SFCMon: An Efficient and Scalable Monitoring System for Network Flows in SFC-enabled Domains

Michel Bonfim[1], **Kelvin Dias**[1], **Stênio Fernandes**[1]

[1]Centro de Informática (CIn) – Universidade Federal de Pernambuco (UFPE)
Recife – PE – Brazil

{msb6,kld,stenio}@cin.ufpe.br

Abstract. *A comprehensive monitoring system is essential to assist solutions for most of SFC problems. Therefore, in this work, we propose SFCMon, an efficient and scalable monitoring solution to keep track network flows in SFC environments. To achieve the desired goals, SFCMon works with a pipeline of probabilistic data structures to detect and store large flows as well as per-flow counters. For evaluation purposes, based on the SFC reference architecture defined by RFC 7665, we implement a Proof-of-Concept (PoC) framework, which provides a P4-based SFC switch and Python-based SFC Controller. Presented initial experiments demonstrate that SFCMon introduces a negligible performance penalty while providing significant scalability gains.*

1. Introduction

Network Functions Virtualization (NFV) promises to reduce the cost of deploying and operating large network infrastructures. Rather than dealing with proprietary hardware equipment, which includes a limited set of specific network applications, NFV allows users to transfer network functions (e.g., Firewall, *Network Address Translation* - NAT, and *Deep Packet Inspection* - DPI) from specific to common-off-the-shelf (COTS) hardware using virtualization technologies. NFV aims at creating and deploying end-to-end network services that interconnect one or more network functions [ETSI 2015].

Service Function Chaining (SFC) is a key enabler for NFV [Bhamare et al. 2016]. SFC is a mechanism that allows an ordered set of network Service Functions (SF), which may or may not be virtual, to be connected to each to form an end-to-end service through multiple data centers, WANs, and application service providers [Halpern and Pignataro 2015, Medhat et al. 2017]. SFCs deal with data traffic, control and monitoring of a specific application/service, applying appropriate policies along the routes according to the service requirements and considering the availability status of the network. The SFC standardization is addressed by the Internet Engineering Task Force (IETF) at the SFC working group (WG). Ongoing work at SFC WG focus on aspects of the architecture (RFC 7665 [Halpern and Pignataro 2015]) and protocol (RFC 8300 [Quinn et al. 2018]) needed to enable this network capability. SFC is a hot topic and encompasses various problems including SFC path selection, optimization of SF placement in an SFC-enabled domain, policy enforcement, troubleshooting, and security.

In this context, a comprehensive monitoring system is essential to assist solutions for the above problems [Bhamare et al. 2016]. Surveys [Medhat et al. 2017, Bhamare et al. 2016] in this area point out that advanced monitoring capabilities are required to provide solutions considered key challenges for SFCs such as Traffic Steering.

In theory, a monitoring task needs to store information from all transmitted packets for complete accuracy. In practice, however, this approach would lead to storage and processing scalability issues [Pereira et al. 2017, Shirali-Shahreza and Ganjali 2015]. It is worth mentioning that scalability is the primary challenge of SFC [Bhamare et al. 2016]. To overcome these issues, we can use sampling techniques to avoid the storage and processing of all packets. NetFlow[1] is the most widely-used sampling tool. However, sampling reduces the accuracy to a level that precludes its use for many applications (e.g., traffic engineering) required in today's large scale networks.

Ideally, for SFC purposes, a monitoring task should take into account all transmitted packets at the same time that keeps memory and processing at acceptable levels. Therefore, in this article, we want to solve the problem of how to provide an efficient and scalable mechanism to track network flows in SFC environments. As a starting point, we argue that, in SFC monitoring tasks, exact results are usually not necessary, and a high-quality approximation is enough. This assumption suggests the use of probabilistic data structures (e.g., Bloom Filter, Sketches) that use smaller amounts of memory and require less computation to achieve the desired goals [Pereira et al. 2017]. Such techniques have been widely accepted in network measurements, thanks to their higher accuracy compared to sampling techniques and their speed [Yang et al. 2018].

Aiming at moving forward on the IETF SFC WG standardization threads, in this work, we present **SFCMon**, an efficient and scalable monitoring solution to keep track network flows in SFC-enabled domains. In summary, our main contributions are as follows:

1. The SFCMon, which offers an efficient and scalable monitoring system for SFC environments. For this, it proposes the use of a pipeline of probabilistic data structures to keep track large flows directly inside the network switches, thus reducing both overall latency and signaling overhead;
2. The Proof-of-Concept (PoC) framework, which provides a reference P4-based SFC switch and a Python-based SFC controller. The former follows the RFC 7665 to implement the two SFC components: Classifier and Service Function Forwarder (SFF), enabling stateful in-switch processing. It also provides SFCMon implementation. The latter implements an SFC control plane that provides an operator with a tool for constructing SFCs and associated Service Function Paths (SFPs). Besides, it performs periodic queries for the monitoring data stored by the SFCMon instances. Such data are made available to the operator through the Kibana tool[2];
3. The validation and performance evaluation processes which point to the potential benefits of SFCMon to be part of SFC architecture.

2. SFCMon: A Monitoring Component for SFC

SFCMon consists of an efficient and scalable fine-grained monitoring solution to keep track network flows in SFC-enabled domains. In this section, we discuss the design guidelines and the architecture of SFCMon.

[1]https://www.ietf.org/rfc/ rfc3954.txt
[2]https://www.elastic.co/products/kibana

2.1. Design Guidelines

Probabilistic data structures provide a deterministic number of operations but probabilistic output. These structures use smaller amounts of memory and require less computation to provide a high-quality approximation of the exact results [Pereira et al. 2017]. Probabilistic data structures such as Bloom Filters and Sketches have been widely accepted in network measurements, thanks to their higher accuracy compared to sampling techniques and their speed [Yang et al. 2018].

Bloom Filter (BF) is a probabilistic data structure that provides a memory-efficient approach for insertions and membership queries [Bloom 1970, Grandi 2018]. A BF consists of a 1-bit vector of M cells. For insertion, an element is used as input in K hash functions, which map to different cells in the array. Then, each position is marked with bit 1. For membership queries, an element is used as input in the same K hash functions. An item is considered in the set if all the hash values map to a cell with bit 1.

In BFs, the False Negative Rate (FNR) is always 0, that is, if at least one hash value map to a cell with bit 0 the probability of the element not being part of the set is 100%. However, false positive matches are possible, that is, if all the hash values map to a cell with bit 1, the element is possibly part of the set. Fortunately, the False Positive Rate (FPR) is controllable and can be defined at design time by the equation (1), considering a design with N elements, M cells, and K hash functions. In practice, BF designers focus on the tradeoff among memory, number of operations, and FPR.

$$FPR = (1 - (1 - \frac{1}{M})^{K \times N})^K \tag{1}$$

According to [Broder et al. 2002], a good approximation to obtain lower FPR is to assume $M > K \times N$. In network measurement, one BF can be used within a switch to detect, rapidly and memory-efficiently, whether or not a flow is part of a set of monitored flows [Broder et al. 2002]. However, it cannot store data from these flows.

For instance, **Invertible Bloom Lookup Table (IBLT)** is an extension of BF that allows the storage of key-value pairs [Goodrich and Mitzenmacher 2011]. In addition to the insertions and membership queries, IBLT also allows for updates, deletions, lookups, and a complete listing. For this, each IBLT cell contains three fields:

- $count$: store the number of entries mapped to this cell;
- $keySum$: store the sum of all the keys mapped to this cell;
- $valueSum$: store the sum all the values mapped to this cell.

The listing method allows listing all the key-value pairs being stored in one IBLT (please refer to [Goodrich and Mitzenmacher 2011] for more details). Therefore, using this method, some manager can retrieve a set of IBLTs and list their entries, with a certain probability of failure (only part of the entries are extracted successfully). In this context, one IBLT can be used to keep track flows as well as per flow counters (metrics). However, some issues must be solved.

First, for each insertion, deletion, or update operation on an IBLT, an algorithm must verify whether or not the input belongs to the set (lookup) beforehand to avoid inconsistencies. Besides, the design of the size of the IBLT will generally be determined

by the desired probability for a successful lookup operation, that follows the equation (1) [Goodrich and Mitzenmacher 2011]. Since an IBLT takes up more memory than a BF, such a restriction may render the use of this structure impracticable on low-memory devices. To reduce memory consumption, we can use an approach that applies a combination of a BF and an IBLT. For instance, the lookup operation will now be performed using the BF and the remaining services in the IBLT. By using this approach, we can now target the probability for succeeding in listing entries, that follows the theorem below.

Theorem 1 *As long as the number of cells M is chosen so that* $M > (C_K + \epsilon) \times T$ *for some* $\epsilon > 0$*, the listing method fails with probability* $\mathcal{O}(T^{2-K})$ *whenever* $N \leq T$*, being* N *the number of elements and* K *the number of hash functions.*

It worth noting that C_K is a constant whose value is determined by K and is always less than K. For example, if $K = 7$ then $C_K = 1.721$. Thus, the assumption that the listing method is the key feature that enables us to design lower-memory IBLTs.

Second, even working with lower-memory IBLT, a large number of flows to manage can also make it unfeasible. For example, consider a dataset trace from a 10Gb/s ISP backbone link, recorded in 2016 and available from CAIDA[3]. This trace is 17 minutes long, and each 20-second chunk contains on average about 400,000 5-tuple flows. According to the theorem, an IBLT must be designed with approximately 518,402 cells to meet this load of flows, where $K = 4$. If we consider that $count$ and $valueSum$ have 4 bytes each and $keySum$ has 13 bytes (5-tuple size), we will have a total of 21 bytes per cell and an IBLT that will occupy about 11 Megabytes. We argue that this value is impractical considering that the switch also needs to support other functionalities related to packet forwarding and access control. Besides, according to [Sivaraman et al. 2017], switches should work a limited amount of memory available (about 1.4 Megabytes) per stage to maintain line-rate processing (at 10-100Gb/s).

Therefore, it is infeasible to accurately measure all flows in high-speed networks [Estan and Varghese 2003]. One solution to this problem is to keep track only the large/heavy flows. A large flow is a flow that includes more than a certain percentage of the packets during a given time interval [Afek et al. 2018]. Studies have shown that the majority of flows tend to be short (mice flows) on high-speed links. In [Mori et al. 2004], the authors showed that a 5% to 10% of flows by number account 60% to 80% of traffic by volume. According to [Sivaraman et al. 2017], identifying flows with large traffic volumes in the data plane is essential for several applications such as flow-size aware routing, DoS detection, and traffic engineering. Besides, [Estan and Varghese 2003] states that both traffic monitoring and attack detection can benefit from accurately measuring only the large flows. Thus, we argue that it is sufficient to keep track only large flows to meet the advanced management capabilities required in SFC environments.

In this context, we can use **Count-Min Sketches (CMS)** to detect large flows. CMS is probabilistic data structure whose goal is to consume a stream of events (e.g., packets) and count the frequency of different types of events. Theses frequencies can be queried at any time [Cormode and Muthukrishnan 2005]. CMS is a two-dimensional vector of w columns and d rows. Given (ϵ, δ), w and d are defined at design time by setting $w = \lceil \frac{e}{\epsilon} \rceil$ and $d = \lceil ln(\frac{1}{\delta}) \rceil$. Each row has a hash function associated $H = (h_1, ..., h_d)$,

[3]http://www.caida.org/data/passive/passive_2016_dataset.xml

chosen from a pairwise-independent family. In turn, each hash function maps to one of the w columns.

A basic CMS has two methods: *Update* and *Estimate*. Once a new event i arrives, for each row j, the *Update* method applies the associated hash function to obtain the column index $h_j(i)$. Then, it increments the index by one. On the other hand, the *Estimate* method returns the estimated frequency of an specific event, i.e. $\hat{f}_i = min_{1 \leq j \leq d}(CMS[j, h_j(i)])$. This estimate ensures that $\hat{f}_i \leq f_i + (\epsilon N)$ with probability $1 - \delta$, where f_i is the true frequency and $N = \sum_{i=0}^{n} f_i$ is the stream size. Note, CMS occasionally overestimates frequencies, but it never underestimates frequencies.

Therefore, we can use CMS to detect large flows. We can formalize this by taking the CMS to solve the *ϵ-approximate heavy hitters problem* [Roughgarden and Valiant 2015]. In this problem, the input is a stream of length T and the user-defined parameters k (where $T >> k$) and ϵ (error tolerance). As an output of a single pass over the stream, we want a list L of elements such that:

- x is in L if x occurs at least $\frac{T}{k}$ times in the stream;
- Every value in L occurs at least $\frac{T}{k} - \epsilon \times T$ times in the stream.

According to [Roughgarden and Valiant 2015], a good approximation to detect heavy hitters with high probability is to assume $\epsilon = \frac{1}{2 \times k}$.

2.2. SFCMon Architecture

To achieve the desired goals, SFCMon implements a pipeline of probabilistic data structures to detect and store heavy/large flows while evicting lighter flows over time. Besides, for each stored flow, SFCMon maintains a set of per-flow counters. For instance, the SFCMon pipeline includes the following structures.

- **SFCMonCMS:** an CMS which is responsible by detect large flows, by solving the *ϵ-approximate heavy hitters (HH) problem*;
- **SFCMonBF:** an BF is used to reduce the SFCMonIBLT size and is responsible for answering lookup queries (membership) concerning the set of current large flows being tracked;
- **SFCMonIBLT:** an IBLT which is responsible for keeping track all large flows needed to assist in the monitoring tasks.

The general process of SFCMon is described in Algorithm 1. Once an SFC encapsulated packet arrives, a flow identifier ($FlowID_1$) is generated using data from the source IP address and from two Network Service Header (NSH) fields: Service Path Identifier (SPI) which identifies a Service Function Path (SFP) and Service Index (SI) which provides location within the SFP. In turn, the $FlowID_1$ is used as input for the *Update* method from the *SFCMonCMS*. Then, if the amount of processed packets from the packet stream crosses a determined threshold ($Threshold_{Eval}$), the SFCMon calls the *Estimate* method from the *SFCMonCMS*. This threshold is necessary since SFCMon does not expect to process all streams to detect HHs, it starts the detection process after this threshold is crossed, as packets are processed, and continues until the entire stream is received. If this estimate is greater than a given threshold ($Threshold_{HH}$), the flow to which the packet belongs will be considered a HH and the algorithm advances to the next stage of the pipeline.

Algorithm 1: SFCMon Packet Processing

Input: A list of field values: $spi, si, srcIP, dstIP, srcPort, dstPort, proto$.
Input: The packet size value: $size_{packet}$.
begin
  $FlowID_1 \longleftarrow (spi, si, srcIP)$;
  $CtrPackets \longleftarrow CtrPackets + 1$;
  $UpdateCMS(FlowID_1)$;
  if $CtrPackets \geq Threshold_{Eval}$ **then**
    $Estimation \longleftarrow EstimateCMS(FlowID_1)$;
    if $Estimation \geq Threshold_{HH}$ **then**
      $FlowID_2 \longleftarrow (spi, si, srcIP, dstIP, srcPort, dstPort, proto)$;
      $IsStored \longleftarrow MembershipQueryBF(FlowID_2)$;
      if $\neg IsStored$ **then**
        $InsertBF(FlowID_2)$;
        $InsertIBLT(FlowID_2)$;
      end
      $UpdateIBLT(FlowID_2, size_{packet})$;
    end
  end
end

Algorithm 2: InsertIBLT Function

Data: A finite set $H = \{H_1, H_2, \ldots, H_k\}$ of hash functions.
Input: A tuple of field values that describes a flow identification: $flowID$.
begin
  for $i \leftarrow 1$ ***to*** k **do**
    $Index \longleftarrow H_i(flowID)$;
    $SFCMonIBLT[Index].count++$;
    $SFCMonIBLT[Index].keySum \longleftarrow SFCMonIBLT[Index].keySum \oplus flowID$;
  end
end

Algorithm 3: UpdateIBLT Function

Data: A finite set $H = \{H_1, H_2, \ldots, H_k\}$ of hash functions.
Input: A tuple of field values that describe a flow identification: $flowID$.
Input: The packet size value: $size_{packet}$.
begin
  for $i \leftarrow 1$ ***to*** k **do**
    $Index \longleftarrow H_i(flowID)$;
    $SFCMonIBLT[Index].valueSum[0]++$;
    $SFCMonIBLT[Index].valueSum[1] \longleftarrow SFCMonIBLT[Index].valueSum[1] + size_{packet}$;
  end
end

In this next stage, a new flow identifier ($FlowID_2$) will be generated using the SPI and SI information in addition to the 5-tuple data (source and destination IP addresses and ports, and protocol type). This time, we used more information aiming to provide a more fine-grained flow monitoring system. According to [Zhou et al. 2018], the 5-tuple contains the main information of network link and existing monitoring mechanisms usually have good efficiency. The algorithm then performs the lookup query in *SFCMonBF* (using the $FlowID_2$ as input) to find out if the flow already exists in *SFCMonIBLT*. If so, the algorithm only updates *SFCMonIBLT* (*UpdateIBLT* method). Otherwise, the algorithm inserts the new flow within *SFCMonBF* (*InsertBF*) and *SFCMonIBLT* (*InsertIBLT*) beforehand, and then updates the latter.

Algorithms 2 and 3 provide a detailed view of the operation of the *InsertIBLT* and *UpdateIBLT* methods respectively. Please refer to [Goodrich and Mitzenmacher 2011] for more details on how an IBLT works.

It is worth noting that those data structures must be flushed periodically by a manager to remove flows that are no longer active. Besides, such an approach also helps in keeping the low probability of false positives [Patgiri et al. 2018]. In this context, SFCMon considers as stream any set of packets that it can process before having its structures reset.

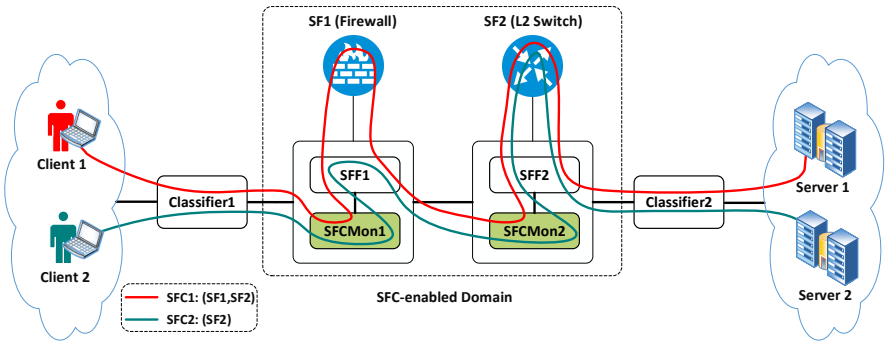


Figure 1. SFCMon as an SFC Component.

Finally, we propose SFCMon as a new SFC component. For this, we need to define how it fits into the SFC-enabled domain. Figure 1 shows how SFCMon relates to the other SFC components. We argue that it must be logically positioned after the Classifier and before the Service Function Forwarder (SFF) so that it can process all the encapsulated packets before they are forwarded to a next component.

3. SFCMon Analyzer

In this section, we present the whole process of planning and conducting the experiments which aim to analyze the SFCMon described in Section 2, focusing on its capability in detect large flows, performance, and scalability.

3.1. P4-based SFC Solution

We implement a Proof-of-Concept (PoC) framework aiming to validate and evaluate the SFCMon. The approach towards our PoC use case implementation follows a typical SDN-based SFC architecture [Boucadair 2016] for the design and development of two prototypes: an SFC-enabled Switch and an SFC Controller, as shown in Figure 2.

The SFC-enabled Switch is in charge of the SFC data plane. We follow the RFC 7665 [Halpern and Pignataro 2015] to provide an implementation that includes two SFC components: Classifier and Service Function Forwarder (SFF). It is worth mentioning that those components are sufficing to create SFC-enabled domains. Besides, our switch also includes an SFCMon reference implementation.

We develop our prototype as a P4 program [Bosshart et al. 2014]. P4 is a programming language to run complex processing in the data plane. It allows the developer to implement its actions using if-else clauses, algebraic and boolean expressions, and different types of counters and hash functions, allowing the calculation of in-switch high-level features and the creation of customizable traffic analysis mechanisms. In this context, we implement both Classifier and SFF as P4 tables (see Figure 2) and SFCMon data structures as an array of registers, which can store states that persist across multiple packets

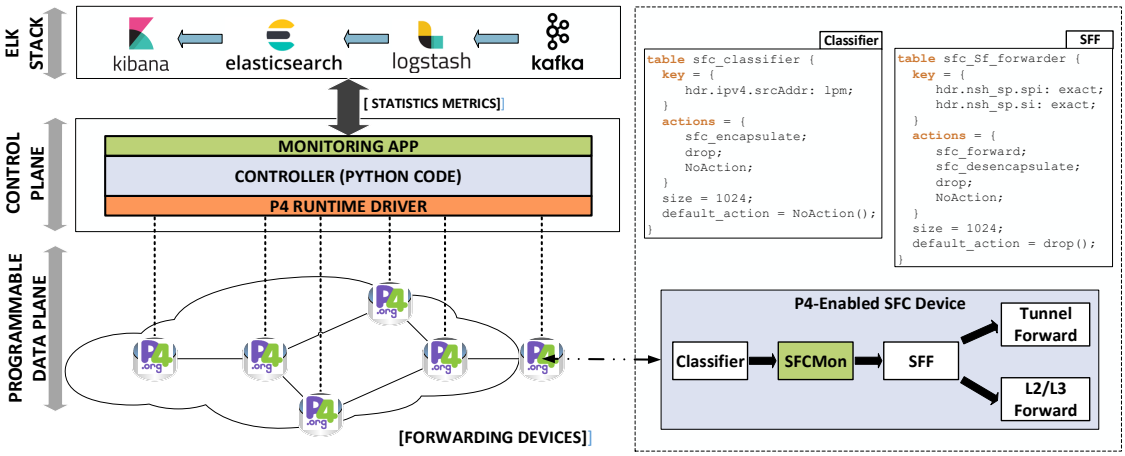


Figure 2. Prototype Architecture.

of a flow (i.e., flow states). To support SFCMon, we extend behavioral-model (bmv2[4]), a public-domain P4 virtual switch, to enable support for multiple pairwise independent hash functions. For this, we implement the algorithm MurmurHash3[5], which yields the 32-bit hash value. Next, we define 22 independent hash functions by just varying the seed of MurmurHash3. Moreover, we follow the RFC 8300 [Quinn et al. 2018] to define a P4 header for the Network Service Header (NSH), an SFC encapsulation protocol required to support the SFC architecture. Usually, the connection between SFC components is performed by NSH.

The SFC Controller implements an SFC control plane. It has been developed in Python and uses a P4 Runtime API[6] to interact with the SFC data plane components. Our prototype performs the following functionalities.

- It provides a Northbound API that enables an operator to manage SFCs and associated Service Function Paths (SFPs);
- It periodically collects statistical data from the *SFCMonIBLTs* on all SFC-enabled Switches. The objective is to keep track flow records (flow ids, packet, and byte counters). Besides, such records are made available to the operator through the Kibana tool[7];
- It periodically resets all SFCMon data structures on all SFC-enabled Switches. As seen in Subsection 2.2, the desired goal is to remove flows that are no longer active. Besides, such an approach also helps in keeping the false positive probability low.

Finally, our framework automates the creation of SFC-enabled domains with any topology for further validations and evaluations. For this, it uses the Mininet[8] emulator

[4]https://github.com/p4lang/behavioral-model
[5]https://en.wikipedia.org/wiki/MurmurHash
[6]https://p4.org/api/p4-runtime-putting-the-control-plane-in-charge-of-the-forwarding-plane.html
[7]https://www.elastic.co/products/kibana
[8]http://mininet.org/

and the bmv2. Therefore, our framework is not only intended to analyze SFCMon but also to evaluate future solutions that might arise for SFCs.

3.2. Evaluating the Capability to Detect Large Flows

To evaluate the SFCMon's ability to detect large flows, we develop a Python program that simulates the execution of an adapted implementation of Algorithm 1, i.e., instead of performing the operations of lines 9 to 14, we just store all five tuples in a structure so that in the end we compare these with a ground truth.

For this experiment, we use as input a public dataset of a backbone link of 10GB/s, measured in 2019 and available by CAIDA. This dataset contains 30 minutes of measurement, with approximately 936 million packets and 7 million flows (considering only the source IP as an identifier). In this experiment, SFCMon processes the entire dataset to evaluate the following performance metrics:

- False Positive Rate (FPR): refers to the proportion of flows that have been reported as HHs but which are not true HHs;
- Accuracy: evaluates how close predicted results are to true results;
- Recall Rate (True Positive Rate): refers to the proportion of flows that are true HHs and that have been reported as HHs;
- Precision Rate: refers to the proportion of flows that have been reported as HHs and that are true HHs.

Moreover, we evaluate the above metrics concerning a single factor: the time window to delimit a stream of packets. It is worth mentioning that SFCMon considers as stream any set of packets that it can process before having its structures reset by the controller. In this context, we assumed the following levels: 6, 7, 10, 15, 30, and 60 seconds.

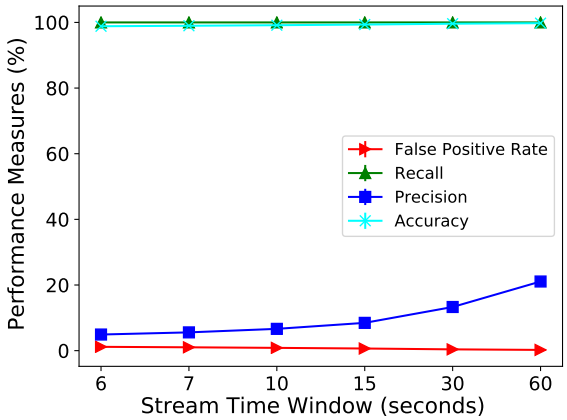


Figure 3. Accuracy analysis concerning the SFCMon's capability to detect large flows.

For this experiment, we consider that an HH is a flow that has more than 0.1% of the total packets that SFCMon has processed so far. Besides, we set the value of $Threshold_{Eval}$ in 1000 packets, i.e., SFCMon only begins performing HH tests after processing more than 1000 packets in a given time window. Moreover, we follow the

recommendations of [Roughgarden and Valiant 2015] to design our CMS. In this context, considering a packet stream size of 10 million, we reach the following parameters: $w = 5436$ and $d = 5$. Besides, we set the cell size to 3 bytes. Therefore, our CMS consumes a total of approximately 82 Kilobytes, less than 1.4 Megabytes (see Subsection 2.1).

Finally, to obtain statistically significant results, we split the dataset into 30 chunks with 60 seconds of measurement each and use each of them as input to the SFCMon. Therefore, each point in the graph contains an average across the 30 trials, as well as the confidence interval obtained from the t-student test.

Figure 3 summarizes the simulation results. We can conclude that SFCMon achieved excellent results, since both accuracy and precision always produce their results close to 100%, regardless of the size of the time window. Also, the FPR kept its results always close to zero, which shows that SFCMon reports a negligible amount of flows that are not true HHs.

However, Precision presented poor results (less than 30%) when compared to the Accuracy. These weak results are provoked by the existence of False Positives (FPs), which, although a negligible amount, is far superior to the number of True Positives (TPs). Since SFCMon does not expect to receive the entire stream to detect HHs, it eventually collects some FPs. This amount decreases as we increase the size of the time window (see Figure 3), since the amount of TPs increases. Nevertheless, it is worth emphasizing that the impact of this evidence on the viability and performance of our solution is negligible since the number of FPs is only a small percentage of the total amount of flows in a stream (2% to 3% on average).

3.3. Evaluating Performance and Scalability

By using our PoC framework, we perform experiments aiming to evaluate the SFCMon regarding its performance and scalability. For this, we have created a typical topology of an SFC-enabled switch connecting two hosts.

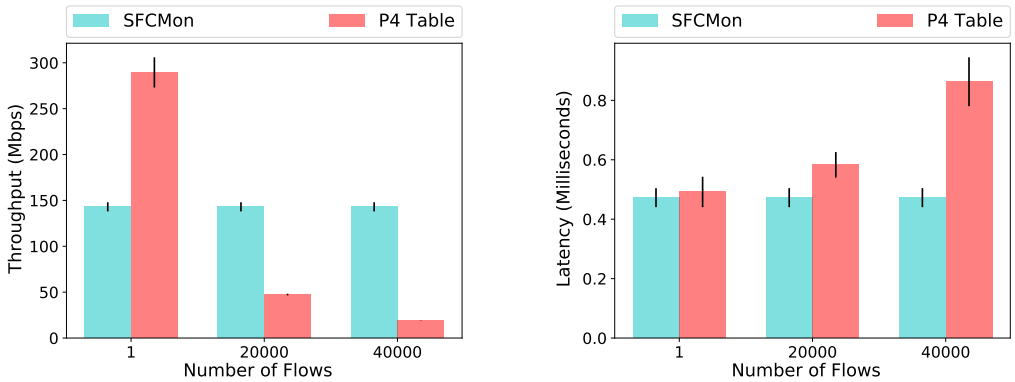
For these experiments, we decide to compare the SFCMon with a P4 table, when used as monitoring solutions for an SFC-enabled domain. Just like the SFCMon, P4 table is a stateful object. However, unlike SFCMon, a P4 table provides deterministic outputs at the cost of a probabilistic number of required operations. Using a P4 table to keep track large flows would be the best since it allows to detect HHs with greater precision since it does not underestimate or overestimate the detection metrics. In addition, it will enable you to manage the flows without producing false positives.

We evaluate the above solutions by considering Throughput (Mbps) and Latency (Milliseconds) metrics and compared their results when varying the number of active flows (1, 20000, and 40000), i.e., the amount of flow currently being tracked. The SFC Controller is responsible for creating these flows in advance. We designed SFCMon's probabilistic data structures to support up to 10 million streams simultaneously, following the recommendations described in Subsection 2.1. For this, we design our CMS with the same values defined in Subsection 3.2. Besides, from the experiments performed in Subsection 3.2, we identify that, for a time window of 20 seconds, the number of HH reported is approximately 200. Therefore, considering the time window in 20 seconds and $K = 3$, we reach the following parameters for SFCMonBF and SFCMonIBLT, respectively: $M_{SFCMonBF} = 1500$ and $M_{SFCMonIBLT} = 245$. For SFCMonIBLT, we consider

that $count$ and $valueSum$ have 4 bytes each and $keySum$ has 13 bytes (5-tuple size), thus a total of 21 bytes per cell and an IBLT that will occupy about 5 Kilobytes per switch, less than 1.4 Megabytes (see Subsection 2.1). It is worth mentioning that, in this case, we only need a small portion of bandwidth (about 4 Mbps per SFC-enabled switch with 10Gbps links) to send statistical data from the SFCMonIBLT to the controller every 10ms. This short time scale provides better visibility in data center networks [Li et al. 2016].

For traffic generation, we choose the Iperf[9] and D-ITG[10] as packet generators. We use Iperf to generate TCP traffic as fast as possible. In this way, we can compare the two solutions according to the impact of the number of active flows in Throughput. On the other hand, we use D-ITG to generate a single UDP flow. For this, we assume a "typical" average packet size of 512 bytes and a transmission rate of 10 Mbps to avoid packet loss. By using D-ITG, we can compare the two solutions according to the impact of the number of active flows in Latency, since this tool allows you to obtain end-to-end traffic statistics such as bandwidth, latency, jitter, and packet loss.

For each relation "Number of Active Flows x Metric x Monitoring Approach", we perform 30 executions with a duration of 10 seconds each, as a way to generate random samples of size 30 and compare the solutions with statistical significance. Finally, as most of the samples showed a tendency to non-normality of the data, the experiments applied the Wilcoxon Test to compare the medians for each sample, while using the confidence intervals from this test with confidence 95%. We decide to work with the median because it is less sensitive to possible outliers. We run the experiments using a Server with Intel Core i7-8550U 1.80GHz processors and 16 GB RAM.



(a) Throughput versus Number of Active Flows.

(b) Latency versus Number of Active Flows.

Figure 4. SFCMon performance evaluation results.

Figure 4 show the results. In terms of performance, with only a few thousand active flows, SFCMon can be better than a P4 table considering both Throughput (see Figure 4a) and Latency (see Figure 4b). It is worth mentioning that this reference implementation of SFCMon was designed to support a stream of up to 10 million flows! This gain in performance occurs because the SFCMon requires only simple operations (e.g., hash function calculations, vector updates), which are independent of the number of entries.

[9]https://iperf.fr/

[10]www.grid.unina.it/software/ITG/

On the other hand, a P4 table requires a lookup in a flow table, a type of operation whose time varies depending on the number of entries, which ends up making it a performance bottleneck.

In terms of scalability, since the SFCMon only uses probabilistic data structures, the number of operations required is deterministic, i.e., regardless of the number of entries, the delay processing time will always be the same. As a consequence, both Throughput (see Figure 4a) and Latency (see Figure 4b) remain the same with increasing active flows.

Therefore, presented initial experiments demonstrate that SFCMon introduces a negligible performance penalty while providing significant scalability gains. These results point to the potential benefits of SFCMon to be part of SFC architecture, aiming to assist with more advanced monitoring applications in an efficient and scalable way.

4. Related Work

We found some studies that focus on the use of probabilistic data structures for performance measurement and volume counting. UnivMon [Liu et al. 2016] proposed an application to detect Heavy Hitters (HHs) and use this information to do better traffic management or monitoring. For this, UnivMon uses a CMS to perform top-k HH detection. The HashPipe [Sivaraman et al. 2017] provides an in-switch processing algorithm that uses a pipeline of hash tables to track heavy flows with high accuracy, aiming to evict lighter flows from switch memory over time. Recently, the Elastic Sketch [Yang et al. 2018] proposed a novel sketch to achieve accurate network measurements no matter how traffic characteristics vary. For this, Elastic Sketch uses a pipeline of hash tables and a technique named Ostracism to keep large flows separated from the mouse flows. However, different from SFCMon, UnivMon, HashPipe, and Elastic Sketch do not save network flow data for further collection by external applications. In contrast, SFCMon uses an IBLT to store flow data as key-value pairs (i.e., the flow tuples and the packet and byte counters).

Closest to our efforts, FlowRadar [Li et al. 2016] provides a monitoring tool that keeps track flows within the constraints of emerging programmable switches. Similar to SFCMon, FlowRadar uses a BF for flow filtering and an IBLT to store flow data and per-flow counters. However, FlowRadar applies for all transient flows while SFCMon store only large flows. In Subsection 2.1, we showed that even working with lower-memory IBLT, a large number of flows to manage can also make it unfeasible. In this case, SFCMon can scale to a large number of flows.

5. Conclusion and Future Work

In this work, we presented SFCMon, an efficient and scalable monitoring solution to keep track network flows in SFC environments. To achieve the desired goals, SFCMon works with a pipeline of probabilistic data structures to detect large flows and to store some of their features such as packet counting. Based on the SFC reference architecture defined by RFC 7665 [Halpern and Pignataro 2015], we implemented a Proof-of-Concept (PoC) framework, which provides a P4-based SFC switch, a Python-based SFC Controller, and an SFCMon reference implementation. The source code is available in the following GitHub repository: `https://github.com/michelsb/SFCMon`. Presented initial experiments point to the potential benefits of SFCMon to be part of SFC architecture.

We can conclude that the results are promising. However, it is worth mentioning that these experiments are only in their initial phase, enabling several opportunities for improvement. For example, the method for setting the thresholds is static. For the growth of Precision metric (see Figure 3), this method can be improved in future work by using adaptive thresholds based on Reinforcement Learning and Deep Reinforcement Learning, for instance. Additionally, it is essential to evaluate how effective SFCMon is compared to related work when collecting data for applications. In this case, for future work, it would be good to provide such an analysis in terms of benefits for applications. Possibly use cases in which SFCMon is advantageous, depending on the service and its traffic profile.

Finally, taking into account latest efforts made by IETF/IRTF, future activities include pursuing contributions to standardization along the way SFCMon, as an SFC component, can be used to assist with more advanced monitoring applications in an efficient and scalable way.

6. Acknowledgements

This work was partially funded by the National Science Foundation (NSF-USA) and the Rede Nacional de Ensino e Pesquisa (RNP-Brazil) under the "EAGER: USBRCCR: Securing Networks in the Programmable Data Plane" project. This work was developed while Stenio Fernandes was with UFPE, Brazil. He is now with Element AI, Canada.

References

Afek, Y., Bremler-Barr, A., Feibish, S. L., and Schiff, L. (2018). Detecting heavy flows in the sdn match and action model. *Computer Networks*, 136:1 – 12.

Bhamare, D., Jain, R., Samaka, M., and Erbad, A. (2016). A survey on service function chaining. *Journal of Network and Computer Applications*, 75:138 – 155.

Bloom, B. H. (1970). Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426.

Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., and Walker, D. (2014). P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95.

Boucadair, M. (2016). Service Function Chaining (SFC) Control Plane Components & Requirements. Internet-Draft draft-ietf-sfc-control-plane-08, Internet Engineering Task Force. Work in Progress.

Broder, A., Mitzenmacher, M., and Mitzenmacher, A. B. I. M. (2002). Network applications of bloom filters: A survey. In *Internet Mathematics*, pages 636–646.

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

Estan, C. and Varghese, G. (2003). New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice. *ACM Trans. Comput. Syst.*, 21(3):270–313.

ETSI (2015). Network functions virtualisation (nfv) - network operator perspectives on industry progress. *White Paper*.

Goodrich, M. T. and Mitzenmacher, M. (2011). Invertible bloom lookup tables. In *2011 49th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pages 792–799.

Grandi, F. (2018). On the analysis of bloom filters. *Information Processing Letters*, 129:35 – 39.

Halpern, J. M. and Pignataro, C. (2015). Service Function Chaining (SFC) Architecture. RFC 7665.

Li, Y., Miao, R., Kim, C., and Yu, M. (2016). Flowradar: A better netflow for data centers. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 311–324, Santa Clara, CA. USENIX Association.

Liu, Z., Manousis, A., Vorsanger, G., Sekar, V., and Braverman, V. (2016). One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 101–114, New York, NY, USA. ACM.

Medhat, A. M., Taleb, T., Elmangoush, A., Carella, G. A., Covaci, S., and Magedanz, T. (2017). Service function chaining in next generation networks: State of the art and research challenges. *IEEE Communications Magazine*, 55(2):216–223.

Mori, T., Uchida, M., Kawahara, R., Pan, J., and Goto, S. (2004). Identifying elephant flows through periodically sampled packets. In *Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement*, IMC '04, pages 115–120, New York, NY, USA. ACM.

Patgiri, R., Nayak, S., and Borgohain, S. K. (2018). Preventing ddos using bloom filter: A survey. *CoRR*, abs/1810.06689.

Pereira, F., Neves, N., and Ramos, F. M. V. (2017). Secure network monitoring using programmable data planes. In *2017 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pages 286–291.

Quinn, P., Elzur, U., and Pignataro, C. (2018). Network Service Header (NSH). RFC 8300.

Roughgarden, T. and Valiant, G. (2015). Approximate Heavy Hitters and the Count-Min Sketch. `http://theory.stanford.edu/~tim/s15/l/l2.pdf`. Online; accessed 01 April 2019.

Shirali-Shahreza, S. and Ganjali, Y. (2015). Rewiflow: Restricted wildcard openflow rules. *SIGCOMM Comput. Commun. Rev.*, 45(5):29–35.

Sivaraman, V., Narayana, S., Rottenstreich, O., Muthukrishnan, S., and Rexford, J. (2017). Heavy-hitter detection entirely in the data plane. In *Proceedings of the Symposium on SDN Research*, SOSR '17, pages 164–176, New York, NY, USA. ACM.

Yang, T., Jiang, J., Liu, P., Huang, Q., Gong, J., Zhou, Y., Miao, R., Li, X., and Uhlig, S. (2018). Elastic sketch: Adaptive and fast network-wide measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 561–575, New York, NY, USA. ACM.

Zhou, D., Yan, Z., Fu, Y., and Yao, Z. (2018). A survey on network data collection. *Journal of Network and Computer Applications*, 116:9 – 23.