end of each epoch. For each flow, the sampling rate can be measured by dividing the number of packets sampled (`flow_samples`) by the total number of packets sent by that flow (`total_sent`). These sampling rates can then be gathered and visualized.

```

1   void Measure()
2   {
3       float calc_eps = 0;
4       for (int k=0;k<num_flows;k++) {
5
6           if (flow_samples.at(k) == 0 && total_sent.at(k) == 0){
7               calc_eps = 0;
8           } else {
9               calc_eps = flow_samples.at(k) / float(total_sent.at(k)
10              );
11              Output<<float(total_sent.at(k))<<"Total at flow "<<k<<
12              endl;
13          }
14      }
15  }

```

Listing 4.10: Sampling Rates Calculation

4.2 Benchmarks

To study the performance of `dSamp`, we present two sets of evaluation results:

- *Micro Benchmarks:* We start with some benchmarks to study the behavior of the approximate algorithm `APX` in various settings in order to understand its sampling accuracy, computational efficiency, and sensitivity to various design assumptions, such as the distribution of traffic rates. Additionally, we provide general benchmarks that show the effect of epoch length, violation capacity, and rate variability on all algorithms.
- *Macro Benchmarks:* We present a set of model-driven simulations to compare `APX` with existing sampling solutions presented in [50] and [21]. Model-driven simulations provide us with the flexibility to simulate a range of network conditions without being restricted to a specific setup as in trace-driven simulations (which consider a traffic trace collected over a specific time period from one ISP). Specifically, we use model-driven experiments to fine-tune the baseline algorithms that are used for comparison, and use these fine-tuned versions in our trace-driven experiments. Finally, we present a set of trace-driven simulations to demonstrate the ability of `APX` to outperform existing sampling solutions using a real-world traffic trace collected from an ISP backbone [5].

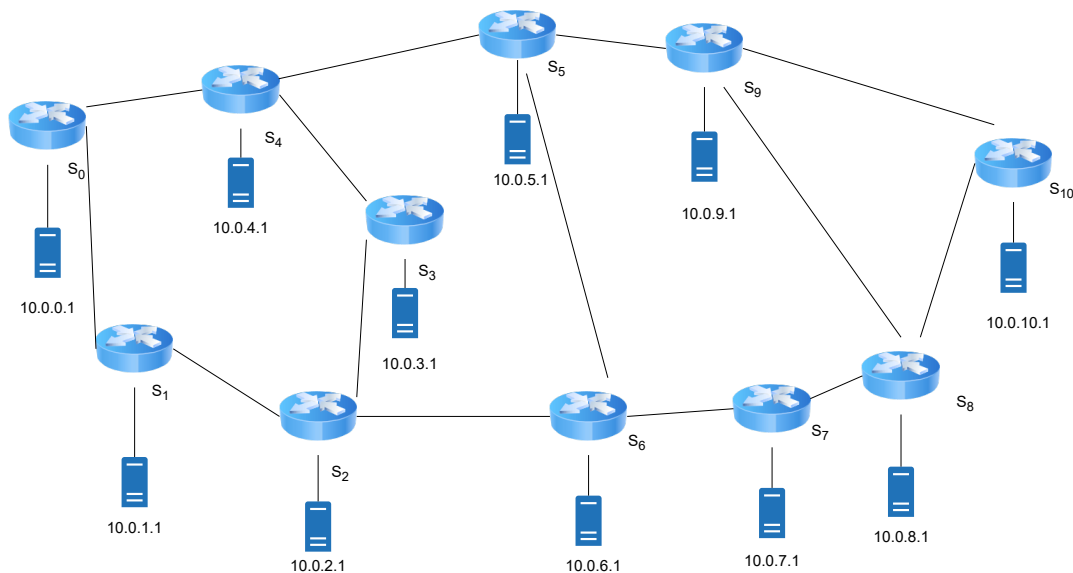


Figure 4.2: Abilene topology used in our trace-driven and model-driven simulations. Link speeds are 10 Gbps.

Setup. Micro and Macro benchmarks are implemented using Python v3.9 and ns-3 v3.40. The simulations were conducted on a standard desktop machine, equipped with an Intel(R) Core(TM) i9-12900 CPU @ 2.40 GHz and 16 GB of RAM. The *default* settings are as follows:

- *Network:* We have used the Abilene ISP topology, selected from the Topology Zoo dataset [9]. The topology consists of 11 nodes and 14 links, with all link capacities set to 10 Gbps. As the link capacity increases, the number of packets to be processed increases accordingly, leading to longer simulation time in ns-3. For example, simulating link capacities of 100 Gbps can take up to ten times longer compared to 10 Gbps. The topology is illustrated in Figure 4.2.
- *Traffic:* The ns-3 On-Off Application was used to generate traffic based on the input traffic model. The packet size is set to 1 KB in all simulations. For each flow, the source and destination are chosen at random. To describe the burstiness of traffic, we use the Coefficient of Variation (CoV), defined as $\text{CoV} = \sigma/\mu$. A high CoV indicates high variability in flow rates, whereas a low CoV suggests low variability.

Parameter	Value
Network Topology	Abilene
Link Capacities	10
Packet Size	1
δ	0.2
α	0.1
MIPGap	0

Table 4.1: Default simulation parameters.

- Sampling:* By default, the switch sampling capacity violation probability (δ) is set to 0.20 while the target sampling rate (α) is set to 0.1. However, it is possible to have different target sampling rates for each flow. In general, increasing the target sampling rate leads to higher performance in various network applications [21]. However, operators must be aware of potential degradation in switch forwarding performance [59]. Additionally, setting a higher target sampling rate may result in fewer flows being sampled due to the limitations of the available sampling capacity. The duration of each sampling epoch is set to 5 seconds by default. The results are computed as the average of 5 independent simulation runs.
- Gurobi:* By default, the Gurobi parameter called MIPGap (Mixed-Integer Programming Gap) is set to 0%. To test the performance of ILP in large-scale scenarios, we will set it to 10%. The MIPGap measures how close the current solution is to the optimal solution in the context of Mixed-Integer Programming (MIP). By increasing the MIPGap, the solver may terminate faster because it doesn't need to exhaustively search the entire solution space.

The above default values are shown in table 4.1.

Implemented Algorithms. In addition to ISOCP, ILP and APX, we have implemented the following algorithms for comparison:

- Deterministic Sampler (DS):** This algorithm makes sampling decisions solely based on the mean flow rates.

- **Deterministic with Headroom ($DS+2\sigma$):** This algorithm is similar to **DS** except that it adds a 2σ headroom to flow rates in order to allow the sampling algorithm to absorb flow rate fluctuations at runtime. It is the approach proposed in [21] to deal with dynamic traffic rates. Notice that, for a Normal distribution, 95% of samples reside within 2σ of the mean.
- **cSamp+ ϵ ($cS+\epsilon$):** This algorithm is presented in [50]. It assumes that the fluctuation in traffic rates around their means is bounded by some $\epsilon > 0$. It then scales the solution computed by the **DS** algorithm to account for the rate fluctuations that can happen in the worst case.

At a high level, **APX** also reserves some headroom on each switch to deal with rate fluctuations. However, in **APX**, the amount of headroom depends on the actual variability (variance) of flow rates, while in **DS+2 σ** , it is a fixed value. Also, while the headroom in **cSamp** depends on the actual flow rates, it is reserved based on the worst case fluctuations of rates leading to a consistently more conservative behavior than **APX**, as we will show later in this section.

Performance Metrics. The main performance metric, referred to as “Fully Sampled Flows”, is the number of flows that are fully sampled by the sampling algorithm, i.e., whose target sampling rate is satisfied by the sampling algorithm. This metric concisely captures the sampling accuracy achieved by an algorithm. However, to provide more insights about the performance of each sampling algorithm, we also report the following metrics:

- *Admitted Flows:* The number of flows that are chosen by the algorithm for sampling. The sampled flows may not be sampled at their desired target sampling rate.
- *Sampling Rate:* The actual sampling rates measured from the simulations. We would like the measured Sampling Rate to be close to the target sampling rate. We use two types of plots to present the measured sampling rates, a box plot and a CDF plot. Flows with zero sampling rates in the simulations are excluded from the plots.

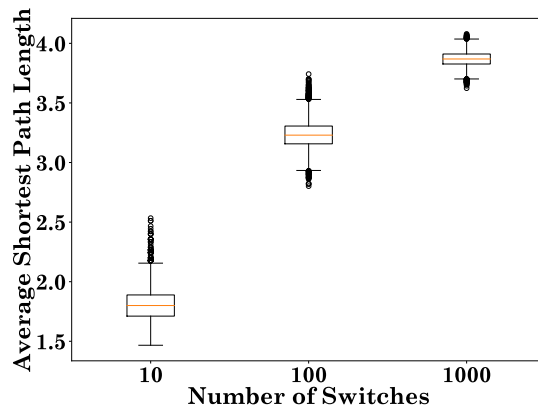


Figure 4.3: Average shortest path lengths for scale-free networks generated using the approach in [26] with varying numbers of switches.

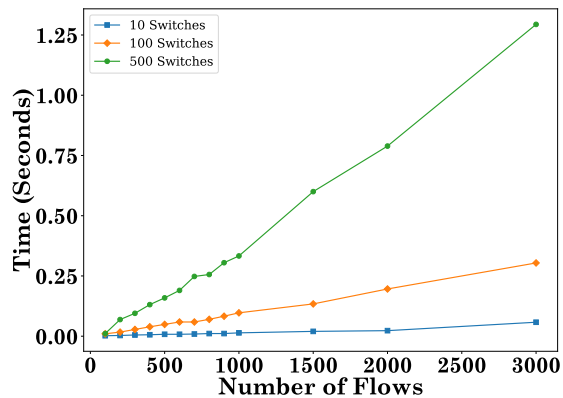


Figure 4.4: Measured runtime of APX.

4.2.1 APX Scalability Analysis

We measure the time for Gurobi solver to find a solution for APX in different network scenarios. We utilized the NetworkX Python library to generate random scale-free networks, which is implemented based on the methodology described in [26]. This algorithm returns an undirected graph resembling the Internet Autonomous System (AS) network.

Figure 4.3 presents the average shortest path lengths obtained from 10,000 different runs using switch sets of 10, 100, and 1,000. This plot demonstrates that generating various scale-free networks does not result in significant changes in the average shortest path lengths. For instance, with 100 nodes, the average shortest path length only varies from approximately 2.6

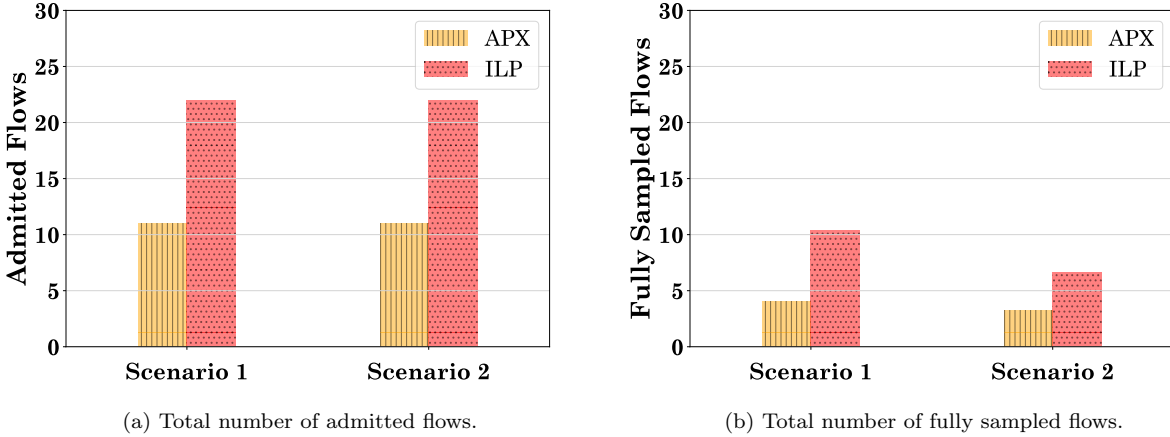


Figure 4.5: Performance comparison between APX and ILP.

Table 4.2: Runtime comparison between APX and ILP.

	(Number of Flows, Switch Sampling Capacity)			
	(50, 70)	(100, 70)	(500, 70)	(100, 100)
APX	0.001s	0.003s	0.005s	0.004s
ILP	33.62s	268.19s	>600s	>600s

to 3.7. Therefore, in our scalability analysis, we will randomly generate different scale-free networks and focus solely on the number of flows and switches, without considering whether variations in average path lengths might impact the algorithm’s runtime.

For our runtime analysis, we set all the switch capacities to 100 pps. The results are presented in Fig. 4.4. While the computation time increases as the number of switches and flows increases, it remains under a second for 100 and 500 switches, and only increases to 1.25 seconds for the large-scale network with 500 switches. We tested all instances with DS, DS+2 σ , cSamp+100, cSamp+150, and cSamp+200, and in all cases, the runtime did not exceed 1.25 seconds. It is worth mentioning that Gurobi failed to resolve the ISOCP problem for all instances within 600 seconds, while for ILP problem, it managed to solve the smallest instance within that timeframe.

Scenario	Number of flows	Switch Sampling Capacity
1	20	70
2	50	70
3	100	70

Table 4.3: Low variation setting (CoV = 0.2).

Scenario	Number of flows	Switch Sampling Capacity
1	20	200
2	50	200
3	100	200

Table 4.4: Mixed variation setting (CoV = [0.2, 0.5, 2]).

Scenario	Number of flows	Switch Sampling Capacity
1	20	200
2	50	200
3	100	200

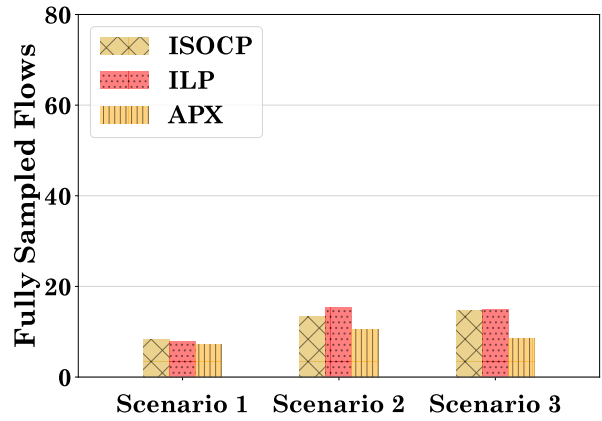
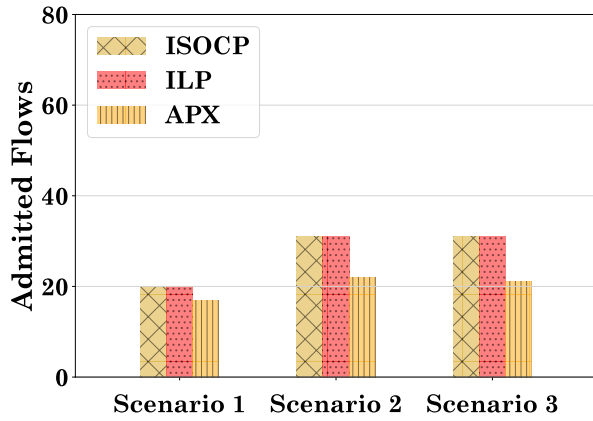
Table 4.5: High variation setting (CoV = 2).

4.2.2 APX Comparison with ILP and ISOCP

We will begin by comparing APX and ILP across various scenarios. Following that, we will extend our simulations to include ISOCP for a broader comparison.

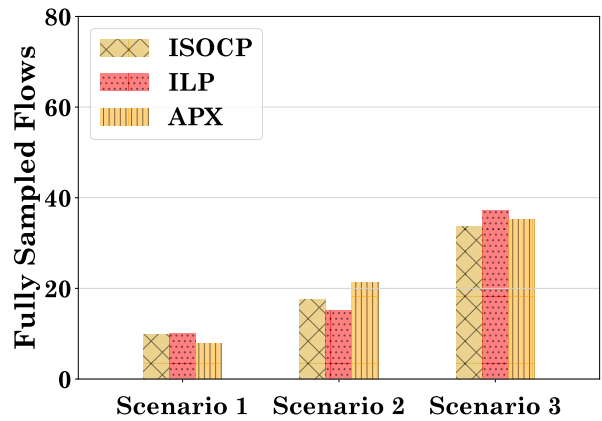
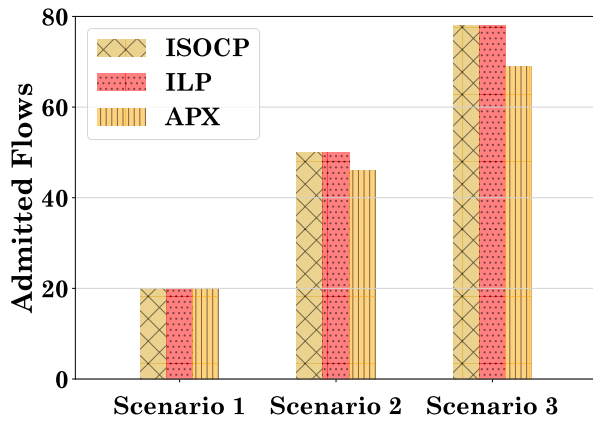
APX vs ILP: To compare APX with ILP, four sampling scenarios were simulated. Each scenario corresponds to a different combination of number of flows and switch sampling capacity as presented in Table 4.2. The flow rates were set at 200 KBps, each with a CoV of 1. The performance results are presented in Fig. 4.5. Only the results for the first two scenarios are presented, as ILP was unable to solve the other two scenarios within the allocated time (600 seconds). As can be seen, in both scenarios, ILP achieves a better performance than APX. However, as shown in Table 4.2, the higher performance of ILP comes at a huge cost in terms of the runtime of the algorithm. Specifically, while ILP can be applied only to small-scale scenarios, it takes APX a few milli-seconds to solve each scenario.

APX vs ILP vs ISOCP: To ensure a fair comparison across all three algorithms, we selected scenarios that could be solved by each within a reasonable time frame. We evaluated



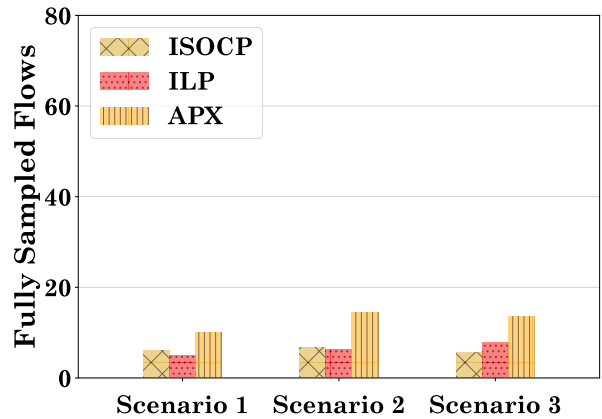
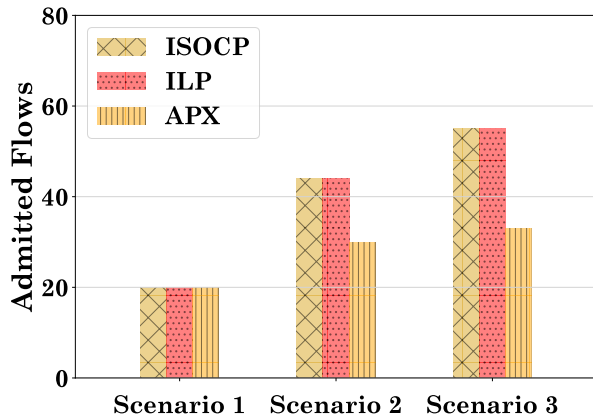
(a) Total number of admitted flows (Low variation scenario).

(b) Total number of fully sampled flows (Low variation scenario).



(c) Total number of admitted flows (Mixed variation scenario).

(d) Total number of fully sampled flows (Mixed variation scenario).



(e) Total number of admitted flows (High variation scenario).

(f) Total number of fully sampled flows. (High variation scenario).

Figure 4.6: Performance comparison between APX, ILP and ISOCP under settings of low variation, mixed variation, and high variation in flow rates.

these algorithms under three different settings: low variation, mixed variation, and high variation. In each setting, we selected three random scenarios, with each scenario having a different number of flows. In all settings, the mean flow rates were set to 200 KBps.

In the low variation setting, the CoV of all flows is set to 0.2, indicating that all flows have low variability. Figures 4.6(a) and 4.6(b) show the comparison between the scenarios described in Table 4.3. In all scenarios, ILP and ISOCP are less conservative than APX and sample more flows fully compared to APX.

In the mixed variation setting, the CoV of all flows is selected randomly from the set $[0.2, 0.5, 2]$, indicating that flows can have low, medium, or high variability. Figures 4.6(c) and 4.6(d) show the comparison between the scenarios described in Table 4.4. In scenario 1, ILP and ISOCP admit the same number of flows as APX; however, in the other scenarios, they are less conservative than APX. In scenario 1, APX fully samples fewer flows compared to the other algorithms, but in scenario 2, it fully samples more. In scenario 3, APX falls between ILP and ISOCP.

In a high variation scenario, the CoV of all flows is set to 2, indicating high variability. Figures 4.6(e) and 4.6(f) show the comparison between the scenarios described in Table 4.5. In terms of fully sampled flows, APX samples more flows than all other algorithms in these scenarios, while being more conservative than ISOCP and ILP (except in scenario 1, where it admits the same number of flows).

In all of these scenarios, ISOCP admits the same number of flows as ILP. However, when considering fully sampled flows, there are some differences: sometimes ISOCP fully samples more flows, and other times fewer. This indicates that while the objective values of both ISOCP and ILP are the same in these scenarios, their sampling allocations may differ.

Conclusion: In general, both ILP and ISOCP are suitable for small-scale scenarios. However, ISOCP has difficulty solving many of these scenarios within a reasonable time frame compared to ILP.

In cases where all algorithms can solve the sampling problems within a reasonable time,

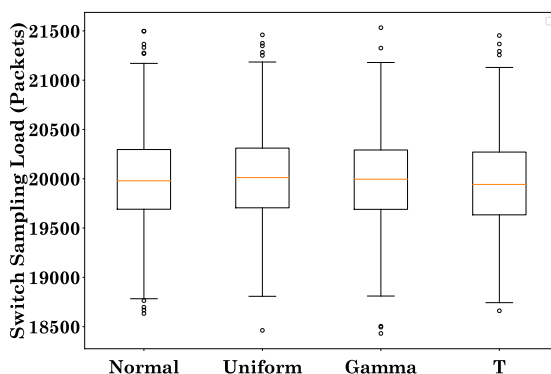


Figure 4.7: Effect of rate distribution on APX.

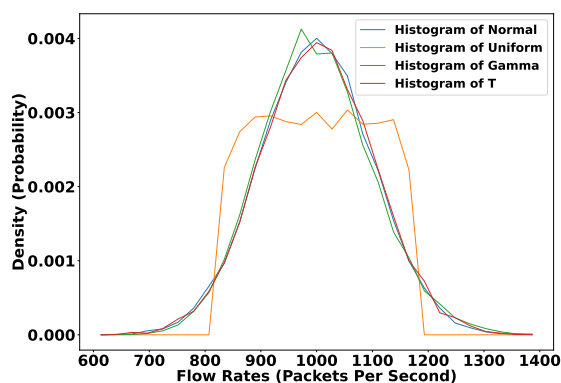


Figure 4.8: Histogram of different distributions.

ILP and ISOCP show similar performance and can outperform APX in low-variability scenarios. However, in high-variability scenarios, APX outperforms these algorithms. In mixed-variability scenarios, there is one case where APX outperforms both (scenario 2).

4.2.3 APX Load Distribution Analysis

APX does not optimally distribute the sampling load across switches. This issue arises due to the objective function, which aims to maximize the number of flows being sampled. As long as the capacity constraints are not violated, any feasible solution is considered acceptable. For example, consider the Abilene topology 4.2 with four flows passing from switch one to switch two. Suppose all flows are set to 200 KBps, with a CoV of 0.1, and both switches have

a high sampling capacity of 1000 KBps. In this scenario, APX may decide to sample all flows on switch one. Even though this results in all the load being concentrated on switch one, the algorithm has high confidence in sampling all flows and might not consider distributing some of the load to switch two. As previously discussed, this behavior can result in a TCAM space overload.

4.2.4 APX Sensitivity to Rate Distribution

In the derivation of APX, we applied the tail probability of the Normal distribution. A natural question is how sensitive APX is to this assumption. To answer this question, we focus on a single switch, as we applied the Normal distribution to compute sampling capacity violation of each switch. We setup 20 flows through the switch, each with an average rate of 1000 pps and a variance of 10,000 pps², translating to a CoV of 0.1. We generate the actual rate of each flow in the simulations using Normal, Gamma, Uniform, and T distribution. The goal is to sample all flows going through the switch, indicated by $x_f = 1$ and $\alpha = 1$ at a violation probability of 5%. Utilizing APX, it was calculated that the switch requires a minimum capacity of 20,735.6 pps. In the simulations, we set the sampling capacity of the switch to a very large value, and then measure the actual sampling load on the switch under each traffic distribution. The results are presented in Fig. 4.7. We can see that the median switch load is close to 20,000 packets, with an inter-quartile range between 19500 and 20500 packets, regardless of the traffic distribution. Specifically, we found that in only about 5% of all the simulation runs for each distribution, the switch's capacity was violated, which was remarkably close to the desired 5% violation probability. The reason for this is that the violation probability is computed over the sum of flow rates for each switch. Based on the central limit theorem, the sum approaches Normal distribution as the number of flows becomes large.

Fig. 4.8 presents the corresponding histogram with 30 bins, where the bin centers are connected by lines. In particular, 10,000 values were generated for each distribution, and

the data is displayed using 30 bins in the plot. This plot shows how the generated numbers closely follow their respective distributions, with a mean of 1000 pps and variance of 10000 pps². To generate values for each distribution with a mean of 1000 pps and a variance of 10,000 pps², we appropriately set the parameters for each distribution as follows:

Normal Distribution: For a Normal distribution, the mean μ and variance σ^2 are directly related to the distribution's parameters.

Uniform Distribution: For a Uniform distribution, the mean μ and variance σ^2 are given by $\mu = \frac{1}{2}(a + b)$ and $\sigma^2 = \frac{1}{12}(b - a)^2$. To generate values for a Uniform distribution with the target mean and variance, the parameters can be set as follows:

$$a = 1000 - 50 \cdot \sqrt{12}, \quad b = 1000 + 50 \cdot \sqrt{12}$$

With these values of a and b , the Uniform distribution will have a mean of 1000 pps and a variance of 10000 pps².

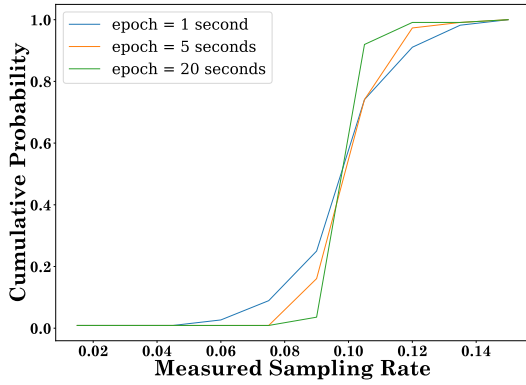
Gamma Distribution: For a Gamma distribution, the parameters are shape parameter k and scale parameter θ . Given the target mean $\mu = 1000$ pps² and variance $\sigma^2 = 10000$ pps², the parameters can be set as follows:

$$\theta = 10, \quad k = \frac{\mu}{\theta} = \frac{1000}{10} = 100$$

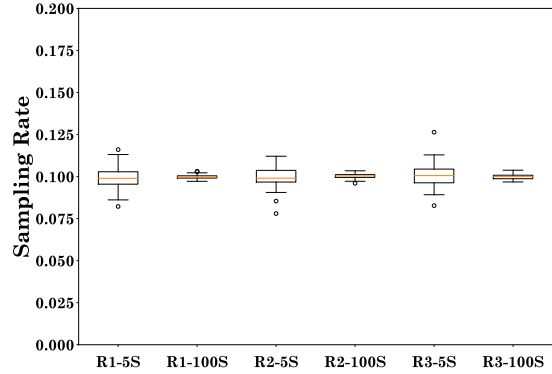
t-Distribution: To generate values for a t-distribution with a specified mean and variance, we will start with a large degree of freedom, say $\nu = 100000$, which approximates a Normal distribution with mean $\mu = 0$ and variance $\sigma^2 = 1$. Then we will shift and scale the t-distribution to achieve the desired mean and variance. The transformation is given by:

$$X = \left(\frac{1000}{\sqrt{10000}} + t_\nu \right) \times \sqrt{10000}$$

Here, t_ν is the value from the t-distribution.



(a) Effect of epoch length on APX (CDF plot).



(b) Effect of epoch length on APX (Box plot).

Figure 4.9: Simulations for testing different epoch lengths on APX.

4.2.5 Effect of Epoch Length

To assess the effect of epoch length on measured sampling rates, we used Abilene topology with flow rates uniformly set at 200 KBps and a CoV of 1. The capacity of all switches was configured to 100 pps. Out of 100 flows chosen randomly, APX admits 22 of these flows. The results are depicted in Fig. 4.9(a). We observe that by increasing the epoch length, the measured sampling rates converge to 0.1, which is the desired target rate. However, longer epochs have the downside of reduced responsiveness to new sampling requests that arrive mid-epoch. This presents a trade-off that can be set by network operators depending on their goals. It is important to note that in this scenario, none of the switch capacities were violated. This confirms that any observed low measured sampling rates are not a consequence of switch capacity limitations.

To further analyze the impact of increasing the epoch length, we generated a box plot illustrating the measured sampling rates for APX over 5-second and 100-second epoch lengths. Specifically, we collected the sampling rates across three different runs with epoch lengths set to 5 seconds (5S) and 100 seconds (100S) for each run (R). As illustrated in Fig. 4.9(b), the runs with a 5-second epoch length show a larger range between the minimum and maximum values compared to the 100-second runs. It can be inferred that as epoch lengths increase,

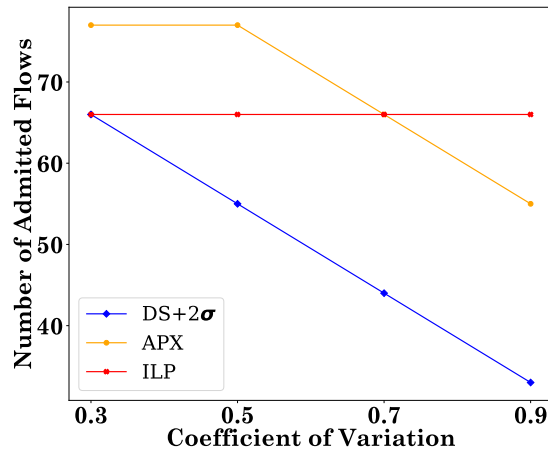


Figure 4.10: Effect of variability in flow rates.

the sampling rates will converge more closely to the target sampling rate. Although we did not provide the effect of epoch length on ILP and ISOCP, we expect to see convergence to the target sampling rate as the epoch length increases.

4.2.6 Effect of Rate Variability

In this experiment, we study the effect of rate variability by increasing the traffic CoV. Since the increase in CoV does not affect the DS algorithm and cSamp solutions, only the DS+2 σ , APX and ILP algorithms are shown in Fig. 4.10. For ISOCP, the number of admitted flows is the same as for ILP, which is not shown in the figure.

We selected 100 flows for sampling and set all flow rates to 200 KBps. All switch sampling capacities were set to 400 KBps. Without changing the mean rate of the flows, we increased the CoV for all flows and then ran the algorithms. As CoV increases, rate uncertainty also increases which results in a reduced number of admitted flows.

In case of ILP, there are two potential scenarios that could explain why the number of admitted flows remains unchanged. The first one is that ILP is not optimal since we use MIPGap parameter to stop the Gurobi sooner than the optimal solution (MIPGap=10%). The other possibility is that sometimes increasing the CoV does affect the final solution in

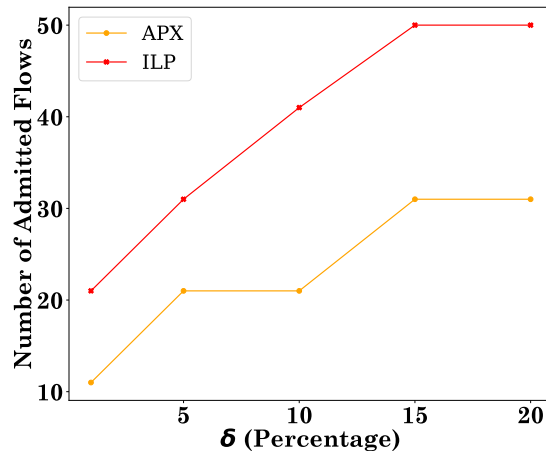


Figure 4.11: Effect of capacity violation probability.

the formulation and that might be the case for ILP.

4.2.7 Effect of Capacity Violation Probability (δ)

In this experiment, we study the effect of increasing the switch capacity violation probability (δ). we selected 50 flows for sampling. The flows' sources and destinations were randomly chosen. We set all flow mean rates to 200 KBps and CoVs to 2. All switch capacities were set to 200 KBps.

As seen in Fig. 4.11, ILP admits more flows compared to APX for different values of δ . This is expected since APX provides a looser bound. ISOCP is not shown in the figure; however, it provides the same number of admitted flows as ILP. As δ increases, all of these algorithms become less strict about admitting flows, which is expected.

4.2.8 Model-Driven Simulations

Flow Settings. Since there is only one path between each source IP and destination IP and only one host is connected to each switch, our simulation with the Abilene topology will have 110 OD pairs. However, to increase the number of OD pairs for measurement, we assume that a new IP address is assigned to the hosts at the start of each epoch. This

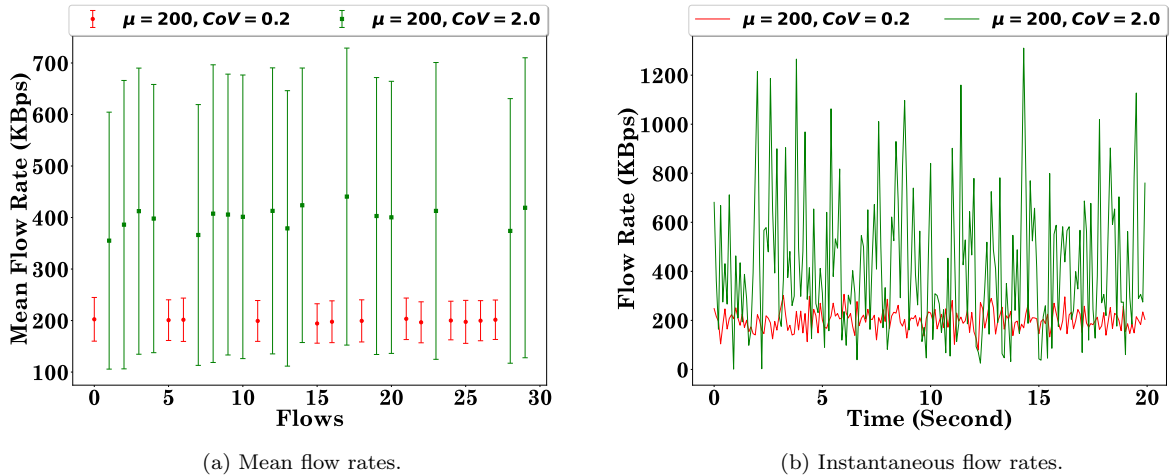


Figure 4.12: Properties of model-driven traffic.

results in 550 flows over 5 epochs available for sampling. Of these, 446 flows (about 80%) were selected across all 5 epochs. We assign flow rates by randomly selecting their mean from the set $\{200, 300, 500\}$ KBps. Additionally, for each flow, with the probability of 30% and 70%, its CoV is set to 0.2 (low variability) or 2 (high variability), respectively.

Fig. 4.12 illustrates some basic statistical properties of the flows that were generated during the model-driven simulations. Specifically, Fig. 4.12(a) shows the mean and standard deviation of the flow rates for a random selection of the flows, each with a mean rate of 200 KBps and varying CoVs. Fig. 4.12(b) presents the instantaneous flow rates of two distinct flows, both with the same mean rate of 200 KBps but different CoVs. It is noteworthy that in our model-driven experiments, we do not use the traffic estimator and assume that the CoV and mean of flows for the subsequent epoch are known.

Capacity Settings. The switch sampling capacities are set to 400 pps.

Admitted Flows Analysis. The results for Admitted Flows are depicted in Fig. 4.13. Specifically, Fig. 4.13(a) shows number of admitted flows for each algorithm (from 446 flows). As expected, DS is the most aggressive algorithm due to its lack of consideration for flow rate fluctuations. We also see that cSamp becomes stricter as the value of ϵ increases.

Fully Sampled Flows Analysis. Fig. 4.13(b) presents the results of Fully Sampled Flows

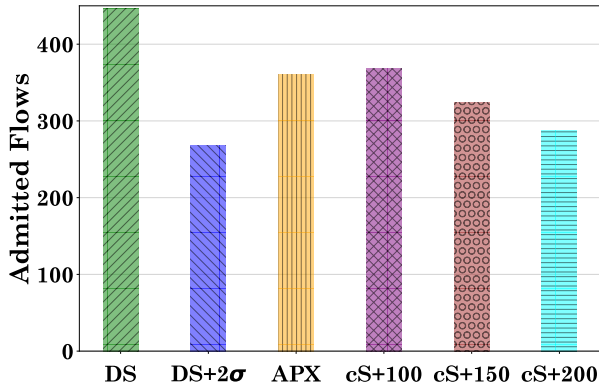
across different algorithms. We see that while DS was the most aggressive in admitting flows, it achieves the lowest number of fully sampled flows among all the algorithms. This outcome aligns with expectations, given that APX takes into account the variation in flow rates, while DS relies solely on the mean rate. As a result, APX adopts a more cautious approach in admitting flows.

By dividing the number of fully sampled flows by the total number of flows being measured, we obtain the percentage of fully sampled flows. In the case of the DS and DS+2 σ algorithms, it was determined that 8.71% and 30.92% percent of flows are fully sampled respectively. For APX, this percentage is significantly higher, reaching 40.94%. Moreover, it shows an improvement of 6.45% over the top-performing cSamp baseline (cS+150), which achieved a rate of 34.49%. Therefore dSamp can increase the percentage of fully sampled flows compared to other algorithms that consider fluctuations by up to 10.02%.

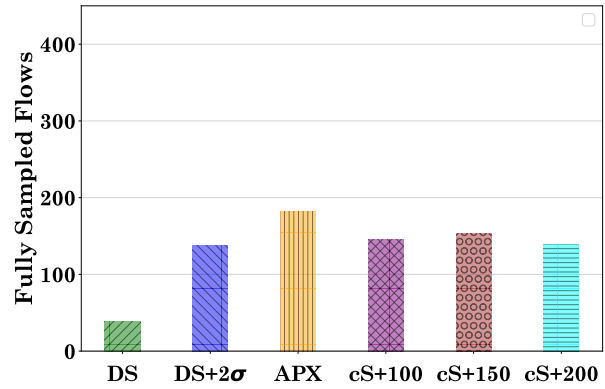
For cSamp, we had to decide about the parameter ϵ . We experimented with a range of values, but have only plotted cS+100, cS+100, and cS+200, as we found that other values make the algorithm too aggressive or conservative leading to lower performance.

Sampling Rate Analysis. Fig. 4.13(c) shows that for DS, the sampling rates typically range between approximately 0.04 and 0.012. It can be seen that except DS and cS+100, all other algorithms can actually achieve sampling rates that are very close to the target sampling rate. In fact, while there are some variations in measured sampling rates for these algorithms, the median sampling rates closely match the 0.10 target rate. This can also be inferred from Fig. 4.13(d), where the curved lines for DS and cS+100 are positioned to the left of those for other algorithms.

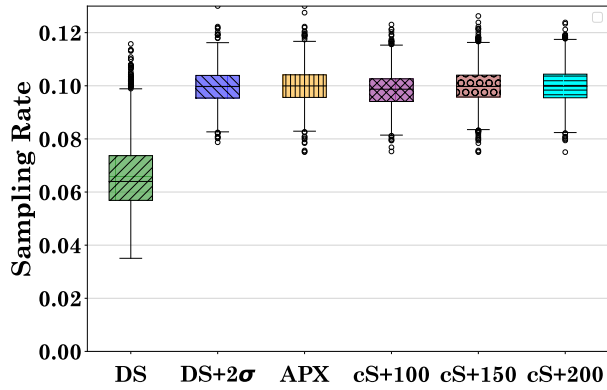
Conclusion. In our model-driven scenario, we observe that APX can fully sample more flows than even the best version of cSamp, namely cS+150. It also outperforms DS+2 σ , which performs close to cS+150.



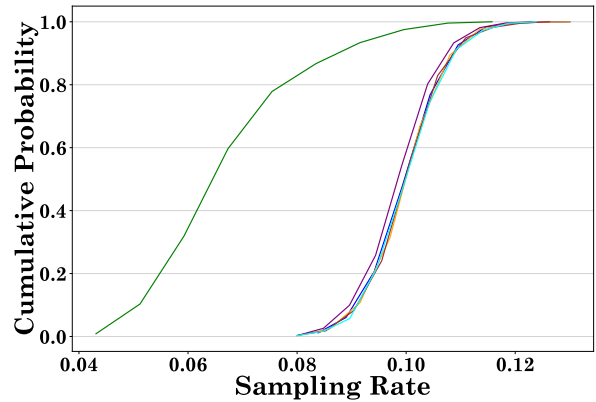
(a) Total number of admitted flows.



(b) Total number of fully sampled flows.



(c) Measured sampling rates (Box plot).



(d) Measured sampling rates (CDF plot).

Figure 4.13: Model-driven ns-3 simulations results.

4.2.9 Trace-Driven Simulations

In this experiment, for comparison between APX and cSamp, we selected cS+150, as it was the top-performing cSamp version in the model-driven simulations.

Flow Settings. In these simulations, we utilized the actual traffic data obtained from an ISP backbone link [5] to generate rates for various flows. Each traffic trace consists of detailed packet-level information, capturing traffic over the ISP link for a duration of 15 minutes on a specific day. We used 5-second samples to simulate each flow during an epoch. For each flow in the first epoch, a 25-second sample from these traces was used to estimate the flow’s mean and variance. This implies that five epochs existed prior to the first epoch, and we

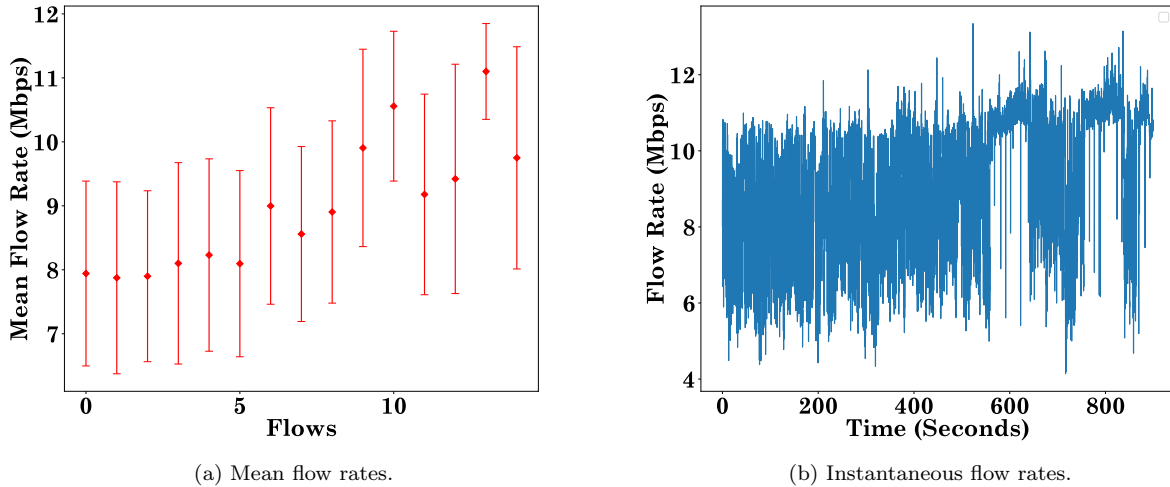


Figure 4.14: Properties of trace-driven traffic.

leveraged that information for estimation. The traffic rate for each flow was changed every 100 milliseconds throughout the simulation, as in the model-driven simulations. Since using the original traffic rates could significantly increase the simulation time in ns-3, all traffic rates from the original files were scaled down by a factor of 100. This scaling down does not alter the CoV in comparison to the original data.

Fig. 4.14 presents the fundamental characteristics of the flow rates obtained from these scaled-down traces. The trace is collected on January 1st, 2023, between 14:00 and 14:15 PM from sample point-F, with rate fluctuations plotted in 100 ms intervals. It is noteworthy that the link capacity is 1 Gbps.

As mentioned in model-driven scenario, our simulation with the Abilene topology has 110 OD pairs. However, to increase the number of OD pairs for measurement, we assume that a new IP address is assigned to the hosts at the start of each epoch. This results in 550 flows over 5 epochs available for sampling. Of these, 443 flows (about 80%) were selected across all 5 epochs.

To further clarify our measurement, we provide an example of simulating traffic between host 1 and host 3 over five epochs, as depicted in Fig. 4.15. Consider the first epoch, where flow 0 travels from switch 1 to switch 3, with the source IP being 10.0.1.1 and the destination

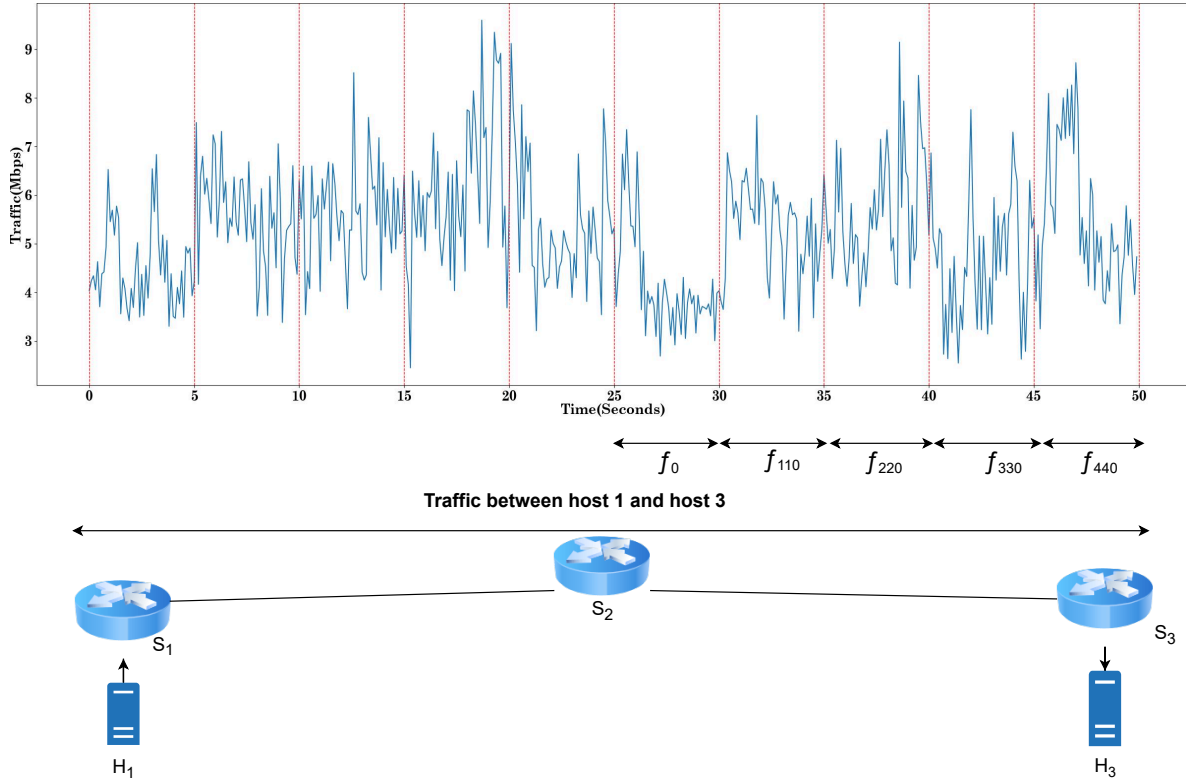


Figure 4.15: Simulating traffic between host 1 and host 3 over five epochs using a 25-second sample from the ISP trace [5].

IP being 10.0.3.1. This flow utilizes 25 seconds of a 50-second ISP trace sample to estimate the mean and variance for the first epoch. In the second epoch, although the routing path remains unchanged, the source and destination IPs will change. Consequently, the flow is treated as a new flow, referred to as flow 110. The mean and variance of flow 110 can be estimated based on the first 30 seconds of the 50-second ISP trace sample. Finally, the fifth epoch, which simulates the traffic for flow 440, uses the previous 45 seconds to estimate its mean and variance. This approach allows us to simulate five times more flows for the sake of presentation while effectively leveraging the mean and variance estimator.

Capacity Settings. The sampling capacity of each switch was set to 200 pps.

Admitted Flows Analysis. Fig. 4.16(a) shows Admitted Flows for each algorithm (from 443 flows). Similar to the model-driven approach, DS admits more flows compared to other

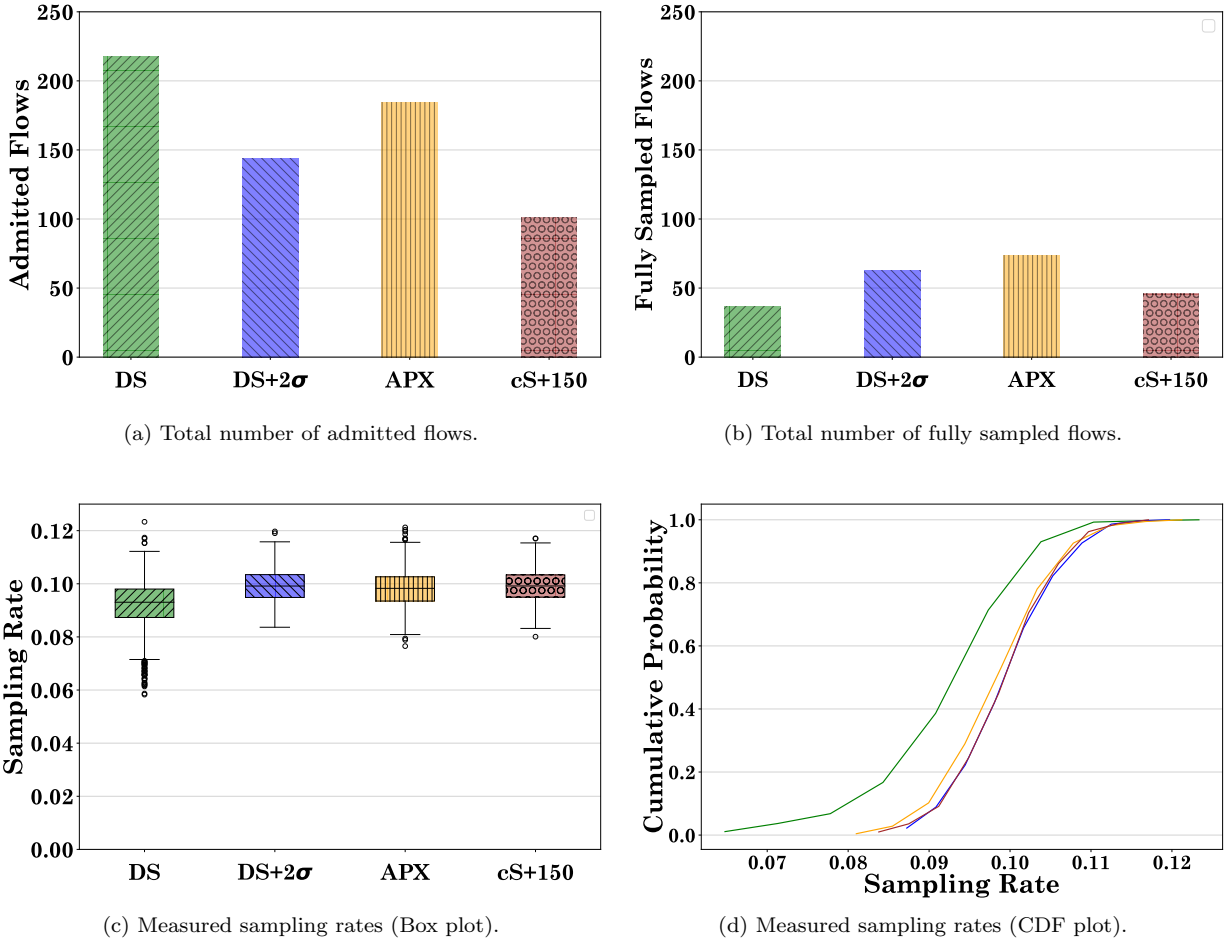


Figure 4.16: Trace-driven ns-3 simulation results.

algorithms due to its lack of consideration for flow rate fluctuations.

Fully Sampled Flows Analysis. The results are shown in Fig. 4.16(b) for Fully Sampled Flows. While the relative performance of the DS, DS+2 σ , and APX algorithms is similar to what was observed in model-driven simulations, cS+150 performs poorly even compared to DS+2 σ . The reason is that cS+150 considers the worst-case rate fluctuations, which makes it very conservative under the ISP trace, which is smoother compared to model-driven traces.

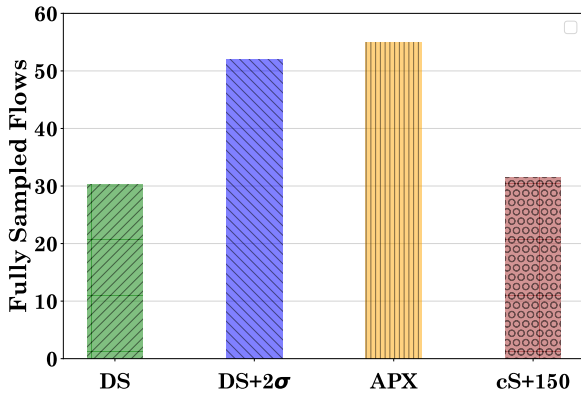
Although APX outperformed all other solutions and fully sampled 16.54% of measured flows, its performance was very close to DS+2 σ , which fully sampled 14.08% of the measured flows. In the case of DS and cS+150, the percentages were 8.25% and 10.31%, respectively. In this scenario, dSamp can increase the fraction of fully sampled flows by up to 6.23%

compared to other solutions, considering fluctuations.

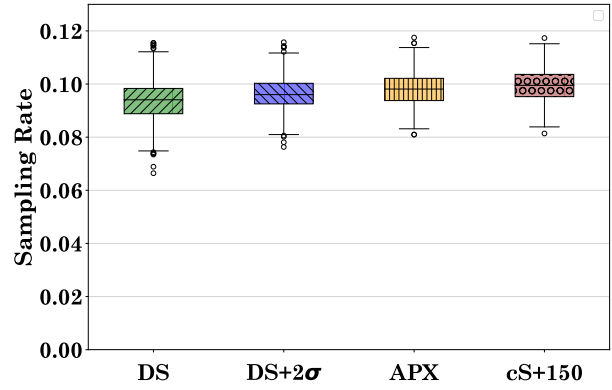
Sampling Rate Analysis. Due to the lower variability in trace-driven flow rates compared to those in model-driven simulations, the Sampling Rate of the DS algorithms tend to be closer to the target sampling rate. This can be inferred by comparing the ranges in Fig. 4.16(c) and Fig. 4.13(c). This can also be inferred by comparing Fig. 4.16(d) and Fig. 4.13(d).

Impact of Statistics Polling Delay on Performance. In the previous simulations, we assumed there was no delay between the time sampling allocations were installed on switches and the start of the flows. However, in a real deployment, the sampling allocations would be installed with a delay. This delay includes the time required to send the polling message to the switches and receive their responses after the optimizer has solved the sampling allocation problem. The delay depends on the solution time of the problem, the size of the polling message and response, and the link speed connecting the controller to the switches. This delay can range from hundreds of milliseconds to seconds [58].

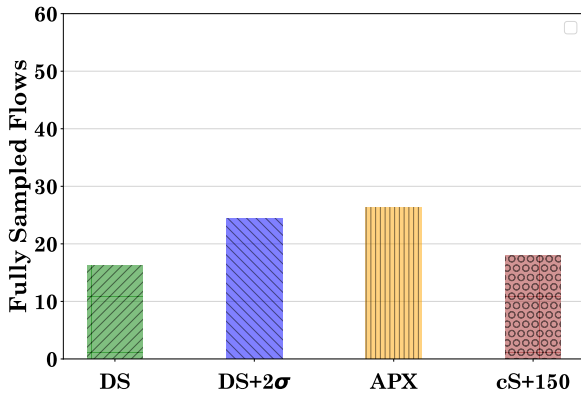
To evaluate the impact of this delay, we compared a scenario with no delay to scenarios with a 200-millisecond delay and a 500-millisecond delay. Figures 4.17(b), 4.17(d), and 4.17(f) show their sampling rates respectively. Similar to the trace-driven scenario, we used samples of 5 seconds from ISP trace to simulate each flow, however the switch sampling capacities were all set to 100 KBps instead of 200 KBps. As seen from 4.17(c) and 4.17(e), the percentage of fully sampled flows decreases as the delay increases for each algorithm. It can be inferred that the trends remain consistent, meaning that APX still results in more fully sampled flows compared to all other algorithms. However, the percentage of fully sampled flows for APX drops from 12.33% in the scenario with no delay to 1.34% in the 500-millisecond delay scenario.



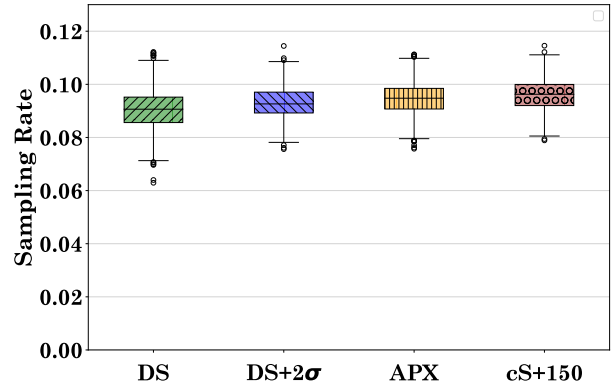
(a) Total number of fully sampled flows (No delay).



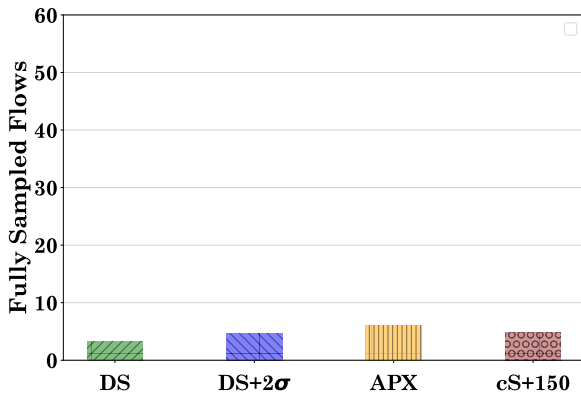
(b) Measured sampling rates (No delay).



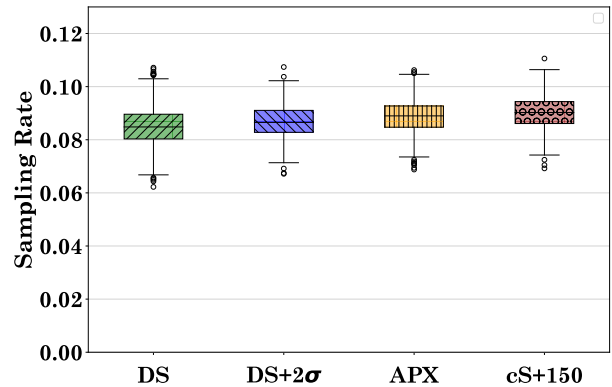
(c) Total number of fully sampled flows (200 ms delay).



(d) Measured sampling rates (200 ms delay).



(e) Total number of fully sampled flows (500 ms delay).



(f) Measured sampling rates (500 ms delay).

Figure 4.17: Trace-driven ns-3 simulation results under different delay scenarios.

Conclusion. We used traces from a backbone ISP to simulate flows and conducted a trace-driven simulation with these data. Our results indicate that APX can outperform other algorithms that consider flow rate fluctuations, such as DS+2 σ and cSamp. These algorithms face significant challenges in networks with dynamic traffic. Specifically, the parameters of these algorithms are tuned once at deployment time based on the expected traffic characteristics. However, as traffic characteristics inevitably change over time, their performance becomes sub-optimal. In contrast, APX has no parameters that need to be tuned for specific types of traffic. Instead, it is designed to adapt to traffic changes, leading to higher performance than the other algorithms.

Furthermore, we observed the impact of adding delay to account for pulling statistics from switches at the beginning of each epoch. While the trends remain consistent and APX continues to outperform other algorithms in scenarios with delay, the delay does cause a reduction in the percentage of flows sampled.

Chapter 5

Conclusion and Future Works

5.1 Conclusion

In this thesis, we presented the design and evaluation of a sampling system called **dSamp**, capable of handling dynamic flow rates. We formulated the problem of network-wide coordinated sampling with dynamic flow rates as an ISOCP, which proved infeasible for even small-scale networks. To address the computational complexity of larger networks, we created two alternative reformulations: one transformation and one approximate. By evaluating their runtime in various scenarios, we demonstrated that the transformation works well for small-scale networks, while the approximate reformulation is more effective for larger-scale networks. Through both model-driven and trace-driven ns-3 simulations, we showed that our proposed system outperforms existing sampling systems in achieving a higher percentage of fully sampled flows.

5.2 Thesis Summary

This thesis can be summarised as follows:

- In *Chapter 1*, we introduced the importance of network monitoring and the limitations

of existing sampling solutions in handling dynamic flow rates. The motivation is to address these challenges in SDNs with a *proactive* solution that accommodates the dynamic nature of flow rates while remaining scalable and maintaining low overhead.

- In *Chapter 2*, we covered the essential mathematical concepts used in the thesis, including optimization. Furthermore, we detailed the background information on network monitoring, focusing on traffic sampling solutions.
- In *Chapter 3*, we presented **dSamp**, a coordinated network-wide traffic sampling system that proposes a high flow visibility. The key idea behind **dSamp** was to incorporate the statistical information of flow rates such as mean and variance in the sampling decision process. We presented the formulation of coordinated flow sampling with dynamic flow rates as an optimization problem, discussed the challenges of solving it, and introduced an approximate solution for large-scale networks.
- In *Chapter 4*, we presented extensive ns-3 simulations to study **dSamp**'s micro and macro behavior. We began by comparing **dSamp**'s algorithms across various scenarios. Next, we presented the results of model-driven and trace-driven simulations, conducted in ns-3, to compare **dSamp** with existing sampling solutions. Our evaluations demonstrated the effectiveness of **dSamp** across a range of scenarios. Specifically, we showed that **dSamp** can outperform other solutions by up to 6.37% in an ISP environment. Additionally, we analyzed the impact of introducing a delay at the start of each epoch.

5.3 Future Works

There are several areas in which this research work can be extended. The following can be considered for the future:

1. **Designing an online algorithm to solve the problem, where sampling requests are processed instantaneously instead of being batched over sam-**

pling epochs. While our current algorithm effectively handles dynamic flow rates in SDNs, it presents a drawback for network operators. Due to its epoch-based setting, the algorithm may show reduced responsiveness to new sampling requests that arrive mid-epoch. As a result, network operators might need to fine-tune the epoch length to meet the minimum requirements of network monitoring applications such as anomaly detection, which introduces a time cost for the operator. In contrast, an online solution does not suffer from reduced responsiveness, as it processes requests immediately upon their arrival, removing the need for additional time spent fine-tuning the epoch length.

2. **Exploring other methods for a better algorithm performance.** We initially applied the central limit theorem to a chance-constrained equation in order to formulate flow sampling with statistical flow rate information as an ISOCP. However, we have not yet explored robust optimization methods or concentration bounds. Investigating alternative mathematical approaches for formulating the flow sampling problem could potentially enhance its overall effectiveness. In general, our formulation does not equally distribute flows among switches, leading to a scenario where one switch might sample too many flows while others remain unused. This can result in overloading the TCAM. Therefore, one potential solution could be to incorporate TCAM space constraints into the formulation.

3. **Enhancing the flexibility of our proposed monitoring system by incorporating programmable switches such as P4.** Using high-level languages like P4, network operators can now program switch ASICs to implement protocols and functionalities directly within the network infrastructure. This ability offers the potential to extend our monitoring systems to leverage programmable switches, significantly reducing the monitoring load by selectively capturing only the necessary portions of traffic. For instance, this approach is particularly advantageous for applications that require only the packet header, such as DDoS detection, or specific parts of the packet

payload, like traffic classification.

4. **Handling multi path per OD pair and adding NAT support.** In our basic assumptions about the sampling system, we presumed that each OD pair uses only one specific path. However, works like [50] considered scenarios where each OD pair might have multiple paths. They leveraged Equal Cost Multi-Path routing (ECMP) to enable multiple paths for each OD pair and demonstrated how their formulation can handle this. Additionally, by incorporating an OD pair identifier and mapping table, they were able to manage conditions where NAT is present. In contrast, our system identifies each flow solely by its source IP and destination IP, which prevents us from supporting NAT.
5. **Deploying algorithm in real environment.** Due to the high cost of network infrastructure, such as OpenFlow switches, we were unable to implement our solution in a real testbed. However, it is important to assess how our system would perform in real-world scenarios. One key consideration is the delay at the start of each epoch, which accounts for both the optimizer’s solution time and the required polling that would occur in real deployments. In our evaluation, we addressed this by simulating the condition with an added delay, demonstrating that our system meets expectations. As the delay increases, a smaller percentage of flows are fully sampled. Regardless of the delay, our system still outperforms other algorithms in terms of overall performance.

Bibliography

- [1] Cisco Inc., “Cisco Catalyst 6500 Series Switches,”. accessed Jul. 29, 2024. [Online]. Available: <https://www.cisco.com/c/en/us/products/switches/catalyst-6500-series-switches/index.html>.
- [2] Gigamon Inc., “Understanding network tap devices,”. accessed Oct. 31, 2022. [Online]. Available: <https://www.gigamon.com/products/access-traffic/network-taps.html>.
- [3] “Gurobi Optimization”. accessed Aug. 06, 2024. [Online]. Available: <http://www.gurobi.com/>.
- [4] Juniper Networks, “Understanding port mirroring and analyzers”. accessed Aug. 13, 2024. [Online]. Available: <https://www.juniper.net/documentation/us/en/software/junos/network-mgmt/topics/topic-map/port-mirroring-and-analyzers.html>.
- [5] MAWI, “MAWI Working Group Traffic Archive”. accessed Aug. 30, 2024. [Online]. Available: <http://mawi.wide.ad.jp/mawi/>.
- [6] “ns-3: Discrete-event network simulator for internet systems”. accessed Aug. 29, 2024. [Online]. Available: <https://www.nsnam.org/about/what-is-ns-3/>.

- [7] “OpenFlow Switch Specification Version 1.5.1”. accessed Aug. 13, 2024. [Online]. Available: <https://opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf>.
- [8] sFlow, “Making the network visible,”. accessed Mar. 18, 2024. [Online]. Available: <http://sflow.org>.
- [9] The university of Adelaide, “The internet topology Zoo”. accessed Aug. 06, 2024. [Online]. Available: <http://www.topology-zoo.org/index.html>.
- [10] Davide Andreoletti, Sebastian Troia, Francesco Musumeci, Silvia Giordano, Guido Maier, and Massimo Tornatore. “Network traffic prediction based on diffusion convolutional recurrent neural networks”. In *Proc. IEEE INFOCOM, Workshop on Network Intelligence*, 2019.
- [11] Mostafa Bastam, Masoud Sabaei, and Ruhollah Yousefpour. “A scalable traffic engineering technique in an SDN-based data center network”. *Transactions on Emerging Telecommunications Technologies*, vol. 29, no. 2, 2018.
- [12] Aharon Ben-Tal, Laurent El Ghaoui, and Arkadi Nemirovski. “*Robust optimization*”. Princeton university press, 2009.
- [13] Marco Canini, Damien Fay, David J Miller, Andrew W Moore, and Raffaele Bolla. “Per flow packet sampling for high-speed network monitoring”. In *Proc. IEEE COMSNETS*, 2009.
- [14] Gion Reto Cantieni, Gianluca Iannaccone, Chadi Barakat, Christophe Diot, and Patrick Thiran. “Reformulating the monitor placement problem: Optimal network-wide sampling”. In *Proc. ACM CoNEXT*, 2006.

- [15] Chia-Wei Chang, Guanyao Huang, Bill Lin, and Chen-Nee Chuah. “Leisure: A framework for load-balanced network-wide traffic measurement”. In *Proc. ACM/IEEE ANCS*, 2011.
- [16] Huan Chen, Lemin Li, Jing Ren, Yang Wang, Yangming Zhao, Xiong Wang, Sheng Wang, and Shizhong Xu. “A scheme to optimize flow routing and polling switch selection of software defined networks”. *PloS one*, vol. 10, no. 12, 2015.
- [17] Wenqing Chen, Melvyn Sim, Jie Sun, and Chung-Piaw Teo. “From CVaR to uncertainty set: Implications in joint chance-constrained optimization”. *Operations research*, vol. 58, no. 2, 2010.
- [18] Herman Chernoff. “A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations”. *The Annals of Mathematical Statistics*, 1952.
- [19] Shihabur Rahman Chowdhury, Md Faizul Bari, Reaz Ahmed, and Raouf Boutaba. “Payless: A low cost network monitoring framework for software defined networks”. In *Proc. IEEE NOMS*, 2014.
- [20] Benoit Claise. “Cisco systems NetFlow services export version 9”. Technical report, 2004.
- [21] Reuven Cohen and Evgeny Moroshko. “Sampling-on-demand in SDN”. *IEEE/ACM Trans. Netw.*, vol. 26, no. 6, 2018.
- [22] Aboagela Dogman, Reza Saatchi, and Samir Al-Khayatt. “An adaptive statistical sampling technique for computer network traffic”. In *Proc. IEEE CSNDSP*, 2010.
- [23] Yang Du, He Huang, Yu-E Sun, Shigang Chen, Guoju Gao, and Xiaocan Wu. “Self-adaptive sampling based per-flow traffic measurement”. *IEEE/ACM Trans. Netw.*, vol. 31, no. 3, 2022.

- [24] Nick Duffield, Carsten Lund, and Mikkel Thorup. “Charging from sampled network usage”. In *Proc. ACM SIGCOMM, Workshop on Internet Measurement*, 2001.
- [25] Alessandro D’Alconzo, Idilio Drago, Andrea Morichetta, Marco Mellia, and Pedro Casas. “A survey on big data for network traffic monitoring and analysis”. *IEEE Trans. Netw. Service Manag.*, vol. 16, no. 3, 2019.
- [26] Ahmed Elmokashfi, Amund Kvalbein, and Constantine Dovrolis. “On the scalability of BGP: The role of topology growth”. *IEEE Journal on Selected Areas in Communications*, vol. 28, no. 8, 2010.
- [27] Leslie R Foulds. “*Optimization techniques: an introduction*”. Springer Science & Business Media, 2012.
- [28] Taejin Ha, Sunghwan Kim, Namwon An, Jargalsaikhan Narantuya, Chiwook Jeong, JongWon Kim, and Hyuk Lim. “Suspicious traffic sampling for intrusion detection in software-defined networks”. *Computer Networks*, vol. 109, 2016.
- [29] Nicolas Hohn and Darryl Veitch. “Inverting sampled traffic”. In *Proc. ACM SIGCOMM*, 2003.
- [30] Martin Holkovič, Ondřej Ryšavý, and Jindřich Dudek. “Automating network security analysis at packet-level by using rule-based engine”. In *Proc. ECBS*, 2019.
- [31] Huawei. “OpenFlow”, 2024. accessed Aug. 13, [Online]. Available: <https://info.support.huawei.com/info-finder/encyclopedia/en/OpenFlow.html>.
- [32] Jared Ivey, Hemin Yang, Chuanji Zhang, and George Riley. “Comparing a scalable SDN simulation framework built on ns-3 and DCE with existing SDN simulators and emulators”. In *Proc. ACM SIGSIM*, 2016.

- [33] Diego Kreutz, Fernando MV Ramos, Paulo Esteves Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. “Software-defined networking: A comprehensive survey”. *Proc. IEEE*, vol. 103, no. 1, 2014.
- [34] Lucien Le Cam. “The central limit theorem around 1935”. *Statistical science*, 1986.
- [35] Tao Li, Shigang Chen, Wen Luo, and Ming Zhang. “Scan detection in high-speed networks based on optimal dynamic bit sharing”. In *Proc. IEEE INFOCOM*, 2011.
- [36] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. “FlowRadar: A better NetFlow for data centers”. In *Proc. USENIX NSDI*, 2016.
- [37] Chang Liu, AMehdi Malboubi, and Chen-Nee Chuah. “OpenMeasure: Adaptive flow measurement & inference with online learning in SDN”. In *Proc. IEEE INFOCOM*, 2016.
- [38] Miguel Sousa Lobo, Lieven Vandenberghe, Stephen Boyd, and Hervé Lebret. “Applications of second-order cone programming”. *Linear algebra and its applications*, vol. 284, no. 1-3, 1998.
- [39] Mehdi Malboubi, Liyuan Wang, Chen-Nee Chuah, and Puneet Sharma. “Intelligent SDN based traffic (de) aggregation and measurement paradigm (iSTAMP)”. In *Proc. IEEE INFOCOM*, 2014.
- [40] Keith McCloghrie and Marshall T Rose. “Management Information Base for network management of TCP/IP-based internets”. Technical report, 1990.
- [41] Luiza Nacshon, Rami Puzis, and Polina Zilberman. “Floware: Balanced flow monitoring in software defined networks”. *arXiv preprint arXiv:1608.03307*, 2016.
- [42] Arkadi Nemirovski and Alexander Shapiro. “Scenario approximations of chance constraints”. *Probabilistic and randomized methods for design under uncertainty*, 2006.

- [43] Kokouvi Benoit Nougnanke, Marc Bruyere, and Yann Labit. “Low-overhead near-real-time flow statistics collection in SDN”. In *Proc. IEEE NetSoft*, 2020.
- [44] Jeff Rasley, Brent Stephens, Colin Dixon, Eric Rozner, Wes Felter, Kanak Agarwal, John Carter, and Rodrigo Fonseca. “Planck: Millisecond-scale monitoring and control for commodity networks”. *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, 2014.
- [45] Frederic Raspall. “Efficient packet sampling for accurate traffic measurements”. *Computer Networks*, vol. 56, no. 6, 2012.
- [46] Sogand Sadrhaghghi, Mahdi Dolati, Majid Ghaderi, and Ahmad Khonsari. “SoftTap: A software-defined TAP via switch-based traffic mirroring”. In *Proc. IEEE NetSoft*, 2021.
- [47] Sogand Sadrhaghghi, Mahdi Dolati, Majid Ghaderi, and Ahmad Khonsari. “FlowShark: Sampling for high flow visibility in SDNs”. In *Proc. IEEE INFOCOM*, 2022.
- [48] Sogand Sadrhaghghi, Mahdi Dolati, Majid Ghaderi, and Ahmad Khonsari. “Monitoring openflow virtual networks via coordinated switch-based traffic mirroring”. *IEEE Trans. Netw. Service Manag.*, vol. 19, no. 3, 2022.
- [49] Vyas Sekar, Anupam Gupta, Michael K Reiter, and Hui Zhang. “Coordinated sampling sans origin-destination identifiers: algorithms and analysis”. In *Proc. IEEE COM-SNETS*, 2010.
- [50] Vyas Sekar, Michael K Reiter, Walter Willinger, Hui Zhang, Ramana Rao Kompella, and David G Andersen. “cSamp: A system for network-wide flow monitoring”. *Proc. USENIX NSDI*, 2008.
- [51] Vyas Sekar, Michael K Reiter, and Hui Zhang. “Revisiting the case for a minimalist approach for network flow monitoring”. In *Proc. ACM SIGCOMM*, 2010.

- [52] Jang-Ping Sheu and Yen-Cheng Chuo. “Wildcard rules caching and cache replacement algorithms in software-defined networking”. *IEEE Trans. Netw. Service Manag.*, vol. 13, no. 1, 2016.
- [53] Jang-Ping Sheu, Woan-Tyng Lin, and Guey-Yun Chang. “Efficient TCAM rules distribution algorithms in software-defined networking”. *IEEE Trans. Netw. Service Manag.*, vol. 15, no. 2, 2018.
- [54] Sajad Shirali-Shahreza and Yashar Ganjali. “Flexam: Flexible sampling extension for monitoring and security applications in openflow”. In *Proc. ACM SIGCOMM, Workshop on Hot Topics in SDN*, 2013.
- [55] Gerard Sierksma and Yori Zwols. “*Linear and integer optimization: theory and practice*”. CRC Press, 2015.
- [56] Chakchai So-In. “A survey of network traffic monitoring and analysis tools”. *Cse 576m computer system analysis project, Washington University in St. Louis*, 2009.
- [57] Zhiyang Su, Ting Wang, Yu Xia, and Mounir Hamdi. “FlowCover: Low-cost flow monitoring scheme in software defined networks”. In *Proc. IEEE GLOBECOM*, 2014.
- [58] Zhiyang Su, Ting Wang, Yu Xia, and Mounir Hamdi. “CeMon: A cost-effective flow monitoring system in software defined networks”. *Computer Networks*, vol. 92, 2015.
- [59] Junho Suh, Ted Taekyoung Kwon, Colin Dixon, Wes Felter, and John Carter. “Open-sample: A low-latency, sampling-based measurement platform for commodity sdn”. In *Proc. IEEE ICDCS*, 2014.
- [60] Hamid Tahaei, Rosli Bin Salleh, Mohd Faizal Ab Razak, Kwangman Ko, and Nor Badrul Anuar. “Cost effective network flow measurement for software defined networks: A distributed controller scenario”. *IEEE Access*, vol. 6, 2018.

- [61] Jixing Tang, Yue Zhang, and Yan Li. “Adarate: A rate-adaptive traffic measurement method in software defined networks”. *International Journal of Performability Engineering*, vol. 13, no. 6, 2017.
- [62] Yang Tian, Weiwei Chen, and Chin-Tau Lea. “An SDN-based traffic matrix estimation framework”. *IEEE Trans. Netw. Service Manag.*, vol. 15, no. 4, 2018.
- [63] Pang-Wei Tsai, Chun-Wei Tsai, Chia-Wei Hsu, and Chu-Sing Yang. “Network monitoring in software-defined networking: A review”. *IEEE Systems Journal*, vol. 12, no. 4, 2018.
- [64] Liang-Min Wang, Tim Miskell, John Morgan, and Edwin Verplanke. “Design of A Multi-Path Reconfigurable Traffic Monitoring System”. In *Proc. IEEE NAS*, 2021.
- [65] Hongli Xu, Shigang Chen, Qianpiao Ma, and Liusheng Huang. “Lightweight flow distribution for collaborative traffic measurement in software defined networks”. In *Proc. IEEE INFOCOM*, 2019.
- [66] Hongli Xu, Zhuolong Yu, Chen Qian, Xiang-Yang Li, Zichun Liu, and Liusheng Huang. “Minimizing flow statistics collection cost using wildcard-based requests in SDNs”. *IEEE/ACM Trans. Netw.*, vol. 25, no. 6, 2017.
- [67] Curtis Yu, Cristian Lumezanu, Yueping Zhang, Vishal Singh, Guofei Jiang, and Harsha V Madhyastha. “Flowsense: Monitoring network utilization with zero measurement cost”. In *Proc. Springer PAM*, 2013.
- [68] Adel Zaalouk, Rahamatullah Khondoker, Ronald Marx, and Kpatcha Bayarou. “Orch-Sec: An orchestrator-based architecture for enhancing network-security using network monitoring and SDN control functions”. In *Proc. IEEE NOMS*, 2014.

- [69] Jun Zhang, Yang Xiang, Wanlei Zhou, and Yu Wang. “Unsupervised traffic classification using flow statistical properties and IP packet payload”. *Journal of Computer and System Sciences*, vol. 79, no. 5, 2013.
- [70] Ying Zhang. “An adaptive flow counting method for anomaly detection in SDN”. In *Proc. ACM CoNEXT*, 2013.
- [71] Bo Zhou, Dan He, and Zhili Sun. “Traffic predictability based on ARIMA/GARCH model”. In *Proc. IEEE NGI*, 2006.
- [72] Donghao Zhou, Zheng Yan, Yulong Fu, and Zhen Yao. “A survey on network data collection”. *Journal of Network and Computer Applications*, vol. 116, 2018.
- [73] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y Zhao, et al. “Packet-level telemetry in large datacenter networks”. In *Proc. ACM SIGCOMM*, 2015.