Empowering Software Defined Network Controller with Packet-Level Information

Sajad Shirali-Shahreza, Yashar Ganjali
Department of Computer Science, University of Toronto, Toronto, Canada

***Abstract*—Packet level information, such as packet content and inter-arrival time, are necessary for some network monitoring and control applications. However, current Software Defined Networks (SDN) such as OpenFlow provide limited access to packet-level information in the controller. In this paper, we propose an extension that enables the controller to access packet-level information through per-flow sampling. Our extension is flexible and powerful, yet it can be implemented entirely in the data plane at line rate. We present a set of possible applications that can take advantage of this new packet-level information, including examples that are extremely difficult, if not impossible in current SDN.**

I. Introduction

A group of network control and monitoring services need packet-level information to operate: security applications such as Intrusion Detection Systems (IDS) rely on the content and ordering of packets to detect different types of attacks; traffic classification schemes require packet contents, or features that can be extracted using packet level information only.

Where should we implement services that require packet-level information in a Software Defined Network (SDN)? The first option is to push them to the edge of the network, keeping the network simple and dedicated to routing (e.g. in Network Fabric proposal [1]). The second alternative is to design and deploy middleboxes providing the service throughout the network. The third option is to add the functionality to existing switches.

Each of these alternatives has its own pros and cons. Although the first option keeps the network simple, it is only applicable to specific types of networks such as datacenters where we can change the edge of the network. Middleboxes can be easily added to a network, but they can be costly, need coordination with the rest of the network, and thus can make network management more complex [2]. Modifying switches means there are fewer devices in the network, and we can have great performance. However, switch modifications can be extremely difficult, as it might need changes to the hardware.

If we had access to packet-level information in the SDN controller, a fourth potential alternative would have been to implement such services as controller applications. This is very flexible and low-cost in terms of deployment and updates, and makes coordination among different services and with the controller very simple. Unfortunately, current SDN controllers have limited access to packet-level information. Controllers in current SDN designs (such as OpenFlow) either see a small subset of all packets, or can be configured to receive all packets of a flow, which can lead to very high loads towards the controller.

In this paper, we propose to add a new information channel for the controller to access packet-level information. Our solution consists of a per-flow sampling feature that is generic and flexible for different applications, simple to implement completely in data-plane and operate at line rate, and completely definable by the controller to balance the overhead and information detail.

We also provide a list of example application that can use this per-flow sampling. Per-flow sampling can enable the controller to access different packet-level information and implement applications that depends on them: sampling a few consecutive packets gives access to packet inter-arrival times, e.g. to detect bursty traffic or a port scan attack, and sampling full packets make packet contents accessible for the controller, e.g. to perform traffic classification or detect worm and virus spread in the network.

Note that we are *not* proposing a new sampling method and do *not* claim that per-flow sampling is the best information channel for all applications that need packet-level information. Sampling can have a high overhead. However, the sampling overhead in our proposal can be controlled in real-time by the controller. The controller can dynamically adjust the sampling rate over time, and can use different rates for various parts of the network. For example, an IDS application can start with a very small sampling rate with low controller overhead. Upon observing suspicious behavior, the application can increase the rate for specific parts of the network acquiring more information about traffic without significant changes in the overhead [3].

As a potential application, we provide implementation details and simulation results of using per-flow sampling to detect elephant flows in a network. We show how the controller can sample only a small fraction to keep the sampling overhead low, while generating results which are comparable to existing solutions that require switch modifications.

II. Per-Flow Sampling for OpenFlow

There are three major information channels for the controller in the current OpenFlow – the well-known example of SDN – specification:

1) *Event-based Messages*. Event-based messages that are sent by the switches, such as the state change of a link or port, usually deliver information about changes in network structure and topology, and do not provide any packet-level information.

2) *Packet-in Messages*. A switch may send a packet-in message either because it did not know what to do with the packet – no matching entry found in the flow table – or as a result of *send-to-controller* action matching flow entry. The

switch may buffers the original packet and only includes part of the packet – usually the first 128 bytes – in the packet-in message. A table-miss packet-in message represents a newly started flow, which means that it is the first packet of the flow, and usually used for flow setup. Considering that the majority of network traffic is TCP, the first packet of the flow is TCPSYN. So even though the controller may receive the full first packet – if it is not buffered in the switch – the packet content is not containing any useful information.

3) *Flow Statistics*. Flow statistics that are collected by the switches (received packets, received bytes and duration in the current specification) are the only information channel for the controller to access to information about active flows.

Although these statistics provides valuable information about flows, they have some limitations. First, pooling statistics by controller imposes a considerable load on the switches and limits the rate they can be read by the controller [4].

Second, existing statistics provide aggregated information over a period of time. As a result, these statistics are not useful for short flows, a significant percentage of the flows found in datacenters [5]. Third, statistics collected from wildcard rules – which matches multiple flows – such as total bytes, might be less useful since viable information could be lost due to aggregation. For example in the elephant flow detection problem – identifying flows that carry high amount of traffic, e.g. more than 10% of link capacity – the controller can use these statistics to detect the rule that matches an elephant flow, but it cannot infer any other information, such as source and destination of the elephant flow, based on this information. This is the main reason that methods such as Hedera [6] and DevoFlow [4] introduce new local counters in switches to be able to detect elephant flows in the switch and then inform the controller.

As we see, the controller has a limited access to packet-level information, mainly through aggregated switch statistics. The only possible way in current specification to access information is asking the switches to send a copy of every packet – using a send-to-controller action – to the controller, which has huge overhead on the network and controller. This can be a limiting factor for the controller which make it extremely hard, if not impossible, to implement applications such as traffic classification, intrusion detection, DDoS prevention and other security applications that heavily depend on the payload of individual flows. Another approach to obtain required information and implement these applications is installing middleboxes in the network, which has its own problems [2], and is outside the scope of this paper.

Sampling is a well-known concept in network monitoring and control. Uniform sampling is relatively easy to implement and has low overhead for switches. sFlow is an uniform sampling methods that is implemented in many switches. Per-flow sampling has many advantages over per-packet sampling such as generating more accurate estimation of traffic statistics [7]. It is also more useful for security applications (such as intrusion detection systems (IDS)) that need data about short-lived flows, which can be missed by uniform sampling [8].

Implementing per-flow sampling in traditional networks and in solutions such as ProgME [9] requires significant changes to the hardware and packet processing flow in the switches. However, identifying the flow that the packet belongs to and keeping a record of the identified flows – a major task for per-flow sampling methods in traditional networks such as ProgME [9] – is an essential part of OpenFlow.

Our Proposed Information Channel. In this paper, we propose to add per-flow sampling as a new information channel for the controller to access packet-level information. Our goal is providing a sampling feature that is general enough for different applications, simple to implement completely in data-plane and operate at line rate, and completely definable by the controller to balance the overhead and information detail.

To reach generality goal, we include two types of sampling: (1) select each packet of the flow with a probability of ρ, and (2) select m consecutive packets from each k consecutive packets, skipping the first δ packets. The first case is the stochastic sampling, which is relatively straightforward to implement: for each packet, a random number is generated. If it is less than ρ, the packet is sent to the controller.

The second case is a generalized version of the deterministic sampling. For $m=1$, it is equivalent to the normal one out of k, or every k^{th} packet sampling. If an application needs more than one consecutive sample, it can set m to a value more than one. By choosing a very large k, an application can ensure it will only receive the first m consecutive packets. This is usually what security applications such as intrusion detection need. Finally, by changing the value of δ, the application can skip the first few packets of each flow. This can have a significant impact on the number of sampled packets, by excluding small and short flows (mice flows).

OpenFlow switches maintain a number of counters for each flow. One of them is the *Received Packets* counter, which counts the number of packets received for each flow. Deterministic sampling of m first packets from each k packets after ignoring δ packets could be done using this counter: if $((Received\_Packet\_Counter-\delta)\ \%\ k)<m$, then the packet will be sampled. This simple expression could be executed in the data path at line rate without need to any new memory for sampling.

Considering that sending full packets could impose a significant load on the network, and not all applications need full packet contents, we let the controller decide what parts of the sampled packets should be sent (e.g. IP header only).

Although there are previous works in the SDN domain that uses sampling, they either use uniform sampling methods like sFlow, such as DevoFlow [4], or use statistics that are gathered by OpenFlow switches – e.g. received packet count – to replace the need to sampling techniques like NetFlow, such as Jose et al. work [10]. The main advantages of our method are:

1) Easy switch implementation. The OpenFlow switch needs to identify the flow that each packet belongs to process it, thus the per-flow sampling can be added easily.

2) Low switch overhead. Our proposed per-flow sampling can be implemented without any additional counters and in constant time ($O(1)$) that could be run at line rate.

3) Access packet contents. Our new information channel enables the controller to gather information that is not accessible in current OpenFlow specification such as packet content and inter-arrival times.

4) Controllable network overhead. The controller can tune the sampling rate to balance the overhead vs. details. By increasing the sampling rate, the controller can collect more information and visibility, at the cost of higher overhead.

III. Possible Applications

Security Applications: Security applications (e.g. IDS [3]) usually need to examine packet contents. If the controller were doubtful about some flows, it could ask the switch to send sample packets so that it can analyze their content. Similar to current network IDS systems like [3], it could change the sampling rate before/after detecting a suspicious activity.

Traffic Classification: The controller could analyze the data payload of sampled packets to determine the traffic type of flow to provide Quality of Service or block certain type of traffics. For example, it could reserve some low-load routes for time-sensitive flows such as VOIP, then detect and reroute VOIP traffic through them.

Quality of Service: The controller could use the arrival time of sampled packets to deduce the overall behavior of the flow for QoS. For example, it could distinguish between constant rate flows and flows that send a burst of data and then wait. These fine-grained traffic properties, that "have a significant impact on effectiveness of switching mechanisms and traffic engineering" [11], could be used to predict possible congestion or provide better QoS for delay-sensitive or loss-sensitive flows.

Diagnostics: Sampling can help the network administrator become aware of what is happening in the switch. In the extreme case of sampling with a rate of 1/1, a copy of each packet would be sent to the controller. The operator could remotely see every packet – or subsets of packets – that passes through a switch to determine possible causes of a problem. A selective sampling approach can control the overhead, e.g. only sample packets on the switch that has problem, or packets from a specific application or machine that may cause the problem.

To better demonstrate how our proposed per-flow sampling can be used to solve these problems, we present a solution for the elephant flow detection problem in the next section. This is a well-known problem, both in traditional networks and SDN, and different solutions are proposed for it, including Hedera [6] and DevoFlow [4] for the OpenFlow networks. We should restate that we do *not* aim to present a better solution for the elephant flow detection problem. Instead, we want to show how our approach provides a simpler way to solve that problem, with comparable, if not better, performance.

IV. Elephant Flow Detection

The flow setup delay is one of well-known problems of reactive routing in OpenFlow: the first packet of the flow is kept in the first switch until the controller receives the "*packet-in*" message from the switch, processes it, *e.g.* calculates the routing path, and replies back with instructions on how to handle that packet, *e.g.* by creating a new flow entry. This delay could significantly impact short-lived flows in networks such as datacenters.

One way to solve this problem is starting with proactive routing – install a set of general (wild-carded) rules to route – and then identifying elephant flows (from all other flows that match a wild-carded-rule) to re-route them optimally. This approach requires elephant flow detection, which is the problem that solutions such as Hedera [6] and DevoFlow [4] aim to solve. As an example application of our proposed per-flow sampling extension to OpenFlow, we show how the controller can use per-flow sampling to detect elephant flows.

Previous solutions such as Hedera [6] and DevoFlow [4] move the task of identifying elephant flows from the controller to the switches because current controllers have no clue about different flows that match a wild-carded flow rule: (1) switches keep track of individual flows that match a wild-carded rule, referred to as *microflows* in DevoFlow [4], (2) identify elephant flows, and (3) notify the controller once an elephant flow is detected. These steps all require considerable modifications to switches [4], which is antithetical to OpenFlow's goal of having simple and future-proof switches, and makes modifications to applications difficult by creating a tie with the underlying hardware. Additionally, switch-based solutions that require modifications to hardware – such as the required ASIC changes for DevoFlow [4] – are usually very costly, and make future changes difficult.

In our method, when the controller is installing wild-carded rules, it also asks switches to send sampled packets – with a probability of ρ – from the flows that match them. The controller processes the sampled packets received from the switches, and detects elephant flows.

The idea of our elephant flow detection method is simple: the controller uses the number of sampled packets it receives to estimate the combined rate of flows that match a wild-carded rule and decide whether there is an elephant flow among them. Once the controller realizes there is an elephant flow, it processes the content of sampled packets to identify the elephant flow.

When the sample arrives in the controller, the first task is detecting whether an elephant flow started or not. Here, we consider a flow to be an elephant flow if its rate is higher than a threshold, such as 10% of the host's link bandwidth used in Hedera [6]. To detect elephant flows, we estimate the throughput of traffic that matches each installed flow by defining a time frame – e.g., the last 1 second – and counting the number of samples we received in that interval. If this number exceeds the detection threshold, it is possible that an elephant flow exists and matches this flow entry. Assume that host links are 1 Gb, packet size is 1500 bytes, and sampling probability (ρ) is 0.001. An elephant flow with more than 10% bandwidth will send more than $10^9/(1500\times8)\times0.1\cong8333$ packets per second, from which 8 packets, on average, will be sampled and sent to the controller. So the controller will see at least 8 packets/second for each elephant flow. This is the detection threshold that the controller will use.

```
SampledPacketProcess (packet, cookie)
1.  current_time = GetTime()
2.  oldest_keep = current_time - sampling_time_frame
3.  while not(queue.IsEmpty()) and queue.top.time > oldest_keep
4.      counters[queue.top.cookie]--
5.      queue.PopTop()
6.  queue.Append(current_time, cookie, packet)
7.  if cookie in counters:    counters[cookie]++
8.  else:    counters[cookie] = 1
9.  if counters[cookie] >= detecting_threshold:
10.     FindElephantFlow(cookie)
FindElephantFlow(cookie)
1.  flow_packets = queue.GetPackets(cookie)
2.  elephant_flow = null
3.  foreach packet in flow_packets:
4.      candidate_flow = ExtractFlow(packet);    matched = 0
5.      foreach test_packet in flow_packets:
6.          if MatchFlow(candidate_flow, test_packet):  matched++
7.      if matched > detecting_threshold:
8.          elephant_flow = candidate_flow;    break
9.  if elephant_flow ≠ null: ProcessElephantFlow(elephant_flow)
```

Figure 1. Pseudocode for elephant flow detection in controller.

Figure 1 shows our algorithm for detecting elephant flows. The algorithm is designed to minimize the set of actions done on each sampled packet arrival so that it can scale well. It has a *queue* that contains packets that arrived during the past *sampling_time_frame* seconds. It also has *counters* array, which keeps the number of packets in the *queue* that match each installed rule. Upon each sampled packet arrival, the *SampledPacketProcess* function is run (Figure 1). The *cookie* parameter refers to the matching flow for this packet. When a new packet arrives, the first task is to remove expired packets (*i.e.*, packets that are more than *sampling_time_frame* seconds old) from the *queue*. The second task is adding the new packet to the queue and incrementing the packet counters for this flow. Finally, it checks to see if the number of samples from this flow during the defined timeframe exceeds the threshold. In that case, it calls the *FindElephantFlow* function to find the elephant flow. The amortized cost of this function for each packet is *O(1)*.

The *FindElephantFlow* function determines whether an elephant flow matching flow rule identified by *cookie* exists, and, if so, calls the *ProcessElephantFlow* function. This latter function performs required actions for elephant flows, similar to that in Hedera [6] and DevoFlow [4]: finding the best route and installing necessary rules. This function acts similarly

Figure 1 contains a simple way to implement *FindElephantFlow* function. It simply iterates through all sampled packets from *cookie* flow, extracts an exact match candidate flow from it, and then counts how many other packets match them. If it matches more than the defined threshold of sampled packets, it considers that to be an elephant flow and calls *ProcessElephantFlow*.

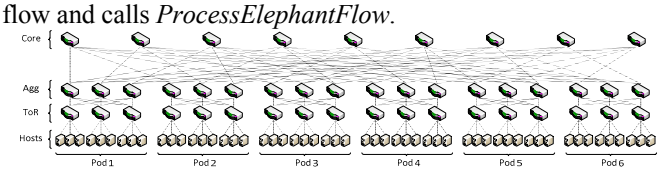


Figure 2. Fat-tree topology used for simulation.

Implementation. We added per-flow sampling to the reference OpenFlow switch implementation and OpenvSwitch (http://openvswitch.org). We also added per-flow sampling to the NOX controller (http://noxrepo.org) and implemented our elephant flow detection as a NOX application. Considering that we designed our sampling to be implementable with minimal modifications, the changes that we made to these systems were small: adding 149 lines of C code to the reference OpenFlow switch implementation, 450 lines of C code to the Open vSwitch, and 186 lines of C code and 19 lines of Python code to NOX to implement the per-flow sampling feature.

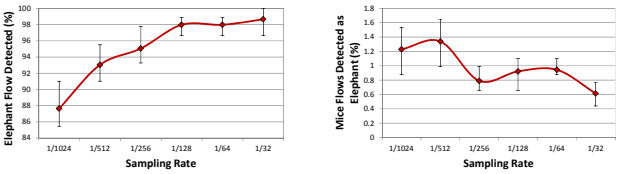
In conducted stress tests – like iperf –we did not see any noticeable difference between the performance before/after adding the per-flow sampling feature to the switches or for the rules with/without sampling. We used the Mininet simulation environment to simulate the network and evaluate our system.

Topology. We used the common fat-tree topology shown in Figure 2 for our simulation. The whole network was divided into *2k pods*, each containing $4k^2$ hosts in k racks. There were two levels of switches in each pod: *Top-of-Rack* (*ToR*) switches are connected to the hosts and *aggregation* switches that are connected to ToR switches and *core* switches. There were $4k^2$ core switches that connect different pods together. One of the features of this topology is multiple equal cost paths between each pair of hosts. For example, there are k different routes between a pair of hosts in the same pod but different racks, and k^2 different routes between a pair of hosts in different pods. A similar topology was used in previous work, such as Hedera [6] and DevoFlow [4]. Due to the limitations of Mininet for simulation – such as a total of 2Gbps total bandwidth and the requirement of running all switches and hosts in the same physical machine – we performed our simulation for k=3, which contains 54 hosts and 45 switches.

Workload. We created our workload from the flow characteristics of data-center networks provided in Kandula et al. [12]. We used their flow rate, flow duration and flow inter-arrival times and generated traffic based on them. Considering that the size of our network was one order of magnitude smaller than the network described there, we multiplied the inter-arrival times by 10 to prevent too many active flows at each host, which could skew the results. We generated 1000 flows for a 200 seconds simulation. The same set of flows with the same timing was used for different runs to provide a fair comparison among them. We used MGEN toolkit (http://cs.itd.nrl.navy.mil/work/mgen/) to generate host's traffic.

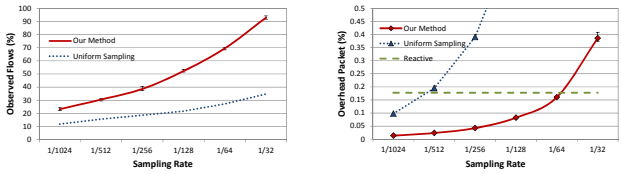
Different Scenarios. Our main goal is to compare the overhead of our method with the reactive approach and to evaluate the effect of different sampling rates. We used 6 different sampling rates for stochastic sampling: 1/1024 (i.e. 1 out of 1024), 1/512, 1/256, 1/128, 1/64, and 1/32.

During the initial setup, the controller installs a set of flows on each switch. In all cases, the controller installs necessary flow rules to route broadcast messages throughout the network. For our elephant flow detection scenarios, it installs a set of default rules to route traffic between hosts.



(a) True Positive (b) False Positive

Figure 3. Elephant flow detection accuracy.



(a) Visibility (b) Overhead

Figure 4. Visibility and overhead of different approaches.

Each switch know the hosts in its sub-tree and forwards other packets to one of the switches in the upper layer (ToR→Agg, Agg→Core). When the controller installs a default routing rule on an aggregation switch, it included the sampling action with stochastic sampling of rate ρ and asks the switch to send the Ethernet+IP+TCP data from each sampled packet, which is the first 66 bytes of each packet. No sampling action was added to other flow rules. To compare our method to uniform per-packet sampling, we also included the results for uniform per-packet sampling with same sampling rates.

Detection Rate. We define elephant flows as flows that have a minimum rate of 12.8 Mbps for at least one second. Of the total 1000 flows, the rate of 89 flows exceeded 12.8 Mbps and should be detected as elephant flows. Figure 3a shows the elephant flows detection rate for different sampling rates. The detection rate is 98% for sampling rates greater than 1/128. Figure 3b shows the false detection rate, which is the percent of non-elephant, – mice – flows that were incorrectly marked as elephant flows. The false detection rate is less than 1% for sampling rates higher than 1/256.

Visibility. One of the main drawbacks of proactive approaches is that the controller has no knowledge of active flows in the network. When the controller receives a sample packet from a flow, it becomes aware of that flow (a bonus advantage of sampling). We define visibility as the percent of flows that the controller is aware of their existence, e.g. received at least one sampled packet from it. In previous elephant flow detection methods, the controller only becomes aware of elephant flows, which are 9% of the total flows. Figure 4a shows the visibility for different sampling rates and compares it to uniform per-packet sampling and also previous elephant flow detection method. With a sampling rate of 1/128, the controller is aware of more than half of the flows. The visibility of per-flow sampling is about 2.4 times of the uniform sampling visibility, which is consistent with previous work on per-flow sampling, e.g., Salem et al. [8], and 5 times of the visibility of previous elephant flow detection methods.

Overhead. Figure 4b shows the packet count overhead of different approaches. We note that only packets that do not belong to detected elephant flows are being sampled, which means that a rate of *1/r* does not directly translate to *1/r* of all traffic being forwarded to the controller. We see that the overhead of sampled packets is even less than the overhead of packet-in messages in reactive approach for sampling rates lower than 1/64. Additionally, the overhead of our per-flow sampling is significantly lower – about 1/10 – than of uniform per-packet sampling, mainly because of excluding elephant flows from sampling in our method.

V. Conclusion

Packet-level information, e.g. packet content and packet inter-arrival, time is essential to implement network control applications such as IDS or provide QoS. However, current OpenFlow specification provides a limited access to it. In this paper, we proposed a new information channel for the controller to access packet-level information via per-flow sampling. Our per-flow sampling extension for OpenFlow is simple enough to be performed in data path at line rate, yet general and flexible for different applications. It enables the controller to access information is not available in current state, such as access to packet contents.

We provided a list of possible applications that can be built using this per-flow sampling. We also presented the implementation and simulation results of elephant flow detection application as a proof-of-concept application uses this per-flow sampling feature. We plan to implement and evaluate other possible applications that could use per-flow sampling.

References

[1] Casado, M., et al. 2012. Fabric: A Restrospective on Evolving SDN. In *HotSDN 2012*.

[2] Sekar, V., et al. 2012. Design and implementation of a consolidated middlebox architecture. In *NSDI'12*. 323-336.

[3] Dreger, H., et al. 2004. Operational experiences with high-volume network intrusion detection. In *CCS '04*. 2-11.

[4] Curtis, A.R., et al. 2011. DevoFlow: scaling flow management for high-performance networks. In *SIGCOMM'11*. 254-265.

[5] Benson, T., et al. 2010. Network traffic characteristics of data centers in the wild. In *IMC '10*. 267-280.

[6] Al-Fares, M., et al. 2010. Hedera: dynamic flow scheduling for data center networks. In *NSDI'10*.

[7] Hohn, N., and Veitch, D. 2006. Inverting sampled traffic. *IEEE/ACM Trans. Netw.* 14(1). 68-80.

[8] Salem, O., et al. 2010. A scalable, efficient and informative approach for anomaly-based intrusion detection systems: theory and practice. *Int. J. Netw. Manag.* 20(5). 271-293.

[9] Yuan,L.,et al. 2011.ProgME:towards programmable network measurement. *IEEE/ACM Trans. Netw.* 19(1).115-128.

[10] Jose, L., Yu, M., and Rexford, J. 2011. Online measurement of large traffic aggregates on commodity switches. In *Hot-ICE'11*. Paper 13.

[11] Benson, T., et al. 2010. Understanding data center traffic characteristics. *SIGCOMM CCR*. 40(1). 92-99.

[12] Kandula, S., et al. 2009. The nature of data center traffic: measurements & analysis. In *IMC '09*. 202-208.