Part I

Efficient Statistics Collection in Pure SDN

Chapter 2

Collection of Globally Consistent Statistics in Software Defined Networks

2.1 Introduction

To perform various network management tasks, the SDN (Software Defined Networking) controller needs to have an up-to-date and globally consistent snapshot of the network. This snapshot is then used to estimate load on the links, to identify the bottleneck links, and to measure packet losses in the network. Accurate estimate of these parameters is essential to perform various network management tasks such as load balancing, QoS assurance, meeting the SLA (service level agreement) requirements etc [121]. The global state of the network is said to be consistent if a packet belonging to a flow is recorded as "received" at a switch then the same packet must have also been recorded as "sent" by all the preceding switches with respect to the flow [41]. Failing to collect a consistent global

---

- Sandhya Rathee, Rahul Sharma, Piyush Kumar Jain, K Haribabu, Ashutosh Bhatia , Sundar Balasubramaniam. *OpenSnap: Collection of Globally Consistent Statistics in Software Defined Networks*, In 2019 11th International Conference on Communication Systems & Networks (COMSNETS), pp. 149-156. IEEE, 2019.

snapshot can lead to the poor estimation of various network parameters such as queue depth, load on links [38].

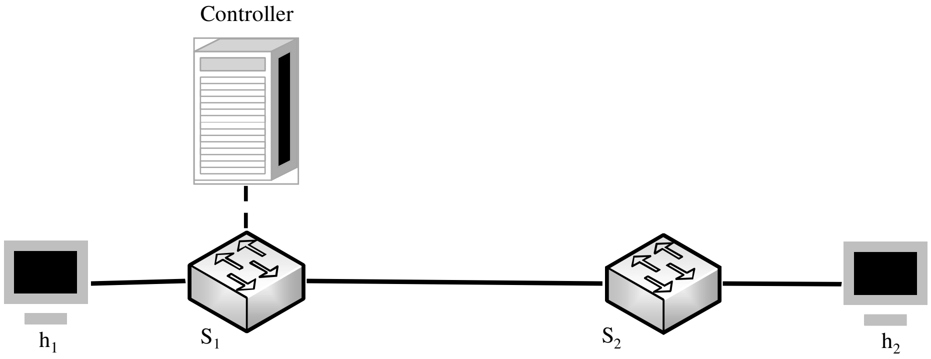


Figure 2.1: Example to illustrate challenges in consistent statistics collection.

Prevalent network monitoring methods focus on per flow or per port statistics collection [1] [2]. These statistics, when viewed across the switches, are likely to be inconsistent if a specific order is not enforced while collecting them. A traditional method to collect global state in an SDN network is to get flow statistics from all the switches by polling them with a specific polling rate. Due to the delay variations between controller and switches, polling based statistics do not guarantee a consistent global state [3]. For example, consider a network as shown in Figure 2.1, in which a packet P is transmitted from switch S_1 to switch S_2. Also, consider that there is no packet loss in the network. We define the following four events,

- E_1: Packet P arrives at switch S_1 and matches[1] with a flow entry.
- E_2: Packet P arrives at switch S_2 and matches with a flow entry.
- E_3: Switch S_1 receives statistics request message from the controller and sends the statistics to the controller.
- E_4: Switch S_2 receives statistics request message from the controller and sends the statistics to the controller.

Now depending on the order of these events w.r.t time, there can be three possible cases. (i) The occurrence of the events is in the order E_1, E_2, E_3, and E_4. In this case,

---

[1] When a packet matches with a flow entry in OpenFlow switch, it increments the packet counter of the matched flow entry.

the packet P is counted in sent statistics of switch S_1 and is also counted in received statistics of switch S_2. Thus, it gives consistent statistics. (ii) The occurrence of the events is in the order E_3, E_1, E_2, E_4. Here, the packet P is counted in the received statistics of switch S_2 but not in the sent statistics of switch S_1. Thus, it gives inconsistent statistics. (iii) The occurrence of the events is in the order E_1, E_3, E_4, E_2. That is, the packet P is recorded as sent at switch S_1 but not as received at switch S_2. This can lead the controller to a wrong conclusion that the packet is lost. The wrong or inconsistent statistics can lead the SDN controller to make erroneous decisions, especially in case of load balancing [38] and bottleneck link identification. Here we considered a single packet, even with large number of packets it will give similar results. The effect of the order of events will remain same as inconsistency in collected statistics is not related to time duration but to the order of occurrence of events. Thus consistency of the collected statistics depends on the order in which the switches receive the statistics request from the controller and send the corresponding statistics reply to the controller. This order can not be enforced by the SDN controller due to variations in delays on the control and data links. Therefore, we need a protocol to enforce the order of statistics collection to collect statistics in a globally consistent manner.

State of the network is a collection of states of switches and links. It can be measured by querying switches. When a part of the state across the switches is causally related i.e., an attribute in one switch is causally affected by the same attribute of another switch. Such a state needs to be measured preserving this causal relation. For example, packet counters or byte counters in a switch are causally related to the same counters in the predecessor switch with respect to a flow. In certain applications such as congestion prediction, trace recording [122], applying updates consistently on all switches [123], dynamic visualization of network traffic patterns [124], a measurement that preserves this causal order is expected to yield accurate results.

In this chapter, we propose an algorithm, OpenSnap, that provides consistent statistics for each flow in OpenFlow based SDNs. The idea is inspired by Chandy-Lamport Algorithm [40] [41]. Chandy-Lamport algorithm is a well known algorithm to determine the global state of a distributed system. The algorithm works by sending a special marker message through the links in underlying network. The nodes have the ability to record their state when they receive the marker. After recording the state, the node forwards the marker to all out-going links. The algorithm obtains a globally consistent state even when the state of all the nodes are not recorded at the same instant. The marker packet delineates the packets which are recorded in the global snapshot and which are not recorded in the global snapshot.

Though the idea is inspired by Chandy-Lamport Algorithm [40], there are a few differences conceptually and in implementation when applied to OpenFlow based SDNs. (i) Chandy-Lamport algorithm requires storage space on the nodes to store the snapshot and channel state. OpenFlow enabled switches do not have the capability to save their state locally at a given time. The switches maintain the cumulative counters for statistics. OpenFlow switches support sending flow statistics upon receiving a flow statistics request from the controller. Our algorithm requires that switches should send the statistics to the controller on receiving a marker packet. But OpenFlow (as for the current OpenFlow [18] Standard) does not have any action which sends the flow statistics on the arrival of a particular packet. Thus, to solve this issue, we use Experimenter action field to extend OpenFlow protocol. We implement a new action called "*send_stats*" in Open vSwitch [54]. On arrival of the marker packet at a switch *send_stats* action is performed, the switch then sends statistics of all of its flows to the controller. The statistics collection process is over when the SDN controller receives the statistics from all the switches in the network. (ii) Chandy-Lamport algorithm assumes that the channels have infinite buffers. But this might not be the case in a network of switches. (iii) Chandy-Lamport Algorithm uses single marker packet. OpenSnap algorithm uses two marker packets to avoid looping over a link and to ensure the termination of the algorithm. To the best of our knowledge, this is the first work to provide consistent statistics in OpenFlow based SDNs. The results show that, OpenSnap outperforms the state-of-the-art approaches in consistent statistics evaluation.

2.2 Related Work

In this section, we discuss the existing approaches related to statistics collection in SDN networks.

OpenNetMon [1] is a network monitoring open-source software that monitors all the flows in a network. OpenNetMon polls the edge switches of every flow and collects the statistics. The collected statistics are used to monitor per-flow metrics, especially delay, throughput, and packet loss. The polling frequency increases when new flows are added and reduces when the flow rate becomes constant. This adaptive rate of sampling reduces the network and switch overhead. OpenTM [55] provides a traffic matrix of SDN networks, representing the volume of traffic between the source and destination pairs of all the flows in the network. It presents different strategies to select switches for polling. There is a trade-off between the measurement accuracy and the maximum load on each switch. OpenTM demonstrates that better performance is accomplished by using a non-uniform distribution querying strategy as it selects the switches which are near to the destination in contrast to uniform schemes.

CeMon [2] proposes two schemes for polling the network, namely, Maximum Coverage Polling Scheme (MCPS) and Adaptive Fine-Grained Scheme (AFPS). MCPS globally optimizes the polling cost. It proposes a greedy strategy to select the switches in a cost-effective manner so that all flows are covered. It proposes a heuristic called Dynamic Adjust and Periodical Reconstruction (DAPR), which dynamically handles the arrival of new flows. If the current polling scheme covers the new flow then no action is taken otherwise it adds one polling for the currently arrived flow. If a flow expires then the expired flow is removed from the polling scheme. AFPS is a complementary scheme for MCPS, that aims at providing a solution when to poll the switch for a given flow. AFPS deploys various schemes to decide the polling frequency for a given flow on a given switch. But the most optimal among the proposed schemes is Sliding Window Based Tuning (SWT). This scheme queries the switches for a flow and calculates the difference between the last two readings. This difference is used to dynamically tune the sampling frequency.

FlowRadar [125], is a better version of NetFlow [14]. In case of high traffic where data processing needs to happen at a very fast rate, NetFlow is unable to keep up with the rate and therefore in some of its implementations, it monitors only a subset of packets. FlowRadar overcomes this limitation by using less bandwidth and small memory overhead. It encodes the per-flow counters in a constant time using little memory of the switches. The decoding and analysis of the network-wide flow occur at a remote controller. LossRadar [126] provides a solution to detect the packets lost in the data center networks independent of their root causes (i.e., congestion, persistent black holes, transient black holes, and random drops). LossRadar installs meters in all the switches to capture unidirectional traffic. It checks for packet loss and reports to the controller immediately. To capture the packet header information of the lost packet, LossRadar provides traffic digest at every switch which stores the information about the lost packet header.

PayLess [127] proposes an adaptive monitoring algorithm. When a PacketIN message is received at the controller, it adds a new flow in active flow table along with its expiry time t. If the flow expires in time t, then the controller gets the statistics of the flow in FlowRemoved message. Otherwise, when the time-out event occurs, the controller sends the flow statistics request message to the switches for that flow. If the difference between the previous byte count and the current byte count is not above the threshold, then the time out is multiplied by a small constant. If the difference is above the threshold, then the time out is divided by a small constant. FlowSense [57] measures the link utilization in the network with zero measurement cost. It uses control messages like PaketIN and FlowRemoved to estimate the network metrics. But the performance metrics estimations are far from the actual values as large flows generate sparse FlowRemoved packets. FlowSense works well only when there are large number of small duration flows. OpenSample [128] is a sampling-based measurement method. It uses one out of N packets for sampling. The network performance metrics are estimated by the sampled packets. This works well in case of elephant flows only. In [129], the authors proposed a solution to create a snapshot of the network at a given time in the history. To create a snapshot in the history, they logged the OpenFlow messages between the SDN controller and switches. Their main goal is to identify the root cause of a problem using history. Whereas, our method provides a consistent snapshot of the current state of the network that would help to take decisions in both present and future.

All the solutions discussed above use different strategies to collect statistics and use the collected statistics to compute network throughput, link utilization, packet loss, and blackholes in the network. Capturing and monitoring the global network state is important for efficient routing, performance monitoring, Quality of Service (QoS) assurance etc. The goal of the solutions is to analyse the performance of the network. They does not guarantee consistent statistics collection.

In-band Network Telemetry (INT) [130] can be used to collect per flow or per path statistics. Though it is possible to record consistent statistics for a given flow but it is not trivial to collect globally consistent statistics for the entire network. SpeedLight [38] collects per-port statistics. However, it does not guarantee consistent statistics collection in every run of the proposed protocol. This scenario occurs when the channel state is considered and difference between the snapshot ID and ID of the upstream neighbor/s is more than 1. If any inconsistency is detected in the collected statistics, the controller has to run the protocol again. Thus SpeedLight is not time-efficient. In addition both INT [130] and SpeedLight [38] require a programmable data plane. In this chapter, we propose an algorithm to collect the consistent statistics in OpenFlow based SDN network.

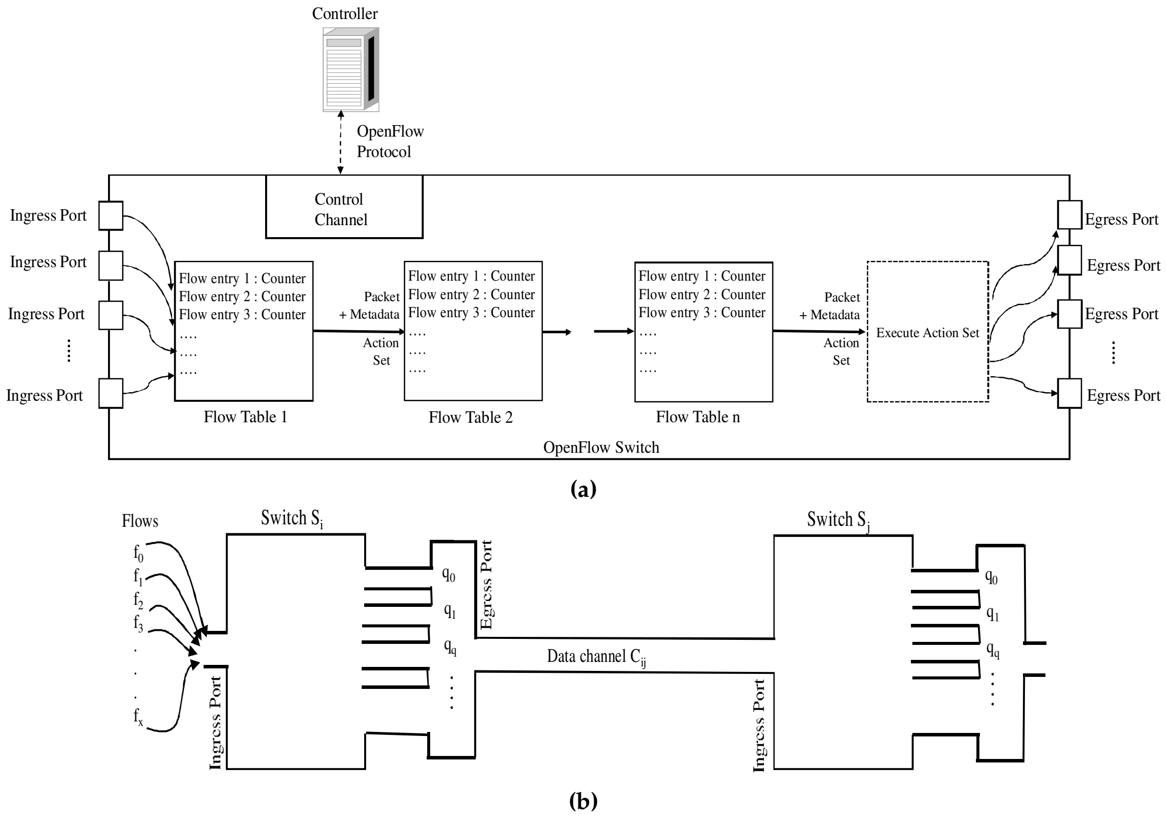


Figure 2.2: (a) OpenFlow Switch, (b) Detailed diagram of two switches directly connected to each other.

2.3 OpenSnap

2.3.1 System Model

We consider a SDN network with OpenFlow 1.3 compatible switches containing multiple ports each connected to a switch or a host where each port supports multiple queues [131]. Figure 2.2 (a) depicts the internals of a switch. A switch consists of multiple flow tables and each flow table consists of multiple flow entries with their counters. Each switch contains multiple ports each connected to a switch or a host and each egress port supports multiple queues [131]. Figure 2.2 (b) shows two switches S_i and S_j connected to each other through a data channel C_{ij}. Flow entries at each switch are defined using input port, source host address, and destination host address. It is assumed that all packets of a given flow go through the same path. The network supports different forwarding classes i.e., flows are assigned to different queues based on their priority or QoS requirements (refer Figure 2.2). The flow to queue mapping is dynamically done by the SDN controller. Now depending on the queue scheduler the order in which packets are transmitted through egress port can be different from the order in which they are received at the ingress port. This introduces non-FIFO (First-In-First-Out) order to the transmission of the packets with respect to the order in which they are received at the switch.

The underlying switches are connected through links/channels. The forwarding of packets from one end of a link to another end can happen in FIFO or Non-FIFO order. In *OpenFlow networks with FIFO channels*, the outgoing packets for transmission are scheduled based on order of their arrival at the switch. Whereas, in *OpenFlow networks with Non-FIFO channels* the outgoing packets for transmission could be scheduled irrespective of the order of their arrival. OpenFlow network with FIFO channels has only a single queue at every output port of the underlying network switches whereas in OpenFlow network with Non-FIFO channels, the switches can have multiple queues configured at the output ports and order of packet transmission depends on the queuing scheduler.

Communication between SDN controller and the underlying switches can happen in two ways: out-of-band and in-band. In an out-of-band controller configuration, the switches are directly connected to the SDN controller through dedicated links. Whereas, in an in-band controller configuration the controller is not connected to each switch through a dedicated link. The controller is just like any other host in the network. There are some advantages of out-of-band configuration like, the communication is more secure, low communication delay between the switches and SDN controller [132]. However, there are some disadvantages also: (i) costs involved in laying dedicated links are huge (ii) scaling can be an issue when new switches are added. Due to these limitations, an in-band controller is preferred. We are considering an in-band controller configuration.

Globally consistent statistics is a set of statistics collected from all the switches for a given flow such that every packet that is recorded as sent at a switch must have been recorded as either received at the next switch or present in the channel [1] or in the queue or is dropped. In OpenFlow packet processing sequence, the packet counter of a flow entry is updated as soon as the packet matches the flow entry. Once the packet exits the processing pipeline, the packet is queued into its respective queue. If the queue does not have enough space, then the packet may be dropped. Similarly, the next switch maintains packet counters for each flow entry. Consider a network with N switches and X number of flows. Also consider I number of queues are configured in every switch. Let S be the set of switches in the network, S = $\{S_1, S_2, S_3, ..., S_N\}$, and F be the set of flows in the network, F = $\{f_1, f_2, f_3, ..., f_X\}$. Given a flow f_k, $1 \leq$ k $\leq$ X, from switch S_i to switch S_j, $1 \leq$ i, j $\leq$ N and i $\neq$ j, the packet counters for flow f_k are labelled as $sent(f_i^k)$ and $recv(f_j^k)$ on switch S_i and S_j respectively. The relationship between them is defined as,

$$sent(f_i^k) = recv(f_j^k) + Q_{iq}^k + C_{ij}^k + drop(f_i^k) \tag{2.1}$$

where C_{ij}^k is the number of packets of k^{th} flow present in the channel connecting switch S_i and switch S_j, Q_{iq}^k is the number of packets of k^{th} flow queued in q^{th}, $1 \leq$ q $\leq$ I, queue of switch S_i for transmission and $drop(f_i^k)$ is the number of packets dropped before queueing. Since C_{ij}^k, Q_{iq}^k, and $drop(f_i^k)$ are always ≥ 0, Equation 2.1 can be written as,

$$sent(f_i^k) \geq recv(f_j^k) \tag{2.2}$$

---

[1] We use channel and link interchangeably in this thesis.

2.3.2 Algorithm

We assume a multi-VLAN enterprise network which uses Spanning Tree Protocol (STP) (802.1d) or Rapid STP (RSTP)(802.1w) [133]. Let there be N switches in the network labelled as $S_1, S_2, ..., S_N$. Assuming that the i^{th} switch has k_i number of interfaces, we label these interfaces as $I_i^1, I_i^2, ..., I_i^{k_i}$. The switches in the underlying network are connected through bidirectional links and the traffic is going in both directions. Let SF_i be the set of flows going through switch S_i. For every l^{th} flow, f_i^l, on switch S_i, we define $IN(f_i^l)$ and $OUT(f_i^l)$, the ingress interface for flow f_i^l and egress interface for flow f_i^l respectively. The number of packets sent and received from/at switch S_i for flow f_i^l are recorded as $sent(f_i^l)$ and $recv(f_i^l)$ respectively. A summary of the symbols used is provided in Table 2.1.

Table 2.1: List of symbols used in consistency statistics collection

Symbol	Meaning
S_i	i^{th} switch in the network
k_i	Number of interfaces in switch S_i
I_i^j	j^{th} interface of switch S_i
SF_i	Set of flows going through switch S_i
f_i^l	l^{th} flow going through i^{th} switch, where $1 \leq l \leq \|SF_i\|$
$IN(f_i^l)$	Ingress interface for flow f_i^l at switch S_i
C_{ij}^l	Number of packets of l^{th} flow present in the channel connecting switch S_i and switch S_j
Q_{iq}^l	Number of packets of l^{th} flow queued in q^{th} queue of switch S_i for transmission
$OUT(f_i^l)$	Egress interface for flow f_i^l at switch S_i
$sent(f_i^l)$	Number of packets sent for flow f_i^l by switch S_i
$recv(f_i^l)$	Number of packets received for flow f_i^l by switch S_i
$drop(f_i^k)$	Number of packets of l^{th} flow dropped before queuing
M_1, M_2	Marker 1, Marker 2 respectively

OpenSnap makes use of two special packets called marker packets denoted by M_1 and M_2. Marker M_1 is used to record the statistics of the flows that reached the switch and marker M_2 is used to record the statistics of the flows in the channel. To start the network statistics collection, the controller sends marker M_1 to one of the switches. Each switch runs OpenSnap Algorithm 2.1. When a switch S_i receives M_1 on its interface I_i^j, it records sent statistics for all the flows which are forwarded through it (line 2-4 Algorithm 2.1). It also records the number of received packets for all the flows which have their input interface same as the interface at which the marker M_1 has arrived (line 5-7 of Algorithm

Algorithm 2.1: OpenSnap Algorithm for Switch S_i

```
1  Input: Marker Packet $M$ received on Interface $I_i^j$
2  if $M$ is $M_1$ then
3      foreach flow $f_i^l \in SF_i$ do
4          Record $sent(f_i^l)$;
5          if $IN(f_i^l) = I_i^j$ then
6              Record $recv(f_i^l)$;
7          end
8      end
9      for $k \leftarrow 1$ to $k_i$ do
10         if $k \neq j$ then
11             Send $M_1$ through interface $I_i^k$;
12         end
13     end
14     Send $M_2$ through interface $I_i^j$;
15 else if $M$ is $M_2$ then
16     foreach flow $f_i^l \in SF_i$ do
17         if $IN(f_i^l) = I_i^j$ then
18             Record $recv(f_i^l)$;
19         end
20     end
21 end
```

2.1). It then forwards marker M_1 to all other interfaces (line 9-13 of Algorithm 2.1) except on the interface on which the marker M_1 is received and sends M_2 back through I_i^j (line 14 of Algorithm 2.1). Once the switch forwards M_1 on an interface, we expect M_2 to be received on that interface, provided the link through this interface is connected to another switch running OpenSnap algorithm. Once M_2 is received on an interface $I_i^{j'}$, the switch records received statistics of all the flows which have input interface same as the interface at which the marker M_2 has arrived (line 15-20 of Algorithm 2.1). Since M_1 is never sent back on the port on which it is received and the network has no loops, the algorithm always terminates and M_1 arrives on each switch exactly once.

2.3.3 Correctness

To prove the correctness of OpenSnap algorithm in terms of collecting global consistent statistics, we consider two types of network, (i) OpenFlow based network with FIFO channels. (ii) OpenFlow based network with Non-FIFO channels.

2.3.3.1 OpenFlow Based Network With FIFO channels

The default queueing mechanism on nearly all the interfaces of the network nodes is FIFO [134]. In a given queue the packet transmission is done in FIFO order. Thus, to simulate a network with FIFO channels, we consider only a single queue on every interface of switch.

We consider an arbitrary flow f_x in our network. Let our flow correspond to a path P, which is an ordered set of switches, $\{S_i, S_{i+1}, ..., S_{N_s}\}$, where $1 \leq i \leq N_s$. Thus, our flow will be f_i^x. Switch S_1 and S_{N_s} are the source and destination switches respectively for the flow f_i^x. Since a path is an ordered set of switches, the packets of the given flow always go from S_i to S_{i+1}. The consistency condition in Equation 2.2 requires that,

$$recv(f_{N_s}^x) \leq sent(f_1^x) \tag{2.3}$$

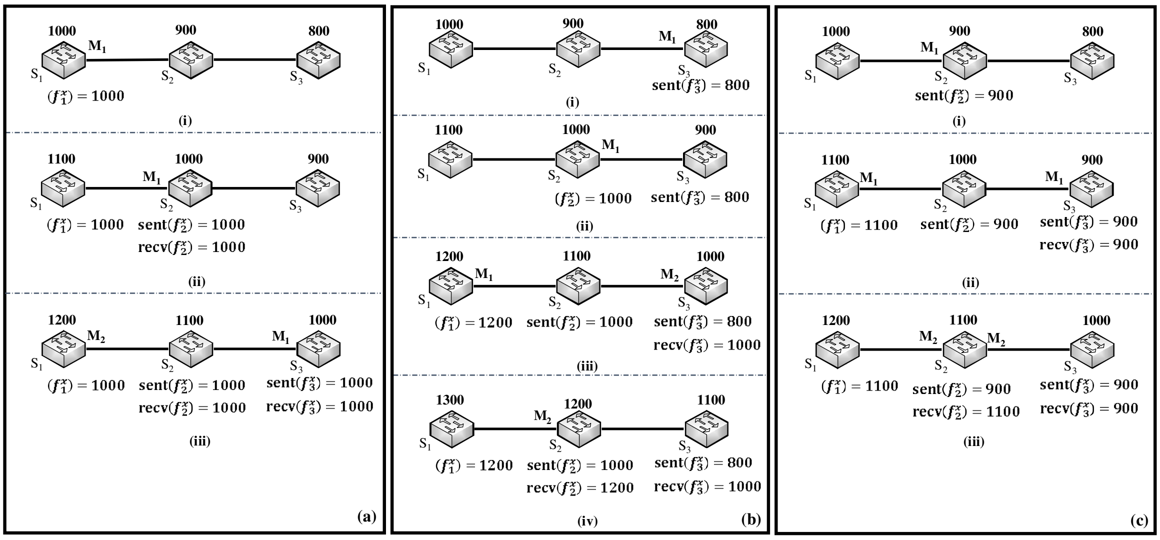


Figure 2.3: Illustrating that OpenSnap gives consistent statistics in all 3 cases ((a) M_1 incident on source switch S_1 (b) M_1 incident on destination switch S_3 (c) M_1 incident on intermediate switch S_2) when the flow is going from switch S_1 to switch S_3.

We argued earlier that M_1 will be received on each switch exactly once. Among the switches which are part of path P, let M_1 reach to a switch S_k first. Depending on the value of k we can have three cases,

(a) $k = 1$, i.e., source switch.

(b) $k = N_s$, i.e., destination switch.

(c) $1 < k < N_s$, i.e., intermediate switch.

We first demonstrate the three possible cases, that are discussed above, with the help of an example. Consider three switches, S_1, S_2, and S_3 connected in a linear fashion with FIFO channels as shown in the Figure 2.3. Also assume that the links between the switches have a delay of 1 sec and a flow f_x is going from switch S_1 to switch S_3 with a constant packet rate of 100 packets/sec. The marker incident on switches S_1, S_3, and S_2 for the cases (a),(b), and (c) respectively. We observe the state of the network after every second. As evident from the Figure 2.3(a), (b), and (c) we get consistent results which satisfy Equation 2.2 for all three cases. We now prove that the consistency condition is maintained in each of the three cases.

In case (a), marker M_1 incidents on the source switch S_1 of flow f_x, marker traces the same path as flow. Thus, except for the source switch, for all other switches, marker M_1 incidents on same interface as flow. Thus, received statistics, $recv(f_i^x)$, and sent statistics, $sent(f_i^x)$, are identical for each switch S_i such that $1 < i \leq N_s$. Further, since packets are sent on the link in the order they are received, a packet counted in $sent(f_i^x)$ is sent before marker M_1 and is counted in $recv(f_{i+1}^x)$. Thus, $sent(f_i^x) = recv(f_{i+1}^x)$ for each $1 \leq i < N_s$. This implies that,

$$recv(f_{N_s}^x) = sent(f_1^x) \tag{2.4}$$

Equation 2.4, satisfies Equation 2.2.

In case (b), marker M_1 incidents on the destination switch of flow f_x. Marker M_1 moves opposite to the packets in the flow. Thus, marker M_1 moves from switch S_{i+1} to switch S_i where $1 \leq i < N_s$. The sent statistics for flow f_x, $sent(f_i^x)$, is recorded when marker M_1 is received on switch S_i. Subsequently, switch S_i sends marker M_2 to switch S_{i+1} and marker M_1 to switch S_{i-1}. When switch S_{i+1} receives marker M_2, it records received statistics as $recv(f_{i+1}^x)$. In order delivery ensures that,

$$recv(f_{i+1}^x) = sent(f_i^x), \quad 1 \leq i < N_s \tag{2.5}$$

When switch S_{i-1} receives marker M_1, it records sent statistics as $sent(f_{i-1}^x)$ and send marker M_2 to switch S_i. When marker M_2 is received on the interface same as input interface of flow, switch S_i records $recv(f_i^x)$. Since switch S_i receives marker M_1 before marker M_2, we have,

$$sent(f_i^x) \leq recv(f_i^x), \quad 1 < i < N_s \tag{2.6}$$

Using equations 2.5 and 2.6 we can say that,

$$recv(f_{i+1}^x) \leq recv(f_i^x), \quad 1 < i < N_s \tag{2.7}$$

Applying equation 2.7 inductively, we get,

$$recv(f_{N_s}^x) \leq recv(f_2^x) \tag{2.8}$$

Using equation 2.5 with $i = 1$ on equation 2.8 gives,

$$recv(f_{N_s}^x) \leq sent(f_1^x) \tag{2.9}$$

Thus case (b) satisfies equation 2.2.

In case (c), the marker M_1 is received on an intermediate switch S_k. For switch S_k, $sent(f_k^x)$ is recorded before $recv(f_k^x)$. Thus

$$sent(f_k^x) \leq recv(f_k^x) \tag{2.10}$$

The path from switch S_1 to switch S_k is case (b) with switch S_1 as source and switch S_k as destination. The path from switch S_k to switch S_{N_s} is case (a) with switch S_k as source and switch S_{N_s} as destination. Thus Equations 2.9 and 2.4 give us,

$$recv(f_k^x) \leq sent(f_1^x) \tag{2.11}$$

$$recv(f_{N_s}^x) = sent(f_k^x) \tag{2.12}$$

Combining Equations 2.10, 2.11 and 2.12 we get:

$$recv(f_{N_s}^x) \leq sent(f_1^x) \tag{2.13}$$

which satisfies Equation 2.2. Thus, we have proved that OpenSnap gives consistent statistics when the marker is send to any switch in the network.

2.3.3.2 OpenFlow Based Network With Non-FIFO channels

In a Non-FIFO network, packets can be processed irrespective of their arrival order. In a Non-FIFO channel packet forwarding is generally implemented using a queueing mechanism [135] e.g., WFQ, PQ, custom queueing, linux-htb queueing discipline [136] etc. A Non-FIFO channel can be seen as a set of FIFO channels (which forward the packets in the order of their arrival), where each FIFO channel connects the queue q_i on source switch to the queue q_i on destination switch. To collect consistent statistics in a Non-FIFO network, we can run OpenSnap algorithm for each queue separately. The assumption is that every switch has the same number of queues and a flow is forwarded through the same queue across all the switches. For instance, for a given flow, each packet will be assigned the same priority and hence be assigned to the same queue in every switch if priority queueing is used. As already proved in the previous section, OpenSnap gives consistent statistics for a network with FIFO channels, and so the statistics obtained for all the flows going through the same queue in a network with Non-FIFO channels will be consistent.

2.4 Implementation Details

To implement OpenSnap, we have used POX [29] controller and Open vSwitch [54]. Open vSwitch is an open-source implementation of a distributed virtual multilayer switch which supports OpenFlow protocol. The marker packets used in our algorithm are recognized based on their destination MAC address. Marker M_1 has destination MAC address as "01:02:03:04:05:06" while marker M_2 has destination MAC address as "01:02:03:04:05:07". We have ensured that our test network does not contain any devices with these special MAC addresses.

To implement Algorithm 2.1, we need to add flow entries on the switch corresponding to marker M_1 and marker M_2. Each OpenFlow enabled switch has a flow table which has zero or more actions corresponding to each flow entry [18]. These actions dictate how to handle a packet matched with the given flow entry. Unfortunately, OpenFlow

Table 2.2: Flow entries corresponding to Marker M_1 and Marker M_2

Match	Action List
`dl_dst=01:02:03:04:05:06`	send_stats, `FLOOD`, `mod_dl_dst:01:02:03:04:05:07, IN_PORT`
`dl_dst=01:02:03:04:05:07`	send_stats

specifications do not have any action to send flow statistics. The OpenFlow specifications have messages for individual flow statistics and aggregate flow statistics. The controller sends flow statistics request to the switches and the switches sends the statistics reply back to the controller. This reply contains the statistics depending on the parameters provided in the request message. Using these messages as the basis, we have implemented a new action in Open vSwitch which sends the flow statistics to the controller. We call this action *send_stats*. "*send_stats*" checks if the marker received is M_1 or M_2. For M_1, the switch sends statistics for all the flow entries. For M_2, the switch sends statistics for the flows which have *in_port* value in the match field same as that of the ingress port of M_2.

The flow entries corresponding to the markers used in OpenSnap are given in Table 2.2. The first entry corresponds to marker M_1 and does the following in order; (1) Execute *send_stats* (2) Flood marker M_1 to all ports except the ingress port (3) convert marker M_1 to marker M_2 (4) Send marker M_2 through the ingress port. The second flow entry corresponds to marker M_2, it executes *send_stats* on receipt of marker M_2.

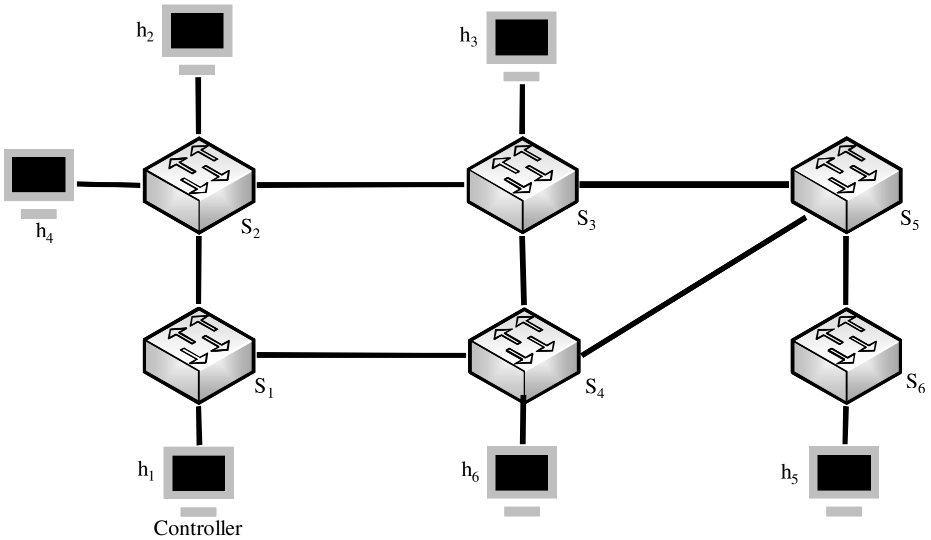


Figure 2.4: Topology for consistency evaluation.

2.5 Experimental Setup And Evaluation

We have used Mininet [137] to perform the experiments. Mininet by default has an out-of-band controller configuration, which means that every switch has a dedicated physical link with the controller. As this is not a practical behaviour in real networks, we expect an in-band controller to be running on one of the hosts. To handle this, we implemented an in-band controller in Mininet [137].

For consistency evaluation experiment, we have used the topology given in Figure 2.4. All links have a 100 Mbps bandwidth. We have three 3 UDP flows, f_1: (h_2,h_5), f_2: (h_4,h_6), and f_3: (h_5,h_3), in our network corresponding to the source-destination pair (h_2,h_5),(h_4,h_6), and (h_5,h_3) respectively. We use D-ITG [138] to generate the UDP traffic at the rate of 8 Mbps. We are generating 2000 packets every second each of size 512 bytes. The POX controller is running on host h_1 and it sends marker M_1 to switch S_1 to initiate statistics collection. Flow f_1: (h_2, h_5) and f_2: (h_4, h_6) trace the same path as marker M_1, whereas, flow f_3: (h_5, h_3) moves opposite to the marker M_1. For a given flow, we calculate the difference between the packets sent from the source switch and the packets received at the destination switch using the collected statistics. We call this difference "λ" and use this as our consistency measure. As per Equation 2.2 negative value of λ implies inconsistent statistics.

2.5.1 Network With FIFO Channels

We run OpenSnap, OpenNetMon [1], and Simple Polling with the same network configuration and traffic generation as discussed above in Section 2.5. Figure 2.5 (a) demonstrates the consistency results of OpenSnap and OpenNetMon [1] for each flow (i.e., f_1: (h_2,h_5), f_2: (h_4,h_6), and f_3: (h_5,h_3)). Figure 2.5 (b) demonstrates the consistency results of OpenSnap and Simple Polling for each flow. Clearly, both Simple Polling and OpenNetMon [1] give inconsistent statistics for flows f_1, and f_2. Whereas, they provide consistent statistics for flow f_3 because the destination host h_3 is connected to switch S_3 and the controller is running on host h_1 which is connected to switch S_1. So, when the controller initiated the statistics collection by sending statistics request message to the switches, for flow f_3, the destination switch S_3 sends the statistics before the source switch S_6. This is because for

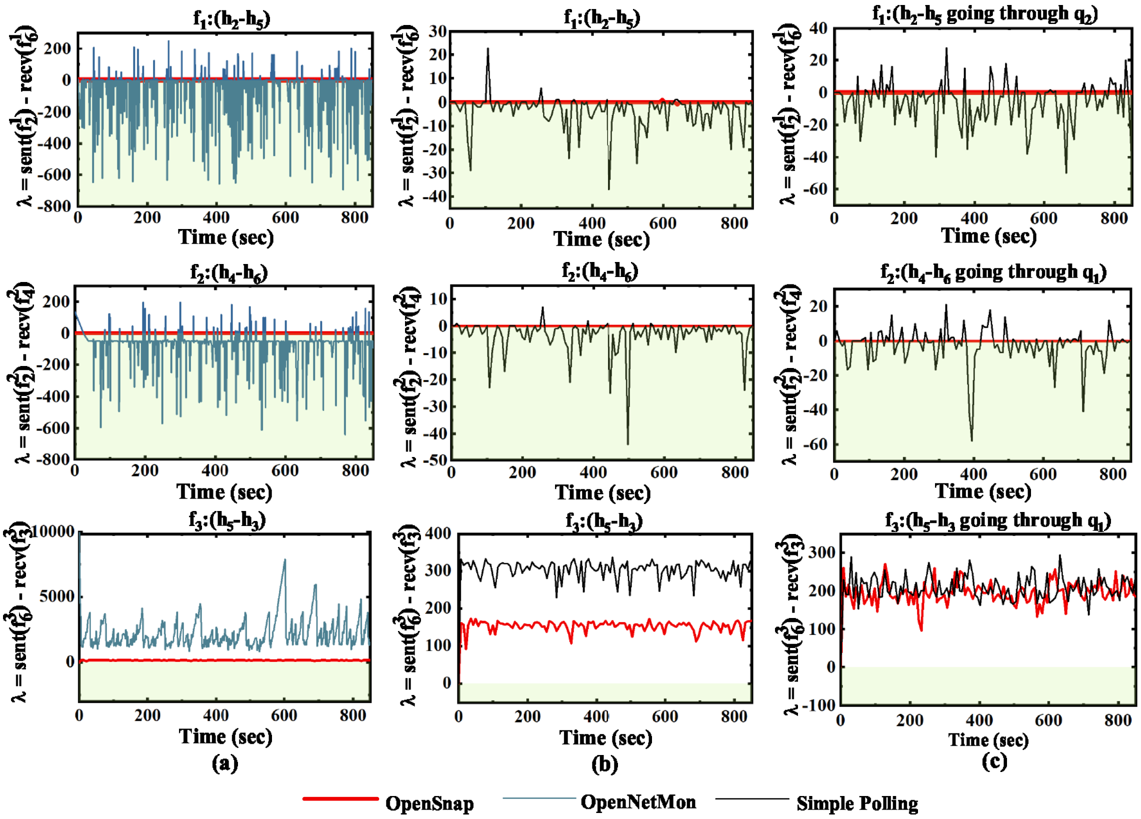


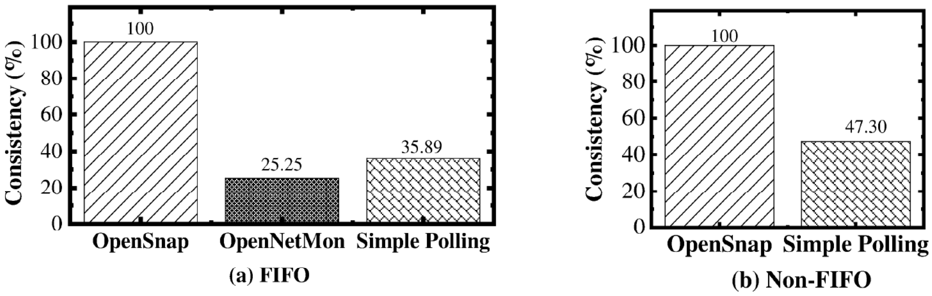
Figure 2.5: Comparing consistency of statistics (a) OpenSnap and OpenNetMon in FIFO network (b) OpenSnap and Simple Polling in FIFO network (c) OpenSnap and Simple Polling in Non-FIFO network. λ represents the difference between packets sent from the source switch and packets received at the destination switch. The shaded region represents area with inconsistent statistics

flow f_3 the source switch is located far from the controller as compared to the destination switch. So, by the time statistics request reaches source switch S_6 of flow f_3, the flow match counter would have increased. Thus, both Simple Polling and OpenNetMon [1] provide consistent statistics for flow f_3. While the statistics collected by OpenSnap are consistent for all flows.

We also compare all these solutions in terms of the percentage of consistency achieved. We define percentage of consistency achieved as the percentage of rounds providing consistent statistics out of the total number of rounds of statistics collection. The percentage of consistency is measured as follows,

$$\%\ consistency = \frac{number\ of\ rounds\ providing\ consistent\ statistics}{total\ number\ of\ rounds\ of\ statistics\ collection} * 100 \tag{2.14}$$

Figure 2.6 (a) shows the percentage of consistency achieved by each mechanism in a network with FIFO channels. As evident from Figure 2.6 (a), OpenSnap gives 100% consistent statistics, whereas, OpenNetMon [1] and Simple Polling mechanisms give only 25.25% and 35.89% consistent statistics respectively.



(a) FIFO

(b) Non-FIFO

Figure 2.6: (a) Percentage of consistency achieved by OpenSnap, OpenNetMon and Simple Polling in FIFO network. (b) Percentage of consistency achieved by OpenSnap and Simple Polling in Non-FIFO network.

2.5.2 Network With Non-FIFO Channels

Open vSwitch is a software switch, which uses a Linux kernel module for forwarding. To enable Non-FIFO packet processing behaviour, we have used *linux-htb* queueing discipline [136]. Every Open vSwitch has default queue called q_0. We have configured two more queues, q_1 and q_2, on each interface of the switches given in Figure 2.4. We have assigned 50 Mbps and 40 Mbps as minimum rate bandwidth for queue q_1 and queue q_2 respectively and the remaining 10 Mbps is given to queue q_0.

Flows f_2 and f_3 are forwarded through q_1 and flow f_1 is forwarded through q_2. We run OpenSnap algorithm for both queues q_1 and q_2. The controller initiates the OpenSnap algorithm by sending the marker M_1 to switch S_1 in both cases. Figure 2.5 (c) shows the consistency results of OpenSnap and Simple Polling for each flow. It is evident from the graph that OpenSnap gives consistent statistics for each flow. Whereas, Simple Polling provides consistent statistics only for flow f_3, because of the same reason explained above in Section 2.5.1. For flow f_3 the source switch is located far from the controller as compared to the destination switch. So, by the time statistics request reaches source switch S_6 of flow f_3, the flow match counter would have increased. Thus, Simple Polling provides consistent statistics only for flow f_3. As evident from Figure 2.6 (b), in a network with Non-FIFO channels, OpenSnap gives 100% consistent statistics whereas Simple Polling gives only 47.30% consistent statistics.

2.6 Summary

In this chapter, we discussed how the polling based network statistics collection mechanisms currently used in SDN fail to provide globally consistent statistics of a network and consequently degrade the effectiveness of various QoS provisioning applied to the network. To address this issue, we proposed OpenSnap, an algorithm to collect globally consistent statistics in SDN. We theoretically proved the correctness of OpenSnap and experimentally compared its performance in terms of consistency with existing mechanisms. The experimental results confirm that the statistics collected by OpenSnap are consistent and show that the amount of discrepancy in terms of the difference between the number of packets sent and received over various flows is always lesser than the existing mechanisms. In a network with Non-FIFO channels, the proposed solution requires multiple runs of the algorithm to generate a consistent view of a network. Also, the statistics collection process has to restart in case of an interruption. Thus the solution is not robust. In the next chapter we propose an efficient and robust solution to collect consistent statistics of a network.