The detection of elephant flow in SDN

陈昊 5110309221

i. Abstract

Software defined network provides better network management and higher utilization, which decouple control plane from data plane. The centralized control of entire network may result in large overhead and limit the scalability of control plane. Previously, some researchers proposed approach for reducing work load on controller with elephant flow detection. However, the threshold of the detection system was pre-configured with a fixed value without considering the dynamically changing traffic characteristics, which would cause high detection error rate. So, we propose a two-stage adaptive elephant flow detection system. Also, we try to have a shorter time delay in detecting the elephant flow, so as to improve the efficiency of our system. We do experiment to test our system.

ii. Introduction

Software defined network (SDN) is an emerging network management paradigm that separates control plane from the underlying physical devices. There is a centralized controller to enforce the management of an entire network, in contrast to conventional network, where operators have to codify functionality by complicated configuration. As SDN provides a globally optimal management of network resources and a flow-level control of network traffic, the centralized network control has also been considered in high-performance networks, such as data center. OpenFlow is the first standard protocol designed specifically for SDN, which has been supported by multiple hardware vendors. This paper is based on OpenFlow protocol to realize the communication between control plane and data plane.

SDN provides much easier network configuration, higher utilization, and better network management for data center. However, there is a bottleneck that control plane is difficult to scale up to operate the rapidly changing traffic. First, the arrival and departure of flows are very fast in data center, while it will take 10ms for controller to allocate resource for every new flow. Secondly, it will need numerous controllers to meet the demands of data center because of the limited processing capacity of existing controller. Third, TCAM storage space of every flow entry installation is a valuable resource in switches which can only afford 1,500 wildcard rules in OpenFlow. Overall, the centralized control mechanism is only practical if it is able to scale up to meet the demands of the dynamic traffic in the data center.

Recently, some researchers propose elephant flow detection to improve the scalability of control plane. This approach can reduce work load on control plane and save the valuable TCAM resource in data plane. In measurements of 10 data centers, Benson et al. Research show that 80% of the flows are smaller than 10KB in size and the most bytes are carried in the top 10% of large flows. It is not necessary for controller to operate all flows and orchestrate their paths. For effective traffic management, controller need only concentrates on the significant (or elephant) flows that have impact on network performance. A large number of small (or mice) flows come and go too fast to wait the flow entries installation according to controller's policy. Thus, identifying elephant

flows is important to construct appropriate forwarding policy for various types of flows. However, existing detection systems are pre-configured with a fixed value without consideration of the dynamically changing traffic characteristics in data center, which will cause a lot of false positive errors and false negative errors. We present a two-stage, adaptive elephant detecting system for data center with SDN. The main contribution of this work includes:

- We propose the two-stage elephant detection system which provides more accurate identification of elephant flows. It reduces the load on controller which only handles the elephant flows. Thus, mice flows will be forwarded without delay.
- We design a dynamic adaptive decision threshold for this detection system. By analyzing the relationship between elephant threshold and traffic characteristic in data center network, there is an optimal point that could balance the positive false rate and negative false rate of detection. Inspired by the optimal receive system of baseband signal transmission through the additional Gauss White Noise channel, we solve this optimization problem by computing the intersection point of the two flow probability distribution curves.
- We evaluated the performance of this detection system on real trace of data center. Numerical experiments and validation results showed that it could significantly reduce the load pressure on control plane. The overhead and delay caused by this system is only a small portion of the typical long-lived, high-throughput elephant flows.

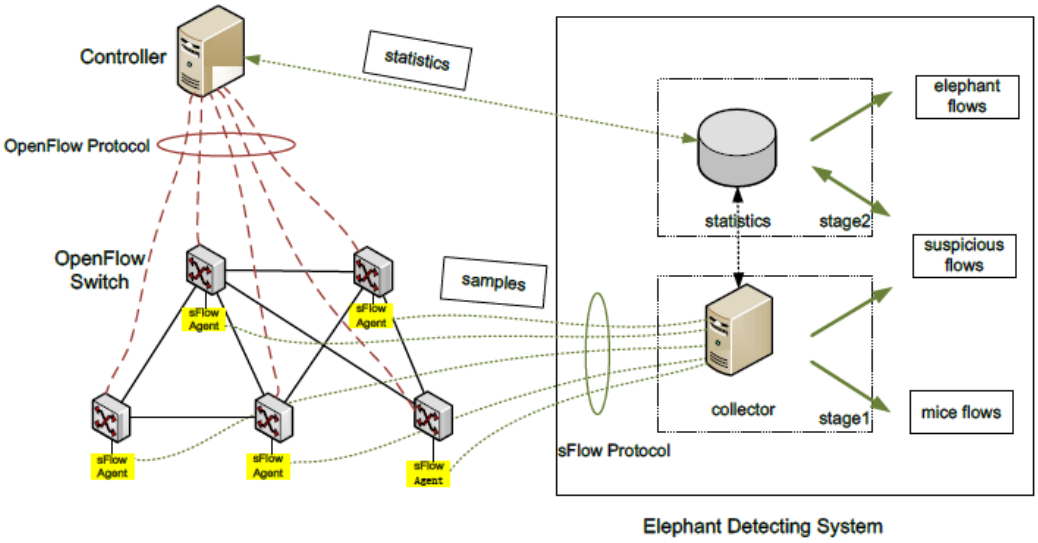

Figure 1 Two-stage elephant flow detection architecture

In this project, my main contribution is in the first stage of the two-stage detection system and the calculation of the time delay in each stage in the first stage. The first stage utilizes the method of sampling. Sampling is a universal approach to obtain traffic measurement profiling. Packet sampling method, sFlow, has been supported by a range of device vendors. It is very simple to implement and adds negligible cost to a switch or router. The sFlow system is scalable to the traffic volume, especially under the frequently changing traffic of data center. Comparing to capturing every flow statistics, the agent samples the packets 1 out of k packets and only sends the sampled header message to the central collector. Thousands of devices can be monitored by a single sFlow Collector. Assuming that each device contributes 10k flows per second, the sFlow detecting system can handle 10million flows per second comparing to 30k per second of NOX controller. This approach has a drawback that it adopts the static decision threshold to select

elephant flows upon existing solution. However, it is significant to adaptively adjust the threshold according to dynamically changing traffic. Moreover, it involves the problem of low accuracy, which might result in positive false error and negative false error. In order to maintain both detecting efficiency and accuracy, we design the two-stage, adaptive elephant flows detecting system. We adopt packet sampling measurement in first stage of detection to distinguish suspicious flows from mice flows initially. The decision threshold of elephant flows can be changed over the dynamic traffic. Then, these suspicious elephant flows are sent to second stage to improve the accuracy.

iii. The first stage: detecting elephant flow using the sampling method

In the first stage, we find the suspicious elephant flow and then the suspicious flows are passed to the second stage for more precise detecting. Because of the inherent drawback of the sampling method, we cannot accurately get the speed of each flow, what we get is just the sampled packets every k packets. In this stage we take the advantage of the sFlow protocol. We make this choice for many reasons. First is that the sFlow protocol is a very mature protocol, they have been many tools that could analyses the sFlow packets. This can help us a lot in the experiments. The second reasons is that a tools called Sflow-RT also has the function of detecting elephant flows. We can compare the time delay of our program and the time delay of the Sflow-RT.

We built up a test environment to compute the delay of our elephant detection system. Loop delay describes how performance aware SDN responds to the appearance of an elephant flow. The first component of delay is the measurement delay, the time taken by the elephant flow detection system to identify the elephant flows.

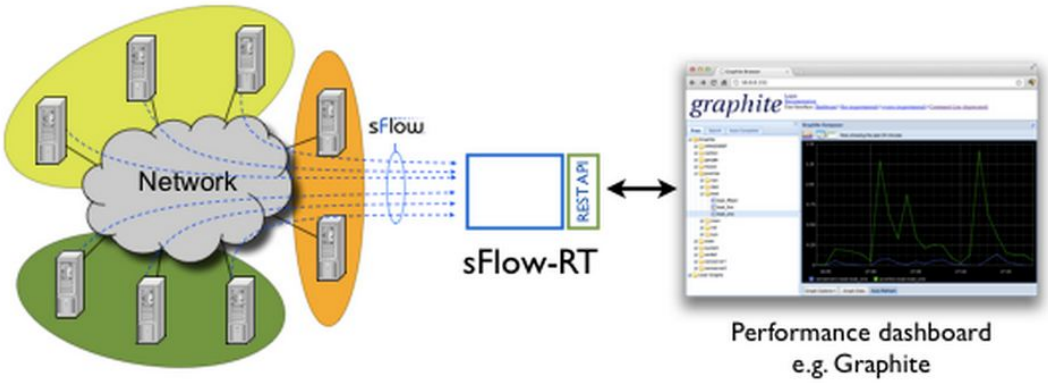

Figure 2. sFlow-RT collector

We selected two different sFlow collector, sFlow-RT and sflowtool. The former provides clear graphic al user interface as illustrated in Figure 1, the latter displays the samples by command line.

The sFlow counter export mechanism extends beyond the network to include server and application performance metrics and the article, Cluster performance metrics, shows how queries to the sFlow-RT analyzer's REST API are used to feed performance metrics from server clusters to operations tools like Graphite. sFlow-RT shows the real-time information summarizing network wide packet loss and error rates can easily be gathered, even if there are thousands of switches and tens of thousands of links. It may take about several hundreds of milliseconds to identify an elephant flow by using sFlow-RT. The sflowtool command line utility is used to convert standard

sFlow records into a variety of different formats. This collector is much effective than sFlow-RT, but more experiments are needed for further validation.

We did experiment to test the difference in the time delay. The result shows that Sflow-RT is really slow in detecting elephant flow. So, in our sequential we abort the usage of sFlow-RT and focus on developing our own algorithm.

iv. A new algorithm in the first stage

Large flow refers to flows who occupy a large proportion of the total bandwidth during a certain time interval. Here we can define a factor to denote this proportion. The definition of large flow seems quite simple, but this is an unknown factor in the definition, the time interval. We can choose one second, one minutes or several minutes. It's hard to say which is the best, it may depend on our network conditions to choose a suitable time interval. Just because of this point, I think that definition of detecting large flows is based on certain assumptions. Only based on this assumption we can retain the accuracy of large flow detection.

In fact what we measure in the average speed of a flow during certain time interval. But the character of large flows vary. Some flow may have small speed but large volumes of total packets. Some may send large quantities of packets in a small time. When we try to choose a suitable time interval, this factor cannot be ignored. Also for different network conditions, it may have special demands. Some networks need low time delay and some networks require high accuracy. So when we try to find a common way to detect a large flow in unknown network conditions, it's hard for us to choose a certain time interval, and how can we solve this problem? I think we can choose two time interval, a small one and a large one. We use two variables to detect large flows. For example the small one is 0.2s, and the large one is 20s. We can get real time speed of the flow, meantime we have the information of the total packets it sends in a long time interval. We can combine these two factors to detect large flow. For flows that exist a long time but the real time speed is small, the packets it send in a long time should be large enough. For flows that may exist shorter but send large volumes of packet in short time, it's real time speed should be large. The program should provide a method that can balance the two factors to detect large flows.

We try to implement large flow detection utilizing sflowtool. It's a slow collector. Sflow agents send two types of sample information: flow sample and counter sample. I this term, the flow sample is useful for us, while the counter sample including information about the condition of the switch can be neglected. We get flow information from sflowtool and store them in databases. Every flow has two counters. We call one counter that count the times of a flow being samples in the small time the small-time counter, and another the long-time counter. We then need to set a threshold. The threshold should takes both the two counter into consideration also, it should be adaptive. That means it should also consider the overall network condition. We then need to define a variable to calculate the character of a certain flow. When the variable exceeds the threshold, it will be detected as a large flow. Here I come up with an idea. We can calculate the percentage of a certain flow's small-time counter and long-time counter in the whole small-time counters and long-time counters. We add up all the flows' small-time counters and long-time counters to get the whole small-time counters and long-time counters. The whole counters have taken the networks' overall conditions into condition. Then we can simply calculate the linear combination of the two percentages and compare it with the threshold.

There are several place that can be improved to better detect large flows. We can include flow predictive model into this method. The coefficients of the linear combination denote the impact of the small-time counter and long-time counter on the threshold. We can change it for better performance. Another place that can be improved is the two time interval chosen, we can choose different time intervals, we can even choose three time intervals, but I think now two is enough. More time intervals means more complexity, and to which extend it improve the performance of detecting large flow is hard to say.

v. Experiment testing the algorithm

We defined a factor to denote the proportion of the total bandwidth during a certain time interval. As the character of elephant flows vary. Some flow may have small speed but large volumes of total packets. Some may send large quantities of packets in a small time. When we try to choose a suitable time interval, this factor cannot be ignored. Also for different network conditions, it may have special demands. Some networks need low time delay and some networks require high accuracy. We choose two time interval threshold, a small one and a large one. We use two variables to detect large flows. For example the small one is 0.2s, and the large one is 20s. We can get real time speed of the flow. Meantime we have the information of the total packets it sends in a long time interval. We can combine these two factors to detect large flow. For flows that exist a long time but the real time speed is small, the packets it send in a long time should be large enough. For flows that may exist shorter but send large volumes of packet in short time, it's real time speed should be large. The program should provide a method that can balance the two factors to detect large flows.

In the experiment, we get flow information from sFlowtool and store them in databases. Every flow has two counters. We call one counter that count the times of a flow being samples in the small time the small-time counter, and another the long-time counter. We then need to set a threshold. The threshold should takes both the two counter into consideration also, it should be adaptive. That means it should also consider the overall network condition. We then need to define a variable to calculate the character of a certain flow. When the variable exceeds the threshold, it will be detected as a large flow. Here I come up with an idea. We can calculate the percentage of a certain flow's small-time counter and long-time counter in the whole small-time counters and long-time counters. We add up all the flows' small-time counters and long-time counters to get the whole small-time counters and long-time counters. The whole counters have taken the networks' overall conditions into condition. Then we can simply calculate the linear combination of the two percentages and compare it with the threshold. The coefficients of the linear combination denote the impact of the small-time counter and long-time counter on the threshold. We can change it for better performance. In another hand, we set up a test bed to measure the total detection delay. We apply the SCP to send large file and record the triggering time in detector. We found that it occupied about 15.1 percent of the transmission time with some detection error. Because the history command can only provide second level and we need smaller scale of time stamp. We will collect the record of tcpdump method for further measurement.

vi. The analysis of the time delay in the first stage

All the experiments were done with the existence of background flows, which were generated by our Openstack cloud computing platform. This time, we mainly focus on the accurate time delay

of the first stage.

Just as shown in the figure above, the first stage is cut off by three timestamp into two parts. We define the first timestamp as the time the first packet is generated by a certain flow. Time stamp two refers to when our program finishing parsing the first Sflow packet and restored that flow's information into our database. The duration between the two points is called the measurement delay. The third timestamp points to the time when a flow is marked as a suspicious elephant flow.

Obviously, we thought the time delay of the measurement part should be quite short. The planning delay should be where most of the time was wasted. But our result contradicts this assumption.

The result was shown below. The time delay in the first was so long, occupied more than two thirds of the total time stage of the first stage. That seems quite ridiculous to us.

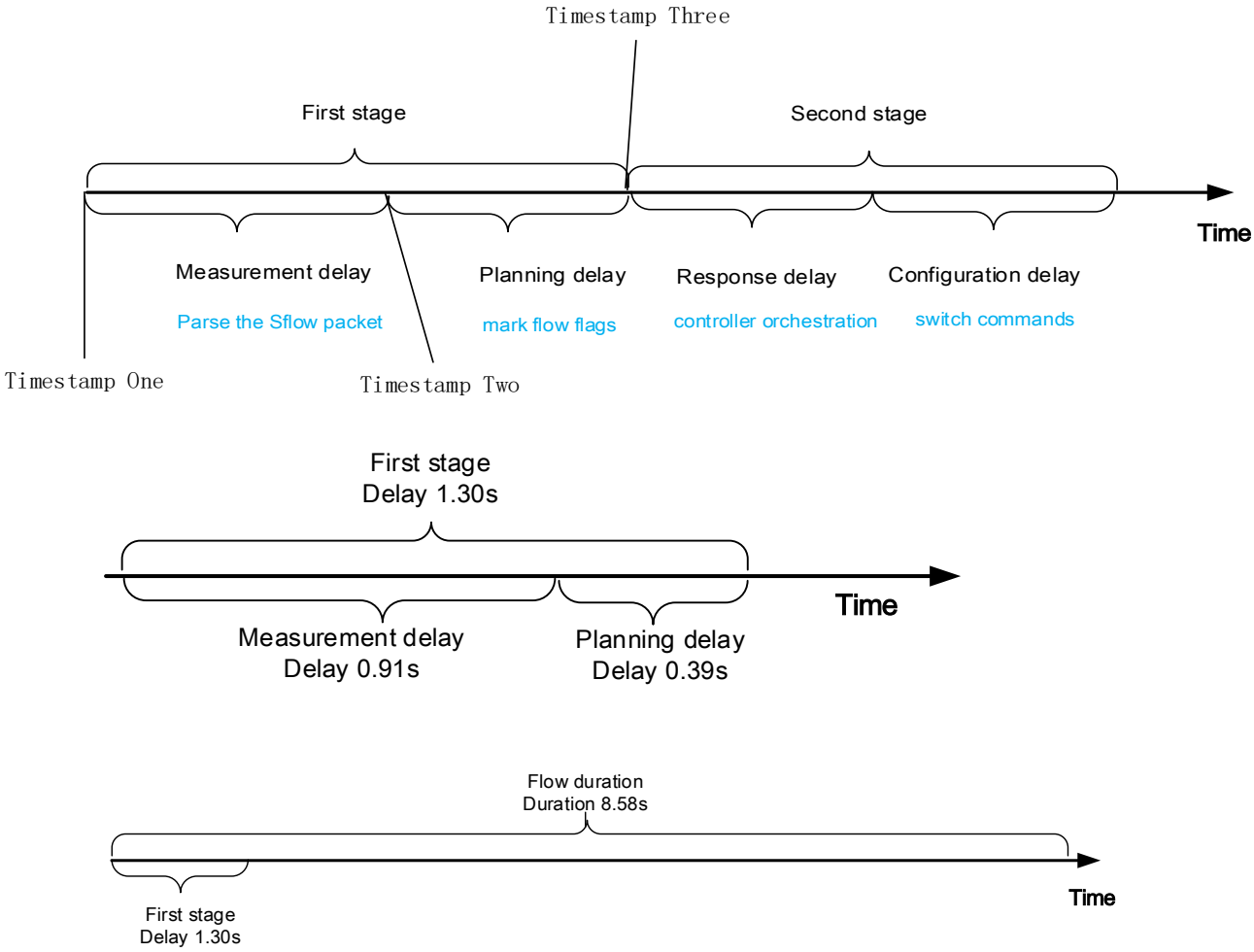

Figure 3 the time delay of the first stage

So we analyze the time duration of the two parts. In the first part, the first sFlow packet is sent to the server, we use the Sflowtool to parse the packet and restore that flow's information

into the database. Why does this process take so long? Is that because the program spends too much time processing the Sflow packet. So extra experiment was designed. This time we used the Wireshark to capture the Sflow packet sent to the server to see when that first Sflow packet arrived. Then we modified the program to print out the time the program finished processing the flow's information. The result proved our previous assumption the program was indeed a little bit slow in processing the flow's information.

The first part is the duration between the time when the elephant flow started and the time its packet got sampled and was sent to the server through the Sflow packets. Obviously, we thought the time delay during this part should be quite short.

Secondly, when the suspicious flow's packets were sampled enough times, the algorithm of our large flow detection program worked to mark this flow as an elephant flow. We thought this should be where most of the time was wasted. This process is quite complex. The time delay during this part depends on various factors which includes not only the algorithm but also the speed of the suspicious elephant flow and the speed of other flows, which also means that we need to control the testing environment to be stable. During all our experiments, all the sFlow packets sent to our server was sampled in the same switch whose flows were quite stable, which means stable background flows. Algorithm is also very important, which is the core part of our program. There are lots of different algorithms for detecting elephant flow, we will later test different methods and gradually optimize our algorithm. During the test, we just use an algorithm which we think is suitable.

The third part is the part a flow is configured as an elephant flow and the controller take actions. The goal of our test is to compute mainly the first two time delays. We also designed experiments to test the time delay of the third part. We think it is mainly composed of the time needed to compute the route and send the route table to the switch.

From the figure below, it isn't hard to find that the time delay of the second part is comparably short, but the time delay of the first part is quite long, which contradicts out previous assumption and why? We first thought the second part needs lots of time because of the algorithm. But in fact, it took quit long for the first sFlow packet to come. When we analyzed this problem, we thought maybe the program took lots of time to analyze the sFlow packet.

Also, we analyze the time delay of the second part. We think the time is short enough. That's a quite good thing. But we cannot ignore that the time delay of the second part depends on several factors. Algorithm is one of them, and we think it is the core of our program. We have already optimized our mechanism from one counter recording the times a certain flow's packet was sampled to two counters, recording more information than before. The two-counter mechanism gives our more option in detecting the elephant flow. We have introduced the mechanism before, so next time we plan to compare the one-counter mechanism and the two-counter mechanism to see which mechanism is more accurate, whether the two-counter mechanism will take more time than the one-counter mechanism. Also, we can modify the parameters in the two-counter mechanism to optimize its performance.

vii. Conclusion

In the detecting of elephant flow, we need to balance the accuracy and the time delay. These two factors are contradictory. We should try our best to attain more information from the sampled packets. We need to use the mathematical method to gain the statistical characteristic of the flows

to choose which are suspicious and send them to stage two for more precise detection. We compare the efficiency of our program and sFlow-RT, the accuracy is similar but the time delay of sFlow-RT is longer. Also, we invent a new algorithm using two counters instead of one. From the experiments, we can see that two-counter algorithm is better than the previous one-counter algorithm. The time delay is similar but the accuracy is improved. That proves that our new algorithm is a successful one.

viii. Appendix: How to use the Sflow-RT

Sflow-RT

一.安装

wget http://www.inmon.com/products/sFlow-RT/sflow-rt.tar.gz
tar -xvzf sflow-rt.tar.gz
cd sflow-rt
./start.sh
使用浏览器登陆 http://localhost:8008 可以登陆图形界面查看 REST API 以及进行相关的设置，如定义 flow，设置 threshold，查看 flow 的信息，查看 Event 信息等。
Sflow-RT 默认监听 localhost 的 6343 端口，agent 采样的信息应该发送到此端口。
OVS 开启 sflow 命令实例：
ovs-vsctl -- --id=@sflow create sflow agent=eth3 target=\"202.120.32.2:6343\" header=128 sampling=64 polling=10 \-- set Bridge br4 sflow=@sflow

二.常用 API

1. define address groups

curl -H "Content-Type:application/json" -X PUT --data "{external:, internal: }" http://localhost:8008/group/json

example:

curl -H "Content-Type:application/json" -X PUT --data "{external:['0.0.0.0/0'], internal:['10.0.0.0/8']}" http://localhost:8008/group/json

组定义是进行流控制的好方法，这个例子就定义了从外网发到内网的 flow 被识别。

2. define flows

curl -H "Content-Type:application/json" -X PUT --data "{keys: , value: , filter: }" http://localhost:8008/flow/（name） /json

example；

curl –H "Content-Type:application/json" -X PUT --data "{keys:'ipsource,ipdestination,ipprotocol',value:'frames',filter:'sourcegroup=external&destinationgroup=internal'}" http://localhost:8008/flow/incoming/json

此例定义了 name 为 incoming 的 flow。Value 的值为 frames 或者 bytes

3. define thresholds

curl -H "Content-Type:application/json" -X PUT --data "{metric:, value: }"

http://localhost:8008/threshold/(name) /json

example:

curl -H "Content-Type:application/json" -X PUT --data "{metric:'incoming', value:1000}" http://localhost:8008/threshold/incoming/json

此例定义了 name 为 incoming 的 threshold,设置的门限值为 1000。在 Sflow-RT 中 value 表示的都是速度，也就是 frames(bytes)/s。metric 为已经定义过的 flow 的 name。此例中 threshold 的 name 设置的与 metric 相同。

4. receive threshold event

curl "http://localhost:8008/events/json

example:

curl "http://localhost:8008/events/json?eventID=4&timeout=60"

此例读取 events 中 eventID 大于 4 的 events 信息，会等待 60 秒接收新触发的 event 信息

5. monitor flow

curl http://localhost:8008/metric/（agent）/(datasource).incoming/json

example:

curl http://localhost:8008/metric/10.0.0.16/4.incoming/json

额外的关于 flow 的信息可以从 event 当中读取。
以上列了最常用的 API，更多的 REST API 可以登录到其 web 界面查看。

三. Event 触发机制初探

在 Sflow-RT 中触发 event 主要分为三步：
1.定义 flow
http//localhost:8008/flow/json
option:post value={name: ,key: ,value:bytes/frame,filter: }
我在自己的机子上建了两台虚机（ip 分别为 192.168.100.103，192.168.100.105）
我设置的 flow：
name:trace(后面设置 threshold 的时候设置的 metric 应该就是定义的 flow 的 name）
key:ipsource,ipdestination,ipprotocol(感觉这个选项很重要，还可以加很多内容，不过没有文档只能看别人的 example 里都用了什么。后面读取到的 flow 的 top key 的内容跟现在定义的 key 是对应的）
filter:ipsource=192.168.100.103&ipdestination=192.168.100.105(我是用 192.168.100.103ping 的 192.168.100.105 来产生流量）
2.设置 threshold
http://localhost:8008/threshold/json
option:post value={name: ,metric: ,value: ,filter: }
我是设置成这样的
name=no1 ,metric=trace (就是刚才定义的 flow 的 name), value=5000(尝试用 5k 不行） ，filter (我没设置，在别的代码里我看到这里可以过滤系统的类别等很多参数，还在测试有怎么用）
3.读取 event 信息
http://localhost:8008/events/json

option:get

｛agent: ,datasource: ,eventID: ,metric: ,threshold: ,thresholdID: ,timestamp: ,value: ｝

datasource 应该是类似与 Ifindex 的东西，sflow 对每个 interface 应该都有自己的编号。在 sflow-RT 中还会读到 dsIndex，应该都是一样的。

timestamp 存储的应该是触发时的时间，value 是当时的值。在 Sflow-RT 中读取到的 value 值都是速度也就是 bytes(frame)/s。

我实验的结果，感觉是超过了设置的 threshold 才会触发，不过感觉它判断的标准并不是检测的瞬时值，有可能是一段时间的均值或什么的，要看源代码才能知道。而且如果一直超过 threshold 的话不会一直触发，而是要等值下降到 threshold 以下，如果再次超过 threshold 才会触发。具体采用什么算法要看到源代码才能知道。

为了测试 Sflow-RT 触发 event 后会不会向 controller 发送 message,采用 wireshark 抓包，一边让 sflow-RT 不断触发 event，一边抓包。在机子上使用的 controller 是 Floodlight，floodlight 会占用两个端口 6633 和 8080。

其中 6633 端口主要用来跟 OVS 通信，通过抓包实验也已经证实。

8080 端口是 Floodlight 的 static flow pusher 占用的端口，允许通过 REST API 来想起传递信息。抓包实验显示 8080 端口存在流量，但我分析下来并没有发现 message 的包。

四.使用小结

Sflow-RT 是一个功能很完善的流量分析工具，其中自带了简单的大象流大检测机制，通过触发 event 来提示检测到了大象流（也就是超过了设置的 threshold 的流）。

Sflow-RT 的一个优点就是提供了功能很完善的图形化界面。在 Sflow-RT 中的 flow 的定义是非常方便的，除了上面提到的通过命令的方式来定义 flow 以及 threshold，在其 web 界面也提供了图形化的定义方式，使用起来也非常方便。同时在 web 界面中可以随时观察各个 flow 的信息，使用图像的形式显示出来，非常直观，能对各个 flow 的信息有个整体的印象，更加详细的信息可以通过上面提到的命令读取。

目前还存在疑惑的主要在两个地方：

1. 由于找不到其源代码，event 的触发机制，目前还不了解。触发 event 之后会不会发送 message 给 controller 以便采取相应的措施？它通过什么算法判断一条流为大象流？目前都需要解决。

2. 由于缺少官方说明文档，在 Sflow-RT 中一些参数的意义不确定。如 datasource，dsIndex，目前猜测是 Sflow-RT 对采样的 interface 的编号。对在 filter 及 key 当中可以使用的参数也缺少相关的资料

五.参考资料

http://www.inmon.com/products/sFlow-RT.php

http://blog.sflow.com

以及安装文件夹中附带的测试脚本