FLight: A Fast and Lightweight Elephant-Flow Detection Mechanism

Amer AlGhadhban and Basem Shihada
CEMSE Division, King Abdullah University of Science and Technology
Email: {amer.alghadhban, basem.shihada}@kaust.edu.sa

***Abstract*—The imperative of detecting an elephant flow, carrying the majority of the transmitted data, has been demonstrated in a wide spread of network applications and services. Existing detection solutions are categorized into in-host, in-network, and centralized approaches. The in-host approach requires full access to all hosts in the network, while the in-network requires stringent settings, particular features or programmable hardware to detect the elephant-flows with high accuracy. Such requirements are not always feasible in today's networks and can create an error prone environment. Moreover, the network switches strive to maintain enough processing capacity to run their primary tasks rather than being overwhelmed with any additional overhead. The centralized approaches suffer from extensive detection delays and being inaccurate. In this work, we propose FLight, a fast, lightweight and adaptive mechanism for detecting elephant-flows while improving the detection accuracy and speed. FLight leverages the TCP communication behavior for its detection algorithm, it demonstrates a 100% elephant-flow detection accuracy, and is 242× faster than other centralized solutions. The overhead of FLight is close to the sFlow packet sampling solution with 1-to-10K sampling-rate and 5× lower than OpenSample-TCP. We used real packet traces in validating the FLight visibility and lightness. The overhead of notification messages in the data sets is reduced by 62×.**

Keywords*—*Flow classification; packet-sampling; flow management; TCP characteristics.

I. Introduction

Identifying the set of flows carrying the majority of transmitted data, i.e., elephant-flows, is a necessary responsibility for several network services and applications including traffic management, troubleshooting, accounting, flow caching, and anomaly detection. For instance, the elephant-flow detection helps the traffic management solutions to efficiently utilize available network resources. The well-known data center (DCN) load balancers, such as Hedera [1], and Hermes [2] has found that performing their algorithms on mice flows is useless and can degrade the network performance significantly. In flow caching, the flow-entries of elephant-flows are given higher priority to accelerate the lookup time. Similarly, identifying elephant-flows helps the detection of malicious traffic. Also, the traffic shaping system found that limiting the rate of the elephant flows enables the mice flows to enjoy the path capacity at their will.

Existing elephant flow detection schemes can be categorized into in-host [3], in-network [4]–[6] and centralized [1], [7]–[9]. As the name suggests, the in-host scheme is implemented in the host kernel where packets/bytes are counted before being inject into the network. When the counter exceeds a pre-defined threshold value, the flow is considered as an elephant. In order to implement this, the in-host solutions need to modify the kernel network stack of the DCN hosts, but this requires higher privileges to execute that cannot always be granted (e.g., closed source operating system and VMs of other autonomous systems). Also, the operation itself (i.e., loading a kernel module in every tenants) is unpleasant administrative overhead and prone to misconfiguration failures.

The in-network scheme detects elephant flows at network switches. The flow information along with their counters are maintained at the switch memory and the comparison task is performed internally at the switch hardware (e.g., HashPipe [4] and OpenSketch [6]) or software (e.g., Software Defined Counter [5]), or otherwise is exposed to a centralized entity (e.g., FlowRadar [10], and NetFlow). One of the main challenges of in-network mechanism is the limited resources of network switches (e.g., memory, processing delay, and power consumption). As a result, the researchers introduced the hybrid solution where some of the classification functions are migrated into cheaper hardware [6] or into a centralized entity [10]. For example, OpenSketch [6] performs the flow classifications at the fast, small and expensive TCAM memory, while the counters are maintained at a larger, slower, and cheaper memory (SRAM). On the other hand, some solutions (e.g., FlowRadar [10] and NetFlow) the flow-statistics are exported from the switches to a centralized entity either periodically or with every cache miss. Another challenge in the in-network scheme, as in HashPipe [4], and OpenSketch [6], is to amend the existing network switches in the DCN or to use a programmable hardware (e.g., P4) to run their algorithms.

Although the centralized scheme is similar to the in-network hybrid scheme, there are no serious modifications to the network switches or end host devices. It employs the available network features (e.g., packet sampling, port mirroring, or statistical polling) to collect network statistics and forward them to a centralized collector which is usually programmed to perform the flow classification functions. However, the existing centralized schemes have some limitations including high monitoring overhead, hardware modifications, latency and/or accuracy [7]–[9], [11].

Can we reap this insight to redesign an elephant-flow detection algorithm without the need to use special purpose hardware, network switches modifications, or to amend the end-host's kernel? The solution we introduce satisfies these requirements in the affirmative. In this paper, we propose FLight, a fast, lightweight and adaptive flow sampling framework. Unlike the standard sampling methods, the packet samples here are collected from every flow and the pace of the sampling rate is proportional to the rate of the flows. Rather than using the conventional packet sampling or programmable hardware methods to tackle the aforementioned challenges, FLight leverages the TCP communication behavior. We have

further categorized it into two phases: low-frequency and high-frequency[1]. In the low-frequency phase, the collector acquires initial network statistics (e.g., TCP initial sequence number and TCP port number), while in the high-frequency phase it exploits detailed flow statistics (e.g., ACK) to be utilized in the elephant-flow detection algorithm. This enables speeding up the classification process and minimizes the associated delays. Furthermore, other than randomly sampling from all the network flows, we collect samples from low-frequency packets and compare them with the collected samples from the high-frequency phase to increase the detection accuracy.

Unfortunately, forwarding all the captured packets (i.e., low-frequency and high-frequency packets) to the central unit through the data network can be an expensive overhead, in the era of bytes/dollar, as well as it adds an extra delay. Therefore, in this work, we introduce a method for mitigating the propagation overhead by collecting indicative packets from the high-frequency phase until the flow is classified once. In order to further reduce the overhead, those packets are steered into a virtual interface to sample them separately.

In summary, our proposed work mitigates the above challenges along five dimensions:

- **Fast:** The detection of elephant-flow needs to be in a time scale proportional to the length of the major traffic carriers (i.e., $< 10\%$ of the average elephant-flow length). The speed of detection in FLight is in milliseconds timescale faster than similar solutions such as OpenSample-TCP and close to Planck [8] during low network traffic.
- **Accurate:** FLight demonstrates high detection accuracy of elephant flows with different traffic loads and link capacities and it outperforms sFlow and OpenSample-TCP.
- **Light:** The proposed solution demonstrates low processing and communication overheads. We compared the communication as well as the processing overheads of FLight with sFlow and OpenSample-TCP. We found that the overheads of FLight in both processing and communication are close to those of sFlow (1-to-10K).
- **Minimal adoption overhead:** The literature has many efficient solutions that are not easily adopted due to overstretched requirements or re-craft ambiguous configurations. Hence, the adoption demands for a proposed method need to be as minimal as possible. In this work, we take a firm decision to work within the boundaries of existing DCN switches capabilities while keeping the end hosts kernels untouched.

The rest of the paper is organized as follows. The main functionality of FLight and how the elephant-flow is detected are presented in Sec. II. Thereafter the evaluation and performance results of FLight are shown in Sec. III. Finally, conclusions are drawn in Sec. IV.

II. FLight Functionality

In this section, we discuss the design decisions of FLight. First, how the elephant-flows have been detected and how the involved challenges have been mitigated. Second, how to capture the packets of interest. Finally, how the propagation overhead has been minimized.

[1]In this work we use low-frequency phase and first phase as well as high-frequency and second phase alternatively.

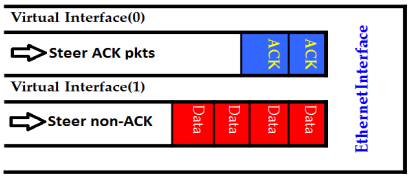


Fig. 1: On every edge switch the FLight steers the ACK packets into a particular virtual interface and the non-ACK packets to another virtual interface. sFlow is configured on the ACK packet interface.

A. Overview

The FLight, elephant-flow detection algorithm, has two main components; the collector and the classifier. The collector is responsible for learning the general flow information from the messages of the first phase. The classifier is responsible for detecting elephant-flows from the captured messages. The classifier is an API extension of OpenFlow controller.

According to the Internet studies [12] and DCN traffic characteristics [13] [14], TCP carries the majority, almost 99% in a DCN, of the bytes. Moreover, the Internet studies found almost all the UDP flows are small in size and initiated by well-known protocols. Due to these findings and similar to previous solutions in this work, we focus on TCP traffic [3] [7]. A TCP flow starts with three-way handshaking to set the initial sequence number and prepare some connection parameters. The three steps are SYN, SYN+ACK and ACK packets, respectively. The SYN, SYN+ACK, and FIN/RST flags appear only once in every flow. Also, the sender sends the data packets labeled with a sequence number, while the receiver sends an acknowledgment packet (ACK) marked with an acknowledgment number to inform the sender that the transmitted bytes have been successfully received.

It is clear that a TCP flow can be divided into two independent phases. The first phase is to establish and terminate a session (i.e., the connection establishment/termination process). The second phase is for data transmission. The packet of the first phase is used in this work to enable the controller learning useful network statistics, such as the number of flows in every path and the flow arrival-rate and maintain global network view. Since the packets of the second phase carry in their headers valuable information about the flow size, we used them in elephant flow detection algorithm. Unfortunately, the packets of the first phase (i.e., low-frequency) have an arrival-rate proportional to the flow arrival-rate, where the packets of the second phase, high-frequency, their arrival-rate is proportional to the data packets. Capturing these packets introduce a high overhead on the switches as well as the collector. Thereby, we adopt two main methods to reduce this overhead; reduce their frequency and keep them at the data-plane as much as possible.

B. The Elephant-Flow Detection Algorithm

A single packet of every TCP flow offers partial information about the flow size (i.e., the TCP sequence number) which is useless when it is individually considered. However, when two packets are captured from the same TCP flow, the collector can discover the number of bytes that have been transmitted between the two captured packets. This technique is the fundamental idea of the OpenSample-TCP and Planck

solutions. The difference between the TCP sequence numbers of the captured packets and the time of capturing are used to measure the link utilization. The OpenSample-TCP used conventional packet sampling (i.e., sFlow), where Planck utilized port-mirroring to accelerate the measurement process ($\approx$ 1 millisecond). Also, Planck needs to find an alternative way to directly connect the centralized collector with every edge switch to avoid adding an extra overhead and congestion from forwarding the mirrored packets through the data network. Moreover, the captured samples could be from the end of the flow and repeatedly from the same flow.

In order to use the packet sampling for elephant flow detection, the two samples need to be approximately originated from the beginning of the flow and from every flow. The sampling rate need to be adaptive to the traffic load and the already classified flow is not captured again. Accordingly, in this work, the data-plane device has been configured to capture the packets of the TCP first phase (i.e., SYN+ACK and RST/FIN). The classifier reads the initial sequence number (ISN) of every TCP flow, which is the first half of the required information to classify a flow, and stores it in the flow-information table. The classifier needs to compare the ISN with every captured packet from the same flow until a threshold value is reached. However, the computational and the network overhead of propagating every packet to a centralized collector, hinder the adoption of this method in a centralized fashion. Therefore, rather than capturing the data packets themselves, here we propose to configure the data-plane devices to capture the ACK packets. The ACK message has the acknowledgment number and the acknowledgment sequence number. For example, when ACK_1 and ACK_{th} of flow f_A, the number of transmitted bytes between these two ACKs is (th - 1). The ACK sequence number only presents the bytes that have been successfully received which means the lost or out-of-order packets are not counted. Thus, the classifier needs only to compare the number of bytes or ACK messages with a predefined threshold value, such as 1MB, to classify a flow as an elephant.

There are two methods to capture and notify the collector for every ACK message: (1) configuring the edge switches to mirror every ACK message to the collector. (2) using sFlow to sample every packet. Both methods have some drawbacks. The former method has a high overhead. Particularly, when all the switches are configured to mirror every ACK message to the collector. The estimated average of ACK packets is about 0.8K for a 1MB flow. On the other hand, the latter method captures ACK and non-ACK packets, where the collector needs only the ACK packets to detect the elephant flow. To get the best of both methods, we combine their advantages. We benefit from the latter method in reducing the overhead of the former method, as explained in Fig. 1 and we add additional overhead mitigation techniques, explained in the next subsection.

C. Minimizing the Network Overhead

Although FLight does not need to mirror all the packets to the controlling unit, the expected number of ACK messages of elephant-flow is crucial. The propagation of such volume of control messages through the data network will introduce a high queuing delay. The challenge is how to reap the benefits of the proposed method without overwhelming the data network. Before we explain our mechanism to reduce the network overhead of ACK messages, we need to demonstrate the feasibility of our fundamental idea. Hence, we use two kinds of datasets; enterprise and DCN datasets. We use three packet traces from LBL [15] (Inter) and four packet traces from the datasets of the university data center (Univ 2) that has been used by Benson [13]. In general, the LBL datasets has packet traces of 8,000 internal hosts and 47,000 remote hosts from different locations on the Internet. In Table. I, we ordered the collections sequentially, where the sequence number is a suffix added to the name of the datasets. The table shows that capturing the low and high frequency packets (i.e., normal FLight which is FLight without the overhead minimization techniques) has a low overhead compared to the naive solution (i.e., capturing all the packets) and better than packet sampling with a rate of 1-to-1000 in most of DCN datasets. For further reduction, we introduced two methods: Samples Only ACKs Packets and Stop Useless Notifications.

TABLE I:
The impact of normal FLight on real packet traces.

Dataset(ID)	Total packets	# of flows	Normal FLight	Ratio
Inter. 1	5700526	662	2313433	=2.5×
Inter. 2	2261261	3512	2174151	=1.04×
Inter. 3	84574	172	82817	=1.02×
Univ. 2.0	11917501	589	6333	=1881.8×
Univ. 2.1	11917527	775	14636	=814.3×
Univ. 2.3	11915859	727	7787	=1530×
Univ. 2.4	11915613	844	10528	=1131.8×

1) Samples Only ACKs Packets: The problem of using a packet-sampling technique is that the sFlow captures ACK and non-ACK packets. In this case, the number of captured packets will be similar to the number of capturing every TCP packet. In order to overcome this challenge, FLight steers the ACK packet to particular interfaces, and the sFlow in the edge switches are configured to sample only from these interfaces as illustrated in Fig. 1. These interfaces could be a high priority queue or a virtual interface associated with every interface in the edge switches. We focused on edge switches to reduce the overhead of capturing multiple packets of the same flow during a short period.

2) Stop Useless Notifications: The useless notifications are the messages that lost their value for the flow classifier. For instance, the flows that can be classified from their packet headers, such as the TCP port numbers, as well as the ACK messages of the classified flows. In FLight the classifier receives the packets of the low-frequency phase that have the packet header information. Thereby, it can classify some of the flows without receiving the packets of the second phase. It is widely known that the packet header information is enough to classify a flow, such as NTP, ICMP, and syslog. The edge switches can be preconfigured to forward the packets of such applications without sending notification messages. Moreover, in FLight when the classifier detects an elephant flow, it triggers the controlling unit to configure the edge switches to stop sending its ACK messages. In OpenFlow, for instance, the controller is responsible for performing such action and installs the necessary flow-entries with higher priority on the edge switches. In this case, the classifier will not receive the notification messages about classified flows.

III. Evaluation

We now present the evaluation of our proposed solution with respect to its ease of implementation in a real environment, accuracy, speed, and low detection overhead. In order to show the capabilities of FLight in a real environment, we run FLight directly on DCN as well as enterprise datasets. This will maintain the network characteristics as they are where these datasets were collected. Thereby, the performance figures of FLight are influenced by the network configurations (e.g., QoS settings) and the links loads of the datasets original networks. Although the coding of such evaluation is complex, it demonstrates the performance potential of our algorithm. Doing so, we are able to evaluate the detection overhead and speed. In order to further evaluate the accuracy, as well as detection overhead and speed, we built a DCN topology by using the techniques explained in the next subsection.

The following subsection shows the classification accuracy of FLight during different traffic loads and communication scenarios. Moreover, the speed of FLight in the detection of elephant-flows as well as its overhead on the controller are shown in the detection speed and detection overhead subsections.

A. Network Setup

We conducted our evaluation using Mininet emulator [16] and POX [17] controller. Minient uses the hardware resources and operating system of the host machine to build the configured network topologies including its switches, end-hosts, and links. The end-hosts are real virtual hosts using the underlay OS. The switches are real software switches (OvS [18]). Our centralized unit applications, implemented as an additional module on POX controller. Although POX controller isn't the most powerful controller, and we used it as a benchmark where any solution demonstrates its capabilities (accuracy, speed, and low overhead of detection) with POX, its performance is expected to be better with other more powerful controllers. The controller, as well as the Mininet topology, are installed on the same physical machine. The speed of links is 1Gbps. The OpenSample-TCP and sFlow are configured with 1-to-1000 sampling-rate, where the virtual interface in FLight is configured with sFlow (1-to-100) sampling-rate.

The system characteristics of the used machine are Ubuntu 14.04 LTS installed on 16 x (2.5GHz-Intel Xeon CPU E5-2680v3), and the memory is 128 GiB. We use Mininet to build a leaf-spine topology that has twelve leafs and six spines where each leaf has 50 hosts. We steered the traffic between the first and second subnet on a single path to make sure that the number of flows on that path is as needed, (i.e., 100, 500, and 1K). The flow arrives according to an exponential distribution with a mean of 2 milliseconds. We use Iperf, a commonly used network performance tool, to generate the flows. During the flow mix evaluations, the percentage of mice is 90% of the total flows. The size of mice and elephant-flows are 100KB, and 128MB, respectively, unless otherwise stated. We use different elephant-flow threshold values for the evaluation of FLight, however, the posted results are for 1MB, and the threshold value of the number of ACK is 15. The ACK threshold value is used during the real traces evaluation.

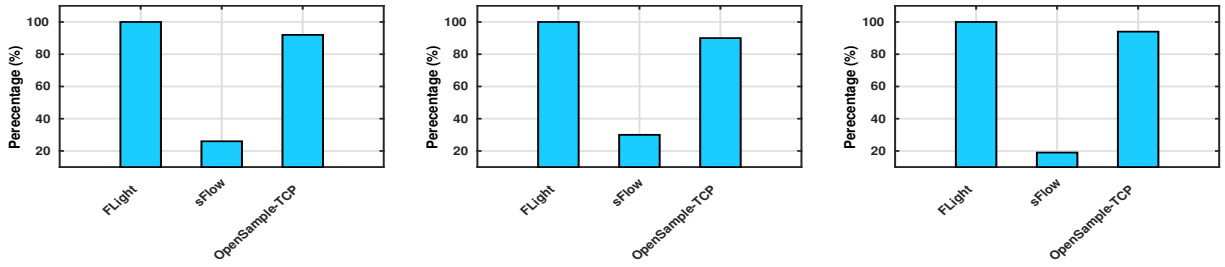
B. Detection Accuracy

The elephant flow detection algorithms may have true-negatives, (i.e., elephant flow is considered a mouse flow), false-positives, (i.e., considering a mouse flow as an elephant). In network load balancers, the reporting of few mouse flows as elephants are somewhat acceptable. However, the percentage of true-negative incidents is critical, where keeping the elephant flow on the path of mice flows causes a sever queuing delay for delay-sensitive flows. In this evaluation, we measure the percentage of true-negative incidents of FLight, OpenSample-TCP, and sFlow during different traffic loads and communication scenarios. In order to extract the results of the true-negative evaluation, we coded the Mininet to record the information of every generated flow in a statistics file and compared them with the algorithm logs.

We investigated FLight with different flow scenarios; pure mice flows, pure elephant-flows as well as a mix of mice and elephant-flows. First, we started with the evaluation of FLight and other solutions with the mix scenario. Fig. 2a illustrates the results when 100 flows are exchanged, where the accuracy of detected flows is 100%, which means the percentage of true-negative incidents is 0%, in the various testing scenarios. The solution is also evaluated during the exchange of 500, and 1000 flows, where the resultant accuracies are 100%, as displayed in Fig. 2b and Fig. 2c, respectively. The high accuracy of FLight is due to its simplicity, as we only needed to subtract the ACK sequence number of the ACK messages from the initial sequence number.

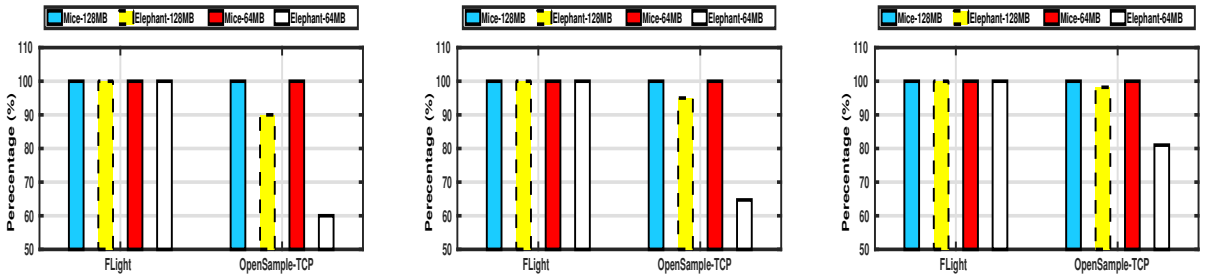
Next, we use the same setup and configurations to examine the accuracy of sFlow and OpenSample-TCP as elephant-flow detection algorithms, where the sampling-rate was 1-to-1000. Also, we follow their suggestion in the setting of the elephant-flow threshold values. Unfortunately, the accuracy of detection was varying as we adjust the threshold value settings, but we selected the best one. The results of sFlow and OpenSample-TCP evaluations are shown in the same figures above. In general, compared to FLight sFlow demonstrated high accuracy, above 90%, when it was examined with 100 and 500 flow loads and about 70% in 1000 evaluation. Similarly, the OpenSample-TCP showed above 90% accuracy in all experiments.

On the other hand, due to the space limitation and the low accuracy of sFlow we focused on OpenSample-TCP in the pure mice and elephant flows experiments. Although, we used a powerful machine we found Mininet crashes when we transmit 1000 elephant flows. Such errors have serious impacts on the accuracy of our evaluations. Due to this, we evaluated the two solutions during 10, 100 and 500 flows. Similar to previous experiments, FLight achieved a high level of accuracy in both pure mice and elephant flows as well. Also, OpenSample-TCP attained 100% in pure mice evaluation and 90%-98% in elephant flows experiment. We found that the accuracy, (i.e., the percentage of true-negatives), increases with the increase in traffic load. This event is due to the long transmission delay of elephant flows during high congestions. In order to examine both schemes in a short transmission delay and high traffic loads, we use a small file size, 64MB. Performing the 10, 100, and 500 experiments with 64MB on the OpenSample-TCP algorithm, we got a high percentage of true-negative incidents; 40%, 35.3%, and 19% respectively, which in turn is corresponding to 60%, 64.7% and 81% accuracy. On the same evaluation basis, the FLight solution achieved 0% of true-negative incidents. The results are displayed in Fig. 3.



(a) When 100 flows are transmitted. (b) When 500 flows are transmitted. (c) When 1000 flows are transmitted.

Fig. 2: The evaluation of the accuracy of FLight in detection of elephant-flows in different traffic configurations and during mix scenario.



(a) When 100 flows are transmitted. (b) When 500 flows are transmitted. (c) When 1000 flows are transmitted.

Fig. 3: The evaluation of the accuracy of FLight in detection of elephant-flows in different traffic configurations and during pure mice and elephant scenarios.

C. Detection Speed

After we demonstrate the high accuracy of FLight in flow classification, we examine the speed of elephant-flow detection. In order to investigate the speed of detection, we examined FLight with different scenarios of network loads; 10, 100, or 200 flows. Also, we run our algorithm on the enterprise measurement datasets. The detection time is extracted from a timer function on the classifier code. However, the results will be faster if it is extracted from the network interface of the controlling unit either by using a software or hardware `timestamp`.

The average speeds of detection during the different traffic loads and configurations are illustrated in Fig. 4, which range from 4 to 48 milliseconds according to the load on the path. FLight is 242× to 20.3× faster than OpenSample-TCP. The detection of elephant flows is slower than measuring the link utilization and this is because the elephant flow detection algorithm needs to wait until the size difference between the first and second captured packets is larger than the elephant flow threshold value. On the other hand, to measure the performance of our algorithms with different network configurations and loads, we run them on the enterprise measurement datasets. We found the speed of detection ranging from 10 to 300 milliseconds, as shown in Fig. 5. When we further investigate the datasets, we found the Inter. 2 dataset has high percentages of elephant-flows up to 77%, the highest among other datasets.

D. Detection Overhead

In this section, we measure the overhead of FLight on the centralized unit as well as on the network. We use the same network setup and topologies of above sections, however, the centralized unit is installed in a virtual machine with four processors and 10GB memory. In order to collect the statistics (e.g., CPU and memory utilization) of the centralized unit we used a `psutil` python library [19], and we configured `top` Linux tool to collect the device statistics for every second and store them on a particular file. The impact of FLight and other solutions on memory were less than 1%, and for this reason we did not include them in the figure. In all the evaluated solutions, the centralized unit needs to setup the network and prepare the OvS switches with necessary configurations as well as sFlow rules, defined as startup phase. This startup phase, in all the evaluated solutions, consumes about 100% of a single CPU processing capacity and installs about 7KB of flow-entries. Since FLight configures sFlow in the virtual interfaces with a fast sampling rate, we expect the communication, as well as the processing overhead of FLight, to be worse than the others. However, we found the communication overhead of FLight is about 70KB while it is 63KB for OpenSample-TCP with 1-to-1k sampling rate. However, when we used a larger flow size (e.g., 256MB), the communication overhead of FLight remained close to 70KB while it increased dramatically for OpenSample-TCP. The OpenSample-TCP continues the collection of samples for all of the flows regardless whether if the flow has been classified or not. Thereby, the communication overhead, as well as the processing capacity, have not been improved after the classification, and the total communication overhead has increased with the increase in the size of the transmitted file. In contrast, FLight has the overhead minimization techniques that capture only the ACK packets of the unclassified flows. On the other hand, due to its simplicity and faster detection, the computational overhead of FLight is 5× lower than the OpenSample-TCP, and close

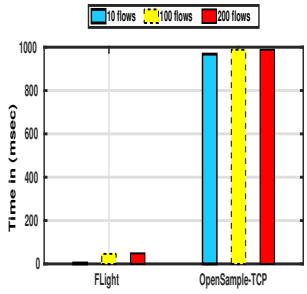


Fig. 4: FLight speed on leaf-spine topology.

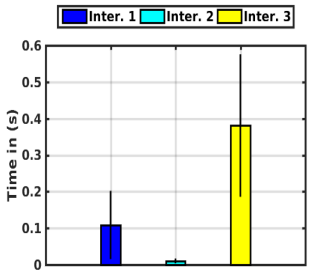


Fig. 5: FLight speed on the enterprise datasets.

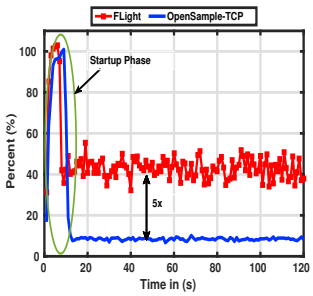


Fig. 6: The processing overhead during the first 120 sec.

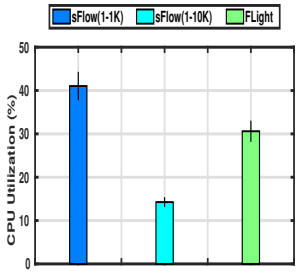


Fig. 7: Avg. processing overhead of FLight vs sFlow.

TABLE II:
The arrival-rate of FLight ACK messages.

Dataset(ID)	Dataset ACK/Sec	FLight ACK/Sec	Ratio	Dataset(ID)	Dataset ACK/Sec	FLight ACK/Sec	Ratio
Inter. 1	3854.2	62.6	61.6x	Univ. 2.0	5.1	3.33	1.5x
Inter. 2	3615	62.5	57.84x	Univ. 2.1	11.9	0.798	15.1x
Inter. 3	137.5	4.15	33.13x	Univ. 2.3	6.892	0.19	36.3x

to sFlow (1-to-10k). The results are illustrated in Fig. 6 and Fig. 7. These statistics were collected during the transmission of 1k mix-flows and 1.5k mix-flows, respectively. When we run our overhead minimization algorithm on the enterprise and DCN datasets, the overhead of ACK messages reduced to minimal values compared to the ACK arrival-rate of the datasets themselves. The results are shown in Table. II, ranging from 1.5× up to 61.6×.

IV. Conclusions

Previous researchers in Internet and cloud network traffic management has demonstrated the significance of detecting and scheduling of elephant-flows. However, recent methods of elephant-flow detection have some technical limitations, such as high monitoring overhead, long detection time, and high percentages of true-negative incidents. Besides, those may not be suitable for networks with a mixture of traffic sources (i.e., modifiable and non-modifiable hosts) such as cloud DCN networks. This kind of network needs a mechanism that has rapid detection of elephant flows but with little processing overhead and also without modifying end hosts network stacks. Thus, we propose FLight, a novel and flexible elephant-flow detection scheme that uses the TCP communication behavior. Our experimental results show that FLight can detect elephant-flows in a milliseconds timescale with full accuracy. FLight uses Stop Useless Notification technique and Samples Only ACK Packets to reduce its overhead, which results in computational and network overhead lower than sFlow (1-to-1K).

References

[1] M. Al-Fares et al., "Hedera: Dynamic flow scheduling for data center networks." in *NSDI*, vol. 10, 2010, pp. 19–19.

[2] H. Zhang et al., "Resilient datacenter load balancing in the wild," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17, 2017, pp. 253–266.

[3] A. R. Curtis et al., "Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection," in *INFOCOM, 2011 Proceedings IEEE*, April 2011, pp. 1629–1637.

[4] V. Sivaraman et al., "Heavy-hitter detection entirely in the data plane," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '17, 2017, pp. 164–176.

[5] J. C. Mogul and P. Congdon, "Hey, you darned counters!: get off my asic!" in *Proceedings of the first workshop on Hot topics in software defined networks*. ACM, 2012, pp. 25–30.

[6] M. Yu et al., "Software defined traffic measurement with opensketch." in *NSDI*, vol. 13, 2013, pp. 29–42.

[7] J. Suh et al., "Opensample: A low-latency, sampling-based measurement platform for commodity sdn," in *Proceedings of the 2014 IEEE 34th International Conference on Distributed Computing Systems*, ser. ICDCS '14, 2014, pp. 228–237.

[8] J. Rasley et al., "Planck: Millisecond-scale monitoring and control for commodity networks," in *Proceedings of the 2014 ACM Conference on SIGCOMM*, ser. SIGCOMM '14, 2014, pp. 407–418.

[9] Y. Zhu et al., "Packet-level telemetry in large datacenter networks," in *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4. ACM, 2015, pp. 479–491.

[10] Y. Li et al., "Flowradar: A better netflow for data centers." in *NSDI*, 2016, pp. 311–324.

[11] M. integrated hybrid OpenFlow testbed. [Online]. Available: http://blog.sflow.com/2014/04/mininet-integrated-hybrid-openflow.html

[12] K.-c. Lan and J. Heidemann, "A measurement study of correlations of internet flow characteristics," *Computer Networks*, vol. 50, no. 1, pp. 46–62, 2006.

[13] T. Benson et al., "Understanding data center traffic characteristics," *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 1, pp. 92–99, 2010.

[14] M. Alizadeh et al., "Data center tcp (dctcp)," in *ACM SIGCOMM computer communication review*, vol. 40, no. 4. ACM, 2010, pp. 63–74.

[15] R. Pang et al., "A first look at modern enterprise traffic," in *Proceedings of the 5th ACM SIGCOMM Conference on Internet Measurement*, ser. IMC '05, 2005, pp. 2–2.

[16] N. Handigol et al., "Reproducible network experiments using container-based emulation," in *Proceedings of the 8th international conference on Emerging networking experiments and technologies*. ACM, 2012, pp. 253–264.

[17] POX. [Online]. Available: http://www.noxrepo.org/pox/about-pox/.

[18] B. Pfaff et al., "Extending networking into the virtualization layer." in *Hotnets*, 2009.

[19] psutil cross-platform process and system monitoring module for Python. [Online]. Available: http://code.google.com/p/psutil/.