Journal of Network and Systems Management (2021) 29:35
https://doi.org/10.1007/s10922-021-09601-z

GlobeSnap: An Efficient Globally Consistent Statistics Collection for Software-Defined Networks

Sandhya Rathee[1] · Nitin Varyani[2] · K. Haribabu[1] · Aakash Bajaj[1] · Ashutosh Bhatia[1] · Ram Jashnani[1] · Zhi-Li Zhang[2]

Received: 21 October 2020 / Revised: 14 February 2021 / Accepted: 30 March 2021 /
Published online: 3 May 2021

Abstract

Software defined networking (SDN) controller requires crucial statistics like flow-wise statistics from the switches to make decisions related to routing, load balancing, and QoS provisioning. These statistics, when viewed across the switches are likely to be inconsistent if a specific order is not enforced while collecting statistics. Collecting consistent statistics requires coordination among all the participating switches. A few approaches in the literature collect globally consistent statistics of a network in the SDN domain. However, these approaches are not time-efficient, robust, and synchronous for OpenFlow based networks. We propose, GlobeSnap, a time-efficient, robust, and synchronous method to collect globally consistent statistics for OpenFlow networks. GlobeSnap collects consistent statistics for all flows in a single round and is therefore, time-efficient. Moreover, GlobeSnap is robust since it resumes the statistics collection process from where it left in case of interruption. GlobeSnap also provides a near-synchronous snapshot of statistics of the switches traversed by a given flow. We also propose a mechanism to persistently store states in OpenFlow based networks using registers, multiple flow tables, and multiple pipelines. We find that GlobeSnap outperforms the state-of-the-art approaches in consistency evaluation. Further we present two use-cases which are sensitive to inconsistent flow statistics, that is, computing packet loss and identifying bottleneck links, to show the time-efficiency, robustness, and synchronicity of GlobeSnap. GlobeSnap provides 100% consistency in OpenFlow based SDN networks. Whereas the existing methods achieve a maximum of 59.89% consistency.

Keywords Consistent statistics · Special control packet · OpenFlow

✉ Sandhya Rathee
p2015007@pilani.bits-pilani.ac.in

Extended author information available on the last page of the article

1 Introduction

Software defined networking (SDN) is an emerging networking paradigm [1]. The SDN architecture is based on the separation of the control plane from the data plane. The control plane consists of a centralized entity called the SDN controller that controls the functioning of the distributed data plane. The data plane is a network of interconnected switches, which does traffic forwarding as per the policies defined by the SDN controller. To perform various network management tasks, the SDN controller needs to have an up-to-date and globally consistent snapshot of the network. These statistics are then used to estimate load on the links, to identify the bottleneck links, and to measure packet loss in the network. Accurate estimate of these parameters is essential to perform various network management tasks such as load balancing, QoS assurance [2], to meet the SLA (service level agreement) requirements etc.

The global state of the network is said to be consistent if a packet belonging to a flow is recorded as "received" at switch A then the same packet must have also been recorded as "sent" by all the preceding switches with respect to a flow. Failing to collect a consistent global snapshot can lead to the poor estimation of various network parameters such as queue depth, and load on links [3].

Prevalent network monitoring methods focus on per flow or per port statistics collection [4, 5]. 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 [6]. For example, consider a network as shown in Fig. 1, in which a packet are 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,

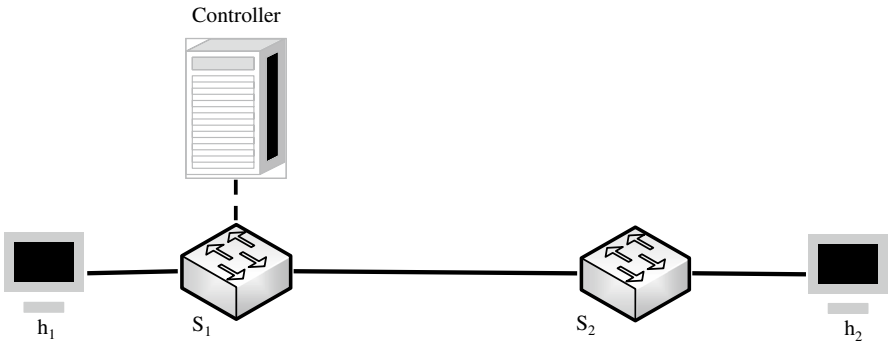


Fig. 1 Example to illustrate challenges in consistent statistics collection

- 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. In the first case, the occurrence of the events is in the order E_1, E_2, E_3, and E_4. In this case, 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. In the second case, the occurrence of the events is in the order E_3, E_1, E_2, E_4. That is, the packet P counted in the received statistics of switch S_2 but not counted in the sent statistics of switch S_1. Thus, it gives inconsistent statistics. In the third case, 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 recorded 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 [3] 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. The experimental results are provided in Appendix B. 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. We need a protocol to enforce the order of statistics collection such that it collects 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 effected 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 [7], applying updates consistently on all switches [8], dynamic visualization of network traffic patterns [9], a measurement that preserves this causal order is expected to yield accurate results.

There exist multiple solutions to collect statistics in SDN networks [3–6, 10–12]. In-band network telemetry (INT) [11] 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. In addition it requires a programmable data plane. There exist a few works that address

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

collection of consistent global state [3, 6]. In [6], the authors present a method, OpenSnap, to collect globally consistent statistics in OpenFlow networks with FIFO[2] and Non-FIFO[3] channels. Whereas, SpeedLight [3] proposes a solution for P4 networks, which works for both FIFO and Non-FIFO channels. In this paper, we are considering an OpenFlow based network with both FIFO and Non-FIFO channels. 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. Both OpenSnap [6] and Speedlight [3] propose a modified version of a traditional distributed systems algorithm "Chandy-Lamport" [13, 14]. Original Chandy-Lamport algorithm works only for FIFO channels. SpeedLight [3] instead of using a marker as in the Chandy-Lamport algorithm, inserts an additional packet header to every packet that carries the snapshot related information. SpeedLight [3] 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.

OpenSnap [6] provides consistent statistics for each flow in OpenFlow based network with FIFO channels. It further extends the solution to provide consistent statistics in OpenFlow based networks with Non-FIFO channels by sending the marker packet to one queue at a time in each round of statistics collection. However, this extension would provide consistent statistics only when a given flow in the network goes through the same output queue ID of all the switches in the path towards its destination, which may not be possible in every network. Also, in a given round it provides consistent statistics only for the flows going through the queue from which the marker packet is sent. This causes a delay in the collection of consistent statistics of all the flows. Also, OpenSnap is not a robust solution as it requires to restart the whole statistics collection process in case of an interruption (we explain this in more detail in Sect. 3.5.4).

We propose, GlobeSnap, a time-efficient, robust, and synchronous method to collect globally consistent statistics for OpenFlow networks which works for both FIFO and Non-FIFO channels. In one round OpenSnap can collect consistent statistics for all the flows going through a queue. Thus for a network with switches that have n queues per port it would take n rounds. Whereas GlobeSnap provides consistent statistics for all flows in a single round irrespective of the number of queues configured on a given switch. Thus GlobeSnap is time efficient in comparison to OpenSnap. Additionally, it does not assume that the data packets of a given flow have to go through the same queue ID on every switch in the path towards the destination.

[2] In *OpenFlow networks with FIFO channels*, the outgoing packets for transmission are scheduled based on order of their arrival at the switch.

[3] In *OpenFlow networks with Non-FIFO channels*, the outgoing packets for transmission could be scheduled irrespective of the order of their arrival.

Moreover, GlobeSnap is a robust solution as it does not require to restart the whole statistics collection process in case of an interruption. It resumes the statistics collection process from where it left. GlobeSnap also provides a near-synchronous snapshot of statistics of the switches traversed by a given flow. The synchronicity of a snapshot is a measure of how contemporaneously switches can record their local snapshots. A near-synchronous snapshot of a network is one in which all switches record their local snapshots almost simultaneously. This is difficult to achieve in practice in a distributed system and if proper care is not taken packets may be reported as received but may not be reflected as sent, thus violating consistency. GlobeSnap, while ensuring consistency, can also provide a near-synchronous snapshot of a flow.

Communication between SDN controller and the underlying switches can happen in two ways: out-of-band and in-band. In an out-of-band configuration, the switches are directly connected to the SDN controller through dedicated links. There are some advantages of out-of-band configuration like, the communication is more secure, low communication delay between the switches and SDN controller [15]. 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. In this paper, we are considering an in-band controller configuration.

The major contributions of the paper are as follows,

1. We propose, GlobeSnap, a time-efficient, robust, and synchronous method to collect globally consistent statistics for OpenFlow based networks.
2. We also propose a mechanism to persistently store the states in OpenFlow based networks using registers, multiple flow tables and multiple pipelines.
3. We evaluate the consistency guarantees of GlobeSnap using Mininet [16] testbed and also compare with the state-of-the-art approaches like OpenSnap [6], OpenNetMon [5] and CeMon [4].
4. We also present two use-cases which are sensitive to inconsistent flow statistics that is, computing packet loss and identifying bottleneck links, to show the time-efficiency, robustness and synchronicity of GlobeSnap.

The rest of the paper is organised as follows: In the next section we discuss the existing works related to statistics collection in SDN networks. We discuss our system model and the problem with the marker-based solution in Sect. 3.1 and 3.3 respectively. In Sect. 3.4, we discuss our protocol. In Sect. 3.5, we discuss the characteristics of GlobSnap. In Sect. 3.6 we compare the existing solutions with ours in terms of overhead incurred. In Sect. 4, we provide the implementation details followed by the experimental evaluations in Sect. 5. In Sect. 6, we conclude our work. In Appendix A we also provide correctness proof for our solution.

2 Related Work

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

OpenNetMon [5] 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 [17] 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 [4] 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 best 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 [18], is a better version of NetFlow [19]. 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 [20] 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.

Table 1 Table summarizing the consistent statistics collection works

Paper/method	Provide consistent statistics	Robust	Time efficient
GlobeSnap	✓	✓	✓
OpenSnap [6]	✓	✗	✗
SpeedLight [3]	✓	✓	✗

PayLess [10] 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 [12] 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 [21] 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 [22], 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.

As already discussed in Sect. 1, SpeedLight [3] provides consistent port statistics. However, it may not provide consistent statistics in every round of statistics collection. In case of an inconsistency in the collected statistics, the controller has to run the protocol again. OpenSnap [6], provides consistent flow statistics in both OpenFlow based network with FIFO channels and OpenFlow based network with Non-FIFO channels. OpenSnap takes multiple rounds to collect consistent statistics of all the flows in a network with Non-FIFO channels. Also, the solution is not robust because it has to restart the whole statistics collection process in case of interruption. Both SpeedLight [3] and OpenSnap [6] may require multiple rounds to collect consistent statistics. Thus they are not time-efficient (Table 1). To handle real time applications the controller should take quick decisions. This requires updated globally consistent view of the network. A delay in consistent statistics collection could degrade the network performance. In this paper, we propose an efficient and robust solution to collect consistent flow statistics in OpenFlow based network.

3 GlobeSnap Method

In this section, we define the system model, issues with marker-based mechanisms, and the proposed GlobeSnap method.

3.1 System Model

We consider an SDN network with OpenFlow 1.3 compatible switches. Figure 2a

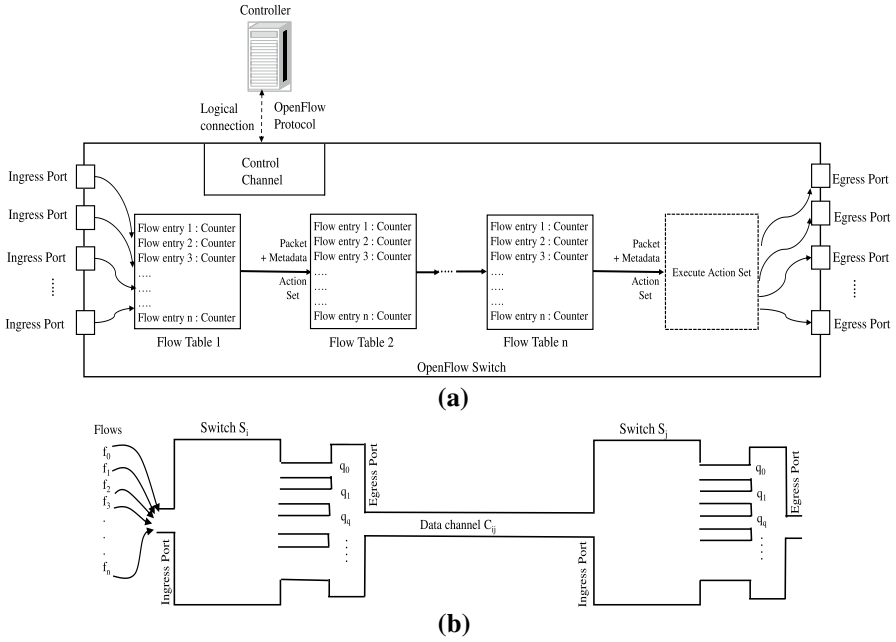


Fig. 2 **a** OpenFlow Switch, **b** detailed diagram of two switches directly connected to each other

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 [23]. Figure 2b shows two switches S_i and S_j connected to each other through a data channel C_{ij}. The network supports different forwarding classes i.e., flows are assigned to different queues based on their priority or QoS requirement. 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. The controller is an in-band controller i.e., the controller is not connected to each switch through a dedicated link. The controller is just like any other host in the network. When a new flow enters the network, if there is no corresponding flow entry then the first packet of the flow is forwarded to the controller as a PacketIn message and the controller inserts the necessary flow entries into

the switches. Flow entries are stored in the flow tables. There can be multiple flow tables in an OpenFlow switch. Depending on the flow entries, a packet may be processed through multiple tables. We assume that switches support metadata as specified in OpenFlow 1.3 specifications [24]. This metadata is used to store the data in one flow table, which can be accessed in another table.

3.2 What are Globally Consistent Statistics?

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[4] 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, $S = \{S_1, S_2, S_3, \ldots, S_N\}$, and F be the set of flows in the network, $F = \{f_1, f_2, f_3, \ldots, f_X\}$. 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,

$$sent(f_i^k) = recv(f_j^k) + Q_{iq}^k + C_{ij}^k + drop(f_i^k), \tag{1}$$

where C_{ij}^k is the number of packets of kth flow present in the channel connecting switch S_i and switch S_j, Q_{iq}^k is the number of packets of kth flow queued in qth, $1 \leq q \leq I$, 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, Eq. 1 can be written as,

$$sent(f_i^k) \geq recv(f_j^k). \tag{2}$$

3.3 Issues with Marker-Based Mechanism

In this section, we show why the marker-based consistent statistics collection method proposed in [6], which sends the marker through only one queue of the switch, fails to collect consistent global state in case of OpenFlow [25] networks with Non-FIFO channels. In marker-based method, the controller initiates the statistics collection process by sending a marker packet in the network. The switches send the statistics only when they receive a marker packet [6]. Consider a network

[4] We use channel and link interchangeably in this paper.

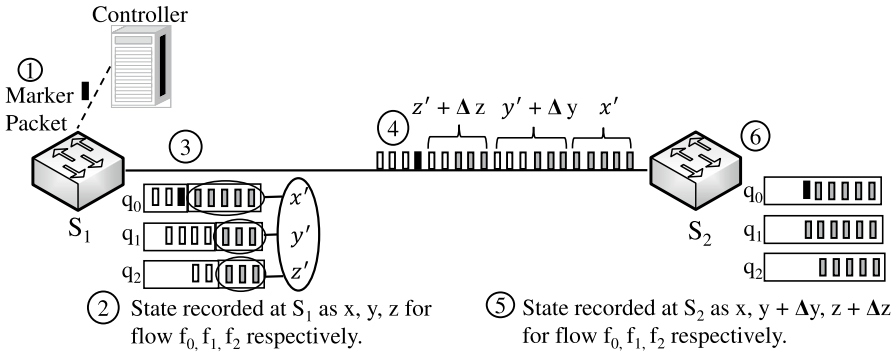


Fig. 3 Illustrating the limitation of OpenSnap for OpenFlow based networks with Non-FIFO channels

of two switches, as shown in Fig. 3. There are three queues configured on both the switches. Let there be three flows f_0, f_1, and f_2 being forwarded through queues q_0, q_1, and q_2 respectively. Let the controller send the marker packet to switch S_1 (1). On receiving the marker packet switch S_1 sends the statistics corresponding to the flows f_0, f_1, and f_2 as x, y, and z respectively to the controller (2). Note that x, y, and z also contain the packets which are already scheduled and waiting in their respective queues to be transmitted. This marker packet is enqueued in one of the queues (say queue q_0) behind the packets that are already present in the queue q_0 (3). Now suppose that the scheduler sends a few packets from queue q_0 of switch S_1, which were already present in the queue q_0 at the time when the marker packet was enqueued. Let x' be the number of such packets. Let Δy be the number of packets arrived in queue q_1 after the arrival of the marker packet in queue q_0. Now suppose that the scheduler sends $y' + \Delta y$ packets on the data channel, where y' is the number of packets that were already present in the queue q_1 when the marker packet was enqueued in queue q_0. Note that, the marker packet is still waiting in the queue q_0. This can happen because of the Non-FIFO nature of the SDN switches.

Let Δz be the number of packets that arrived in queue q_2 after the arrival of the marker packet in queue q_0. Now suppose that the scheduler sends $z' + \Delta z$ packets on data channel from queue q_2, where z' is the number of packets that were already present in the queue q_2 when the marker packet was enqueued in queue q_0. Now the scheduler sends the marker packet on the data channel (4). Note that the marker packet will reach switch S_2 after all the packets of flows f_0, f_1, and f_2 which were scheduled on the data channel before the marker packet. When the marker packet hits the switch S_2, it sends the statistics for all three flows f_0, f_1, and f_2 as x, $y + \Delta y$, and $z + \Delta z$ respectively to the controller (5) and forwards the marker packet through queue q_0 (6). OpenSnap gives inconsistent statistics for flows f_1 and f_2, as sent statistics minus received statistics is -Δy, and -Δz respectively. That is, Δy and Δz packets for flows f_2 and f_3 respectively are recorded as received at destination switch but not recorded as sent at source switch.

3.4 GlobeSnap Protocol

3.4.1 Overview

Our idea in this paper is inspired by an algorithm by Lai-Yang [26] proposed for construction of a global snapshot in distributed systems with Non-FIFO channels. This algorithm uses a color scheme to overcome the drawbacks of previous approaches. Data packets and switches are marked as red or white, depending on the following conditions. Initially, all switches (processes) are in white state and they turn red when they receive a red data or flow statistics request packet. The packet sent by a white (or red) switch is colored white (or red).

Though the idea is inspired by Lai-Yang [26], there are a few differences conceptually and in implementation when applied to OpenFlow networks. (i) The original algorithm assumes that traffic of red packets goes through all the processes in the network. This may not be true in a network of switches. A flow colored red may not convert all the switches in the network to red. We adapt the algorithm to take care of this. (ii) The original algorithm is designed to collect a global snapshot for a single round. We make changes to the algorithm to collect global snapshots in a continuous manner using multiple colors. (iii) The original algorithm mandates keeping the history of the messages to compute the channel state correctly. This is not required when applied to OpenFlow networks as the statistics are maintained cumulatively in the switches. (iv) The original algorithm depends on the process turning red atomically when the snapshot is recorded. This is non-trivial to implement in a network of switches. We make use of flow tables in OpenFlow to implement this requirement. (v) The original algorithm requires storage space to store messages received after the snapshot is recorded. OpenFlow switches do not support user-specified values in-memory or disk storage. We use a combination of flow rule priorities, flow rules, and metadata to achieve this.

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 colored packet different from its own state. But OpenFlow (as for the current OpenFlow [25] Standard) does not have any action which sends the flow statistics on the arrival of a particular packet. Thus, to solve this issue, we extend the OpenFlow protocol by implementing a new action called "send_stats" in Open vSwitch [27]. On arrival of the first colored 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. This is the case when the controller is collecting statistics (recording snapshot) for all the flows in the network. Assuming the network has X number of flows. Let F be the set of flows, $\mathrm{F} = \{f_1, f_2, f_3, \ldots, f_X\}$. In case the controller wants to poll a subset, $\mathbb{F} \subseteq F$, of the flows then it has to wait for the statistics replies only from the switches processing the flows in the subset $\mathbb{F}$. In order to enable statistics collection in multiple rounds, the controller must use a different color for special control packet differing from the current state of the switches. This is to differentiate between the packets in-transit and the special control packet for the next round. To satisfy this requirement

the controller needs at least two colors for special control packet. In the proposed method the controller uses red and green colors for special control packets alternatively to minimize the cool-down period between subsequent rounds. Detailed explanation is provided in Sect. 3.4.4.

We divide the algorithm into two parts, one executing at the data plane or switch level and the other executing at the control plane or SDN controller level. Data plane is responsible for taking a copy of flow statistics, sending the statistics to SDN controller, and changing the state of the switch. At a given time, a switch can be in one of the three possible states: WHITE, RED, or GREEN. Initially, at the start, all the switches are considered to be in WHITE state. Any packet going out of the processing pipeline in the switch is colored with the same color as that of the switch state. Incoming packets on the ingress ports can be of one of the three colors: white, red, or green. Initially, all packets are considered to be of white color. When a switch receives a colored packet (special color packet or data packet) different from its state, it sends the statistics of all the flows to the controller. To start the network-wide statistics collection, the SDN controller sends a special control packet to one of the switches, with a color different from the color of the switches in the network. This triggers state changes in the rest of the switches as the packets move around in the network. Sometimes, packets with the changed color may not reach certain portions of the network. SDN controller with its global knowledge of the flows and their paths in the network, can identify the disconnected portions of the networks, and send a special control packet to one of the switches in each of the disconnected network portions.

3.4.2 An Example to Illustrate GlobeSnap Method

Consider the same example which we used in Sect. 3.3 to demonstrate the limitation of marker-based solution. Consider the same set up with two switches as shown in Fig. 4a. There are three queues configured on both the switches. Let there be three flows f_0, f_1, and f_2 being forwarded through queue q_0, q_1, and q_2 respectively. Let the controller send a special color packet (let's say red packet) to switch S_1, initially in WHITE state, to initiate the statistics collection process (1). When the red packet hits the switch S_1, it sends the statistics of all three flows f_0, f_1, and f_2 as x_1, y_1, and z_1 respectively to the controller (2). Once the switch S_1 sends its statistics to the controller, it changes its state from WHITE to RED (3). Now, traffic going out of switch S_1 is colored red (4). Now suppose the scheduler sends $x_1^{'}$ packets on data channel from queue q_0, where $x_1^{'}$ is the number of packets of flow f_0 that arrived in the queue q_0 before sending the statistics of switch S_1 to the controller. Let Δy_1 and Δz_1 be the number of packets of the flows f_1 and f_2 respectively that arrived in queues q_1 and q_2 respectively after switch S_1 has sent its statistics to the controller. Thus, these Δy_1 and Δz_1 packets are colored red. Now suppose the scheduler sends $y_1^{'} + \Delta y_1$ packets from queue q_1 on data channel, where $y_1^{'}$ is the number of packets of flow f_1 that arrived in queue q_1 before switch S_1 sent its statistics to the controller. Now the scheduler sends $z_1' + \Delta z_1$ packets from queue q_2 on data channel, where $z_1^{'}$ is the number of packets of flow f_2 that arrived in the queue q_2 before switch S_1 sent its statistics to the controller (5). When the first red packet of flow f_1 (i.e., the first

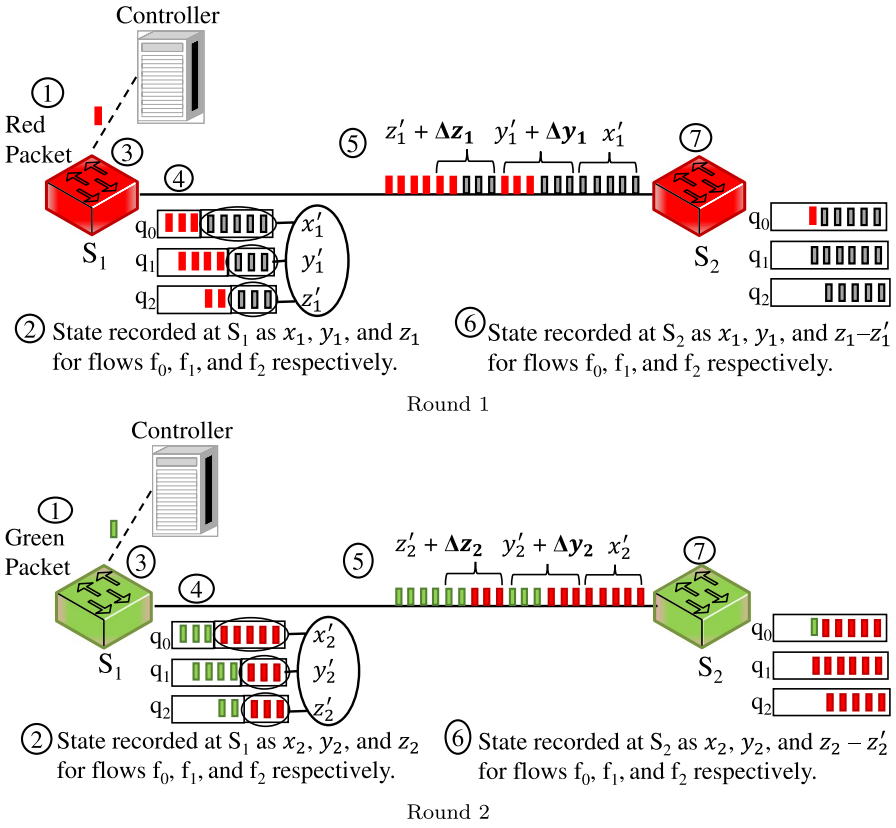


Fig. 4 An example to illustrate the working of GlobeSnap on a link between two switches

packet of Δy_1) hits switch S_2, it sends the statistics for all three flows f_0, f_1, and f_2 as x_1, y_1, and $z_1 - z_1^{'}$ respectively to the controller (6). After sending the statistics to the controller, the switch changes its state from WHITE to RED (7). Sent statistics at switch S_1 for flows f_0 and f_1 is equal to received statistics of flows f_0 and f_1 at switch S_2. Whereas, for flow f_2 the sent statistics is greater than the received statistics (i.e., $z_1 > z_1 - z_1^{'}$). Thus, it satisfies the consistency condition given in Eq. 2 and gives consistent statistics for all three flows. A list variables used in this section is given in Table 2.

One round of statistics collection is complete when the controller gets the statistics from all the switches in the network. In the second round the controller sends a special color packet of color different from the first round (the reason for this is explained later in Sect. 3.4.4). Let's assume that in second round the controller send a special color packet of green color to switch S_1 to initiate the statistics collection process (1) (as shown in Fig. 4b). When the green packet hits the switch S_1, it sends the statistics of all three flows f_0, f_1, and f_2 as x_2, y_2, and z_2 respectively to the controller (2). OpenFlow switches maintain cumulative counters for each flow entry.

Table 2 List of variables used in Sect. 3.4.2

Variable	Meaning
q_0, q_1, q_2	Queues configured on the output port of each switch
x_i, y_i, z_i	Sent statistics of flows f_0, f_1, and f_2, respectively in i th round of statistics collection w.r.t source switch and received statistics w.r.t destination switch
$x_i^{'}$	Number of packets of flow f_0 that arrived in the queue q_0 before sending the statistics of switch S_1 in ith round of statistics collection
$y_i^{'}$	Number of packets of flow f_1 that arrived in queue q_1 before switch S_1 sent its statistics to the controller in ith round of statistics collection
$z_i^{'}$	Number of packets of flow f_2 that arrived in the queue q_2 before switch S_1 sent its statistics to the controller in ith round of statistics collection
Δy_i	Number of packets of the flows f_1 that arrived in queues q_1 after switch S_1 has sent its statistics to the controller in ith round of statistics collection and are colored red/green
Δz_i	Number of packets of the flows f_2 that arrived in queues q_1 after switch S_1 has sent its statistics to the controller in ith round of statistics collection and are colored red/green

Thus, switches send cumulative counters as statistics reply. x_2 is the total number packets matched to flow entry corresponding to flow f_0 when the green packet hits the switch S_1. That is, x_2 includes x_1 (the number packets match to flow entry corresponding to flow f_0 in first round). Similarly, y_2 includes y_1 and Δy_1, and z_2 includes z_1 and Δz_1.

Once the switch S_1 sends its statistics to the controller, it changes its state from RED to GREEN (3). Now, traffic going out of switch S_1 is colored green (4). Now suppose the scheduler sends $x_2^{'}$ packets on data channel from queue q_0, where $x_2^{'}$ is the number of packets of flow f_0 that arrived in queue q_0 before switch S_1 sent its statistics to the controller. Let Δy_2 and Δz_2 be the number of packets of flows f_1 and f_2 respectively that arrived in queues q_1 and q_2 respectively after switch S_1 has sent its statistics to the controller. Thus, these Δy_2 and Δz_2 packets are colored green. Now suppose the scheduler sends $y_2^{'} + \Delta y_2$ packets from queue q_1 on data channel, where $y_2^{'}$ is the number of packets of flow f_1 that arrived in the queue q_1 before switch S_1 sent its statistics to the controller. Now suppose the scheduler sends $z_2^{'} + \Delta z_2$ packets from queue q_2 on data channel where $z_2^{'}$ is the number of packets of flow f_2 that arrived in the queue q_2 before switch S_1 sent its statistics to the controller (5). When the first green packet of flow f_1 (i.e., the first packet of Δy_2) hits the switch S_2, it sends the statistics for all three flows f_0, f_1, and f_2 as x_2, y_2, and $z_2 - z_2^{'}$ respectively to the controller (6). After sending the statistics to the controller, the switch changes its state from RED to GREEN (7). Sent statistics at switch S_1 for flows f_0 and f_1 is equal to received statistics of flows f_0 and f_1 at switch S_2. Whereas, for flow f_2 the sent statistics is greater than the received statistics (i.e., $z_2 > z_2 - z_2^{'}$). Thus, it satisfies the consistency condition given in Eq. 2 and gives consistent statistics for all three flows. In every alternate round, the controller sends a special control packet of the same color to initiate the statistics collection process.

3.4.3 Algorithm at the Controller

We represent the network by a bi-directed graph G = (V,E), where V = $\{s_1, s_2, s_3, \ldots, s_N\}$ is the set of switches in the network and E is the set of bi-directed edges representing the physical links between two switches. Let F be the current set of flows in the network. We create a directed graph FlowG = (V,E$'$) corresponding to the set F such that E$'$ is the set of directed physical links between two switches which carry traffic of at least one flow in the set F.

Initially all the switches are in WHITE state and the traffic in the network is also white. In every round of statistics collection, the controller sends the FlowMod command to all the switches to update the current round color (line 3–5 of Algorithm 1). The controller initiates the statistics collection process by sending a special control packet which is like any other data packet but colored[5] with the same color as the round color. Initially the round color is initialized to red color (line 1 of Algorithm 1).

Algorithm 1: Controller Side

```
   // RoundColor holds the Color of the current run
1  round_color ← RED;
2  foreach round do
       // N is the total number of switches in the network
3      foreach i in {1,2,...,N} do
           // sends command to a switch to update the current round color
4          COMMAND_SET_ROUND_COLOR(v_i, round_color);
5      end
6      RemainingSet ← FlowG ;
7      while RemainingSet ≠ φ do
           // Returns a switch with maximum flows fanning out
8          s ← MAX_OUT_DEGREE_VERTEX(remainingSet);
           // Breadth-First-Search routine returns set of all vertices
              reachable from node s
9          ReachableSet ← BFS(FlowG,s);
           // sends a special color packet to the root switch of a subtree
10         SEND_CONTROL_PACKET(s, round_color);
11         RemainingSet ← RemainingSet − ReachableSet;
12     end
       // Round terminates when it receives stats from all switches
13     foreach i in {1,2,...,N} do
           // receives flow statistics from a switch
14         All_stats[i] = RECEIVE_FLOW_STATS_SWITCH(i) ;
15     end
16     if round_color == RED then
17         round_color ← GREEN;
18     else
19         round_color ← RED ;
20     end
21 end
```

The objective of sending a special control packet to a particular switch is that it should eventually spread to all switches. This may not happen always. There can

[5] To color a packet, ECN field of IP header is used. ECN field has 2 bits, therefore there can be 4 ways of using it.

be two or more groups of switches where the flow in one group does not reach another group. Therefore, we choose the switch with maximum out-degree in the graph FlowG (line 8 of Algorithm 1) and find reachable nodes from it using Breadth First Search (BFS) (line 9 of Algorithm 1). This gives us a disconnected tree. We send a special control packet to the root switch of this tree (line 10 of Algorithm 1). Then we remove this tree from the FlowG (line 11 of Algorithm 1) and repeat this process on the remaining graph until all the switches in the network are covered. The controller waits for statistics from all N switches (line 13–15 of Algorithm 1). Once the controller receives the statistics from all the switches, it concludes the one round of statistics collection and updates the round color for next run (line 16–20 of Algorithm 1).

3.4.4 Need of Two Colors for Switch State

One round of statistics collection is complete when the controller receives the statistics from all the switches in the network (line 13–15 of Algorithm 1) and thus network becomes RED. For the next round, the state of all the switches needs to be reset back to WHITE. After each round of statistics collection, the controller sends the command to the switches to reset their state. Only after successful reset of the state of all the switches, the network comes back again in WHITE state and becomes ready for the next round. But the above approach has a limitation. By the time a round of statistics collection completes the whole network has turned RED and all the switches can not be reset back to WHITE state simultaneously. In a network with in-band controller configuration, some switches will be near to the controller and some switches will be some hops away. The switches nearer to the controller are reset faster compared to those which are far from the controller. Thus, the switches which are far from the controller and have not been turned back WHITE will keep on generating red packets. These stray red packets can hit the switches which the controller has reset for the next round. So, the controller can not start a new round until there is a red packet in the network. This increases the delay in statistics collection for the next round. Thus, to resolve this limitation, we propose use of two different color packets. Now for the second round of statistics collection, the controller sets the current round color different from the first round, say green (line 17 of Algorithm 1). The switches will consider all the packets of color different from current round color as normal packets and send the statistics for the current round only on reception of a green packet. Thus, the controller does not need to wait for all the switches to reset back to WHITE state before starting the next round statistics collection. After the end of second round of statistics collection the whole network turns GREEN. So, for the third round the controller can use a special control packet of RED color and so on. Thus, the proposed method requires at least two special control packets of different colors.

3.4.5 Algorithm at the Switch

The switch has multiple forwarding tables as shown in Fig. 2a. Each table has multiple flow entries. Counters for each flow entry are maintained in the switch and

these counters get updated when a packet matches the flow entry. The switch sends the value of these counters as statistics reply to the controller on arrival of a colored packet.

Algorithm 2: SwitchSidePacketProcessingProcedure(Packet P)

```
1  CurrentRoundColor ← none;
2  SwitchState ← WHITE;
3  if P.Type == CommandToUpdateColor then
       // Command from SDN Controller to setup new round
4      CurrentRoundColor ← P.RoundColor;
5  else
       // Normal data packet
6      if P.Color == CurrentRoundColor then
           // Packets which have seen the current round or Special Control
              Packet
7          if SwitchState == CurrentRoundColor then
               // Flow Statistics already sent to the controller
8              Forward Packet P;
9          else
               // Statistics not sent
10             Send statistics of all the flows to the controller;
11             SwitchState ← CurrentRoundColor;
12             Forward Packet P ;
13         end
14     else
           // Packets which have not seen the current round
15         if SwitchState = CurrentRoundColor then
16             P.Color ← CurrentRoundColor;
17             Forward Packet P;
18         else
19             Forward Packet P ;
20         end
21     end
22 end
```

The execution of algorithm on a switch in the data plane depends on three things:*CurrentRoundColor*, *SwitchState*, and input packet. *CurrentRound-Color* is used for starting a new round and *SwitchState* is used for making the switch remember whether it has sent the statistics for the current round or not. *CurrentRoundColor* is updated by the controller to start a new round and *SwitchState* is set by the switch itself. It is not straightforward to implement this in an OpenFlow enabled switches. We provide more details about this in Sect. 4. The input to Algorithm 2 *SwitchSide-PacketProcessingProcedure*() is a packet. The packet can be a data packet (default color is white) or a special control packet (red or green) from SDN controller to initiate the statistics collection process or a command sent by SDN controller to update the current round color. When the switch receives a command to update the round color it updates the current round color (line 3–4 of Algorithm 2). As discussed above, initially all the switches are in WHITE state and the traffic in the network is also white. The controller initiates the statistics collection process by sending a red or green special control packet. We call the first round of statistics collection as

bootstrapping round. Packet processing is slightly different for bootstrapping round when compared to subsequent rounds.

During bootstrapping round, packet processing at a given switch w.r.t the input packet's color and current state of the switch is given in Table 3.

Table 3 Packet processing at a switch during bootstrapping round

Input	Switch state	Action
Red packet	WHITE	The switch sends the statistics to the controller, updates its state to RED, and forwards the packet (line 9–13 of Algorithm 2)
Red packet	RED	The switch will not send the statistics to the controller and simply forwards the received packet (line 7–8 of Algorithm 2)
White packet	RED	The switch will not send the statistics to the controller. However, it will color every incoming white packet as red and forwards it (line 15–17 of Algorithm 2)

Table 4 Packet processing at a switch during subsequent rounds

Input	Switch state	Action
Packet of *CurrentRoundColor*	{WHITE, GREEN, RED} - {CurrentRoundColor}	When the switch state is not CurrentRoundColor and it receives a packet of same color as CurrentRoundColor, the switch sends the statistics to the controller, updates its state to CurrentRoundColor, and forwards the packet (line 9–13 of Algorithm 2)
Packet of *CurrentRoundColor*	CurrentRoundColor	Once the switch is in state same as CurrentRoundColor, the switch will not send the statistics to the controller and simply forwards the received packet (line 7–8 of Algorithm 2)
Packet of color = {white,green,red}-{*CurrentRoundColor*}	CurrentRoundColor	Once the switch is in state same as CurrentRoundColor, the switch will not send the statistics to the controller, it colors every incoming packet with CurrentRoundColor and forwards the packet (line 15–17 of Algorithm 2)

In subsequent rounds, packet processing at a given switch w.r.t the input packet's color and current state of the switch is given in Table 4.

3.5 Characteristics of GlobeSnap

Consider two adjacent switches S_i and S_j in a connected subgraph of a network. There are many flows going through these two switches such as forward flows F, reverse flows R, and orthogonal flows O. A forward flow, w.r.t switch S_i, goes from switch S_i to switch S_j. A reverse flow, w.r.t switch S_i, goes from switch S_j to switch S_i. An orthogonal flow goes either through switch S_i or switch S_j but not both.

3.5.1 Consistent Snapshots

A forward flow f_k when it goes through several switches on its path, at any given instant, for switches S_i and S_j, $sent(f_i^k) \geq recv(f_j^k)$. This is due to the fact that any packet received at switch S_j, should have been sent from switch S_i. This is known as causal consistency. GlobeSnap ensures consistency according to this relation. Let us now understand how it works.

Case 1: forward flow carrying red packets In GlobeSnap, once a red packet is received at switch S_i, statistics of all the flows at switch S_i are sent to the controller and any further packet is colored as red. As soon as any of these red packets reach switch S_j, statistics of all the flows of switch S_j are sent to the controller. This avoids the possibility of a packet not recorded in the statistics of switch S_i reaching switch S_j before switch S_j sends its statistics. This is formally proven in Appendix A.

When a forward flow carries red packets, statistics recorded at switch S_i and switch S_j also contain the statistics of reverse flows. For a reverse flow r_k, $sent(r_j^k) \geq recv(r_i^k)$ is true. This is because, by the time the first red packet reaches to switch S_j, zero or more packets would have been transmitted from switch S_j to switch S_i for flow r_k. Therefore, the sent statistics of flow r_k at switch S_j will be more than or equal to received statistics of flow r_k at switch S_i. Consider the following example, let at time t_1 the flow match counter values of flow f_k and r_k at switch S_i are x_1 and y_1, respectively. Now a red packet arrives at switch S_i and the switch S_i sends statistics of both the flows to the controller as $sent(f_i^k) = x_1$, and $recv(r_i^k) = y_1$. Let after Δ time a red packet of flow f_k from switch S_i arrives at switch S_j. This will invoke statistics collection at switch S_j. During this Δ time interval switch S_j may or may not have sent packets to switch S_i for flow r_k. Thus, for flow r^k the statistics recorded at switch S_j will be greater than or equal to the statistics recorded at switch S_i.

Case 2: orthogonal flows carrying red packets Consider that two orthogonal flows o_i, o_j carry red packets and these red packets reach switch S_i and switch S_j at T_i and T_j, respectively. Also consider that $T_i < T_j$ and during the time $|T_i - T_j|$ no red packet is exchanged between the switches. At time T_i red packet from orthogonal flow o_i reaches switch S_i and it sends the statistics of both the flows f_k and r_k as $sent(f_i^k)$ and $recv(r_i^k)$, respectively. In $|T_i - T_j|$ time only the white packets are sent from switch S_i to switch S_j. All these white packets would either have reached switch S_j or still be in the data channel. Thus when switch S_j receives a red packet of flow o_j at time T_j, the recorded statistics of flow f_k i.e., $recv(f_j^k)$ will be less than or equal to sent statistics from switch S_i i.e., $sent(f_i^k)$. Similarly for the reverse flow r_k, during the time $|T_i - T_j|$ switch S_j may or may not have sent packets to switch S_i. Thus, the recorded statistics for flow r_k at switch S_j i.e., $sent(r_j^k)$ is greater than or equal to the received statistics at switch S_i i.e., $recv(r_i^k)$. Similarly the consistency condition will also hold when $T_i > T_j$.

3.5.2 Near-synchronous Snapshot

Synchronicity of a snapshot is a measure of how contemporaneously switches can record their local snapshots. A synchronous snapshot of a network is one in which

all switches record their local statistics simultaneously. This is difficult to achieve in practice in a distributed system because if proper care is not taken packets may be reported as received but may not be reflected as sent thus violating consistency. GlobeSnap while ensuring consistency as described above, provides a near-synchronous snapshot of a flow. If $T_1, T_2, \ldots, T_N$ are the timestamps at which switches $S_1, S_2, ..., S_N$ have recorded local statistics of a flow f_k then GlobeSnap ensures that $T_{max} - T_{min} \leq RTT(f_k)/2$ where T_{max} is maximum timestamp and T_{min} is minimum timestamp and $RTT(f_k)$ is round trip time measured from S_1 to S_N of f_k. This is possible because red packet initiated at S_1 can reach S_N in $RTT(f_k)/2$ time.

3.5.3 Efficient and Flexible Recording of Snapshot

In a network, let us assume that it takes t_F time for a packet to reach the farthest switch from the SDN controller and t_R time in reverse direction.

As already explained in Sect. 3.3, OpenSnap provides inconsistent statistics for OpenFlow based networks with Non-FIFO channels. In a given round of statistics collection, OpenSnap sends marker packet through a queue say q_i, and can ensure consistency only for the set of flows going through queue q_i. Every round takes $t_F + t_R$ time, t_F time for marker packet to reach the farthest switch and t_R is the time taken by the farthest switch to send the statistics reply to the controller. In order to cover all flows, marker has to be sent separately through each queue, each taking a round of its own. If there are Q number of queues supported on all outgoing interfaces in the network, OpenSnap needs $Q \times (t_F + t_R)$ time to collect the statistics of all the flows for single snapshot. On the other hand, GlobeSnap collects statistics of all flows in a single round taking $t_F + t_R$ time. Therefore, GlobeSnap is more efficient than OpenSnap.

In GlobeSnap, t_F is the time taken by red packet to reach the farthest switch and t_R is the time taken by the farthest switch to send statistics to the controller. The component t_F can be further reduced, if red packets are introduced in more than one switch in the connected subgraph. This leads to faster spread of red packets reducing t_F time. t_F will be zero in an ideal case where red packets are introduced in all switches at the same time. Depending on how many special color packets are introduced in the network, to that degree, t_F will be lesser. However, t_R can not be reduced as it is required that the farthest switch has to send its statistics to the controller. Therefore, GlobeSnap takes a minimum of t_R time and a maximum of $t_F + t_R$ time to collect consistent statistics in a single round. Thus, giving the flexibility in adjusting the time required to complete a round.

3.5.4 Robustness

A method is robust if it provides consistent statistics in a given run without restarting the whole statistics collection process in case of a link failure, switch failure or packet loss event. GlobeSnap method robustly collects consistent statistics. It initiates the statistics collection separately in each connected subgraph of a network as shown in Algorithm 1. Failure in initiating, i.e., loss of special control packet or FlowMod packet can be rectified by retransmission, which is inbuilt in TCP.

Another type of failure that may happen is link or switch failure when snapshot is in progress within a connected subgraph. Snapshot recording is not hindered if there are multiple paths to reach the adjacent switch. In case there are no redundant paths and a link fails, snapshot recording resumes when the failed link restores and adjacent switch sends colored packets. In case of a switch failure, the switch sends the statistics once it is restored and receives a colored packet. Another type of failure is packet loss. Packet loss does not hinder snapshot recording since every packet is colored as per the switch state. As soon as new packets arrive at the switch, they will be colored red/green and transmitted which will make the next switch to record the snapshot. Whereas, OpenSnap has to restart the statistics collection again when a link/switch fails or a packet drop event happens, as this can lead to no marker packet being forwarded from one switch to another.

3.5.5 Estimating Network Parameters from Snapshot

A snapshot recorded with Globesnap offers some insights into computing network parameters. Considering that f_k is a forward flow with red packets, r^l is a reverse flow, and link is symmetric, the following observations can be made: Considering that the first red packet takes d amount of time to reach from switch S_i to switch S_j, and during the same amount of time, all reverse flows in R would have transmitted $\sum_{l=1}^{|R|} sent(r_j^l) - recv(r_i^l)$ number of packets i.e., the sum of the differences between sent and received statistics of all reverse flows. This fact states that flow rate of any reverse flow r^l can be derived by $\frac{sent(r_j^l) - recv(r_i^l)}{d}$. If at least one forward flow has its $sent() - recv() > 0$, then it means that there were enough packets to utilize link capacity on the link connecting switch S_i to switch S_j. In addition, if the link is symmetric, then it can be stated that $\sum_{l=1}^{|R|} sent(f_j^l) - recv(f_i^l)$, i.e., the number of packets transmitted during time d from switch S_j to switch S_i is same as the number of

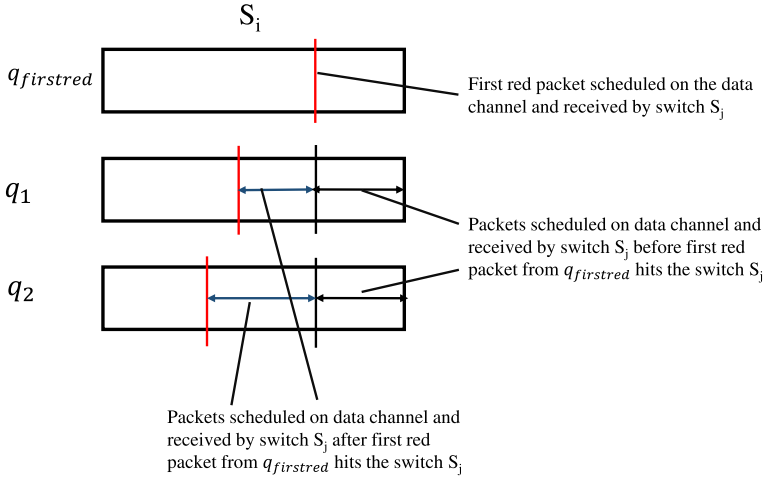


Fig. 5 Network parameter estimation

packets transmitted from switch S_i to switch S_j. Using the relation stated above, the aggregate input traffic rate can be estimated on the forward link. Also, controller has the timestamps at which the statistics are received from the switches. From these timestamps, controller can estimate d.

Now let us look at switch S_i having multiple queues (as shown in Fig. 5), each queue carrying multiple flows. When a red packet arrives at switch S_i, it sends the statistics of all the flows to the controller and packets coming thereafter are marked red. The first red packet, depending on which flow it belongs to, is placed in one of the queues. Similarly, packets of different flows arriving hereafter will be colored red and placed in different queues. Depending on the packet scheduling at switch S_i, the first red packet in any of the queues may reach switch S_j. When the first red packet reaches switch S_j, it sends the statistics of all the flows to the controller. If the first red packet which reached switch S_j is from $q_{firstred}$ at switch S_i, then all the flows in $q_{firstred}$ will have their *sent*() and *recv*() statistics equal i.e., the number of packets recorded at switch S_i and yet to reach switch S_j is zero. For queues other than $q_{firstred}$ at switch S_i, flows will have $sent() > recv()$. In a queue q_i, where i ≠ firstred, the ratio of $sent() - recv()$ of flows gives the ratio of their traffic rates. This is attributed to the distribution of packets of different flows in the region of a queue between the first red packet of that queue and the point when the first red packet from $q_{firstred}$ has reached switch S_j.

From the above observations, various parameters can be estimated using the statistics collected by Globesnap. Traffic rates of flows and links are important statistics in networks. Input traffic rate of forward link and reverse link between switch S_i and switch S_j are estimated using d and $\sum_{l=1}^{|R|} sent(r_j^l) - recv(r_i^l)$ as explained above. RFC 3272 [28] defines a bottleneck network element as whose input traffic rate tends to be greater than its output rate. The input traffic rate on link connecting switch S_j to switch S_i can be calculated as $\frac{\sum_{l=1}^{|R|} sent(f_j^l) - recv(f_i^l)}{d}$. This input rate is compared with the transmission rate of the link and with a suitable threshold the link can be identified as bottleneck link. Flow rates of reverse flows is estimated using the individual *sent*() and *recv*() statistics of each flow. For forward flows, queue bandwidth is split in the ratio of $sent() - recv()$ of the flows in a given queue resulting in individual flow rates. For a given flow, across the links, these flow rates can be compared and the minimum flow rate is taken as the end-to-end flow rate of the flow. This also lets us identify the bottleneck link for a given flow.

Computing packet loss on a link connecting switch S_j to switch S_i is achieved by taking the sum of $sent() - recv()$ of all reverse flows and differencing it from the sum of total queue capacity and channel capacity. Computing packet loss on forward link connecting switch S_i to switch S_j is achieved by taking the sum of $sent() - recv()$ of all reverse flows and forward flows and differencing it with the sum of total queue capacity and channel capacity.

3.6 Overhead

In this section, we discuss the overhead in the network in terms of the number of control messages required to collect the statistics of the underlying network using CeMon [4], OpenNetMon [5], OpenSnap [6] and GlobeSnap.

Consider a network of N switches, which has a diameter d and an in-band SDN controller configuration. Let the average distance (in terms of the number of links) from a switch to the SDN controller be $\frac{d}{2}$. Therefore, round trip distance from the SDN controller to a switch is d. In OpenNetMon [5], the controller sends the flow statistics request only to the edge switches of every flow. Therefore, the number of control packets in the network will be $f \times d$, where f is the total number of flows in the network. In CeMon [4], instead of collecting statistics only from the edge switches, it collects the statistics from some of the switches for each flow. We assume that the number of switches from which the controller collects the statistics for a flow are $\frac{N}{2}$. The total number of switches that will be polled for all flow statistics are $\frac{N \times f \times d}{2}$. Therefore, the total number of control packets in the network will be $\frac{(N \times f \times d)}{2}$. In OpenSnap [6], two marker packets per link are required to collect the statistics from all the switches. As OpenSnap is designed for a spanning tree protocol (STP) networks, the number of links in the network would be $N - 1$. Therefore, the total number of marker packets in the network will be $\frac{(2 \times (N-1)) \times d}{2}$. There will be N statistics replies to the controller. Therefore, the total number of control packets in the network will be $\frac{((3 \times N)-2) \times d}{2}$. This overhead increases in case of a network with Non-FIFO channels. If each switch has q queues configured on each interface then the overhead to collect consistent statistics for all the flows is $\frac{((3 \times N)-2) \times q \times d}{2}$

In GlobeSnap, there is no marker-like control packet on the data channel. To start the statistics collection process, the controller sends special control packets to a few switches. In the worst case, it can be N switches. Since usually, a flow will go through at least two switches, it can be averaged to $\frac{N}{2}$ switches. There will be one statistics reply from all switches. The total number of control packets required to initiate and collect the statistics from all the switches will be $\frac{((N/2)+N) \times d}{2}$. However, after the collection of statistics, the controller has to reset the state of the switches. The overhead to reset a switch is $\frac{d}{2}$. Therefore, the overhead to reset all the switches is $\frac{N \times d}{2}$. To update the current round color the controller has to send a control messages to each switch. This introduces an overhead of $\frac{N \times d}{2}$ messages. Thus, the total number of control packet required in GlobeSnap is $\frac{3.5 \times N \times d}{2}$.

The control message overhead in GlobeSnap is independent of the number of flows in the network as compared to CeMon and OpenNetMon in which the overhead increases with the increase in the number of flows. Usually $f >> N$. The overhead in OpenSnap with Non-FIFO channels is q times more compared to GlobeSnap, where q is the number of queues configured on an output interface of a switch.

4 Implementation Details

We use Mininet [16] to emulate a network of Open vSwitch [27] switches and ryu [29] controller to communicate with Open vSwitch switches using OpenFlow 1.3 [25]. In Mininet, the default configuration of controller is out-of-band, which is not practical as it requires a dedicated link from every switch to the controller. We implemented an in-band configuration of controller for our experiments.

4.1 Overview of OpenFlow Features

In this section, we give an overview of OpenFlow features and its extensions that we use to implement GlobeSnap.

4.1.1 Registers

Open vSwitch has multiple 32-bit registers that retain their state until a packet is being processed in the switch processing pipeline. These registers act like variables, they provide space to Open vSwitch for temporary storage while packet is being processed. We use the Nicira extension [30] for OpenFlow 1.3 that supports setting and matching of these registers.

4.1.2 Multiple Forwarding Tables Pipeline

OpenFlow switches can have multiple forwarding tables and each forwarding table can contain multiple flow entries to forward the network traffic. When a packet comes to an OpenFlow switch, the packet can match against a flow entry in any of the forwarding tables. Once we find a match, the packet can be forwarded to another forwarding table using "Goto Instruction", where the same process will be repeated. A given flow entry can only forward the packets to another forwarding table with a greater table number than its own table number. That is, the packet processing pipeline always goes in the forward direction not in backward. The processing pipeline stops when the packet cannot be forwarded to any further forwarding table. At the end of the processing pipeline the associated actions are performed. If a packet does not match any flow entry in the forwarding table, the table miss actions are performed [24].

4.2 Implementing GlobeSnap in OpenFlow

In this section, we explain how we use registers, multiple forwarding table and multiple pipelines to maintain the states in OpenFlow switches and thus implement GlobeSnap.

Figure 6 shows the processing of the incoming packet in GlobeSnap in a given run. We use four tables in each switch namely, PREPROCESSING table, STATE table, LOGIC table, and FORWARDING table to perform conditional forwarding w.r.t to a given color. The PREPROCESSING table identifies the incoming packet's

color as current round color or other color and forwards it to the STATE table and LOGIC table. The STATE table is responsible for maintaining the state of the switch as RED, GREEN, or WHITE that it does with the help of a 32-bit register provided by Open vSwitch [27] as part of Nicira Extensions register. Note that, the register alone can not store the state of the switch as the registers are always initialized to '0' and their values can not persist from one packet processing to another. This is the reason for using STATE table. The STATE table has the flow entries to set the value of the register which can then be accessed in the subsequent flow tables. This is how state of a switch is maintained. LOGIC table implements the main logic of the algorithm by accessing the register value.

1. If Register_value = 0, then it forwards the packet to the FORWARDING table.
2. If Register_value = 1, then it does the following,

    (a) Send the statistics to the controller.
    (b) Color the packet with current round color.
    (c) Add a new flow entry in STATE table with higher priority, which sets the register value to 2.
    (d) Forward the packet to the FORWARDING table.

3. If Register_value = 2, then it colors every incoming packet with the current round color and forwards the packet to the FORWARDING table.

The register values 1 and 2 are used to distinguish between the first red/green packet and the packets received after the first red/green packet. FORWARDING table processes the packet based on the flow entries added by the controller.

Once the switch has sent the statistics to the controller, the STATE table of a given pipeline updates the register value to 2 to denote the RED state of a switch. Resetting of switch state requires deletion of the flow entries, from the STATE table, which correspond to the RED state of the switch. After each run of statistics collection, the controller sends a command to the switches to delete the flow entries from their STATE tables. On successful removal of the flow entries from STATE tables of all the switches, the network comes back again in WHITE state for the next run. As already discussed in Sect. 3.4.4, due to stray red packets, it would be difficult to simultaneously delete the flow entries from STATE tables of all the switches. So, we need a separate processing pipeline for green color.

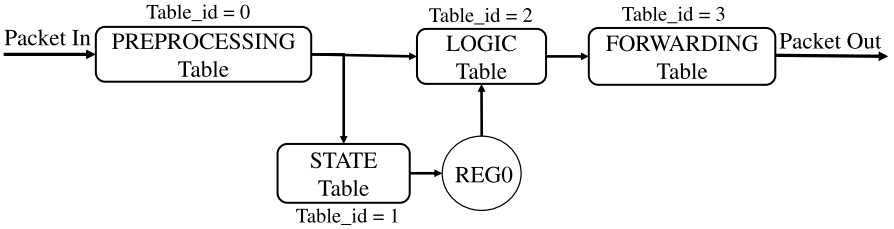


Fig. 6 Processing of incoming packets using multiple flow tables in an OpenFlow switch

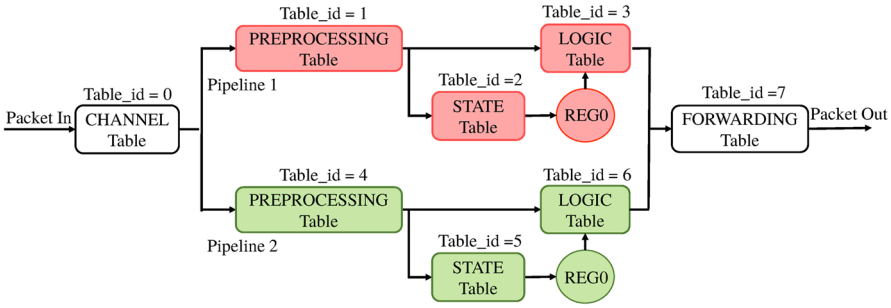


Fig. 7 Processing of packets in GlobeSnap using two pipelines

If we consider the processing of packets as shown in Fig. 6 as a single pipeline which processes the packets for a given round of statistics collection (let us say for red color), then to process the packets for next round of statistics collection (i.e., green color) we use another pipeline, as shown in Fig. 7. Pipeline1 and pipeline2 correspond to red and green color respectively. Both the pipelines consist of separate PREPROCESSING, STATE, and LOGIC tables with different table numbers, for example, table_id 1, 2, and 3 correspond to PREPROCESSING table, STATE table, and LOGIC table respectively of Pipeline1 and table_id 4, 5, and 6 correspond to PREPROCESSING table, STATE table, and LOGIC table respectively of pipeline2. The decision to which pipeline the packet needs to be forwarded is taken by the CHANNEL table (i.e., table_id = 0) given in Fig. 7. Once the pipeline is decided for packet forwarding, all the packets are processed through the selected pipeline for the given round of statistics collection and the other pipeline is being reset for the next round of statistics collection. In every alternate round, the controller uses the same pipeline for statistics collection.

5 Experimental Evaluation

In this section, we evaluate the performance of GlobeSnap w.r.t to collection of consistent statistics and compare the results with Simple Polling, CeMon [4], OpenNetMon [5], and OpenSnap [6] with FIFO and Non-FIFO channels. All experiments are performed on the same network topology as given in Fig. 8 and configurations as given in Table 5. The controller is running on host h_1. Flow f_1 traces the same path as statistics request messages, flow f_2 traces the opposite path to the statistics request messages and flow f_3 is not following any strict direction w.r.t statistics messages.

5.1 Consistency Evaluation

The experiments are performed with a constant bit rate (CBR) traffic over both TCP and UDP and variable bit rate (VBR) traffic over UDP. For consistency evaluation, we got similar results for all three kinds of traffic. However, in this paper, only the results of CBR traffic running over UDP are presented, we did not observe

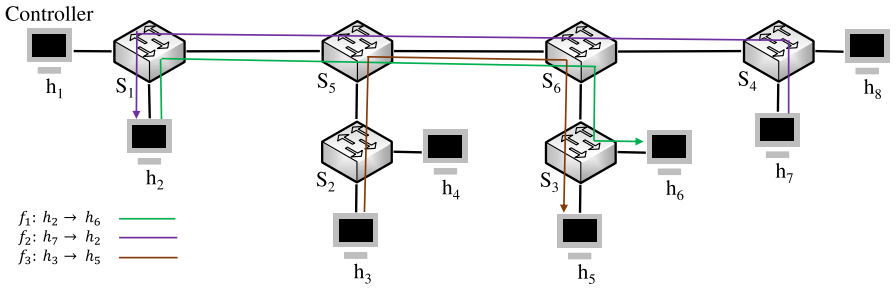


Fig. 8 Topology used for consistency evaluation

Table 5 Network configurations for consistency evaluation experiments

Topology	Given in Fig. 8
Number of queues per port	3
Number of flows	3 (f_1 : $h_2 \rightarrow h_6$, f_2 : $h_7 \rightarrow h_2$, and f_3 : $h_3 \rightarrow h_5$)
Traffic generator	D-ITG [31] (4 Mbps per flow)
Controller configuration	In-band

considerable variations in the results when experiments are performed with VBR traffic for all the methods. For the experiments, we have configured three queues q_0, q_1, and q_2 on each port of the switches. Flows f_1 and f_3 are forwarded through queue q_1, and flow f_2 is forwarded through queue q_2.

For a particular flow f_n, we calculate $\lambda = sent(f_i^n) - recv(f_j^n)$ as a measure to compare the consistency achieved by different methods, where i is the switch connected to the source host and j is the switch connected to the destination host of flow f_n. For OpenNetMon [5] the controller polls the source and destination switches of all the flows. Simple polling and OpenNetMon [5] provide inconsistent statistics for flows f_1 and f_3 (as shown in Fig. 9a, c, d, f respectively). Whereas, they provide consistent statistics for flow f_2 (as shown in Fig. 9b, e respectively) because the controller is running on host h_1 which is connected to switch S_1 and the destination host h_2 of flow f_2 is also connected to switch S_1. Thus, when the controller initiates the statistics collection process by sending the statistics request messages to the switches, for flow f_2 the destination switch S_1 sends the statistics before the source switch S_4. This is because for flow f_2 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_4 of flow f_2 its flow match counter would have increased. Thus, it gives consistent statistics as sent statistics of flow f_2 is greater than its received statistics.

CeMon [4] proposes an algorithm to calculate the polling frequency for each flow on a given switch. We run this algorithm for all three flows f_1,f_2, and f_3 and the controller polls the source and destination switches at the calculated frequency. As

shown in Fig. 10a–c, CeMon gives inconsistent statistics for all three flows, f_1, f_2, and f_3. Whereas, GlobeSnap provides consistent statistics for all three flows.

In OpenSnap with FIFO channels, the statistics are consistent for all three flows as shown in Fig. 11a–c. Whereas, the statistics for flows f_1 and f_3 are inconsistent in case of OpenSnap with Non-FIFO channels as shown in Fig. 10d and f. It provides consistent statistics for flow f_2, as shown in Fig. 10e because the controller is running on host h_1 which is connected to switch S_1 and the destination host h_2 of the flow f_2 is also connected to the switch S_1. Thus, when the controller initiates the statistics collection process, switch S_1 sends the received statistics of flow f_2 to the controller and forwards the marker packet to all the adjacent switches. So, by the time marker packet reaches source switch S_4 of flow f_2 its flow match counter would have increased. Thus, it gives consistent statistics as sent statistics of flow f_2 is greater than its received statistics. Whereas, GlobeSnap provides consistent statistics for all three flows in both, network with FIFO channels and network with Non-FIFO channels.

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,

$$\%\ consistency = \frac{Number\ of\ rounds\ providing\ consistent\ statistics}{Total\ number\ of\ rounds\ of\ statistics\ collection} \times 100. \quad (3)$$

Figure 11d, shows the percentage of consistency achieved by each solution. OpenSnap [6] with Non-FIFO channels provides least consistency whereas, simple polling, CeMon [4], and OpenNetMon [5] provides 59.89%, 52.25%, and 43.19% consistent statistics respectively. Both OpenSnap with FIFO channels [6] and GlobeSnap provides 100% of consistent statistics. As already explained in Sects. 1 and 3.3, OpenSnap is not an efficient solution for OpenFlow based networks with Non-FIFO channels.

5.2 Synchronicity

As discussed in Sect. 3.5, synchronicity is measured as the difference between highest timestamp and lowest timestamp in a snapshot. Globesnap method ensures that the synchronicity of a snapshot of a given flow does not exceed half of its RTT. This is supported by experimental results as shown in Fig. 12. Maximum RTT of flow f_1 is 1.35 and half of it is 0.67. The synchronicity of snapshot recorded for flow f_1 is always below 0.67.

5.3 Use Cases of GlobeSnap

As already discussed in Sect. 3.4.2, the controller can use the collected consistent statistics to identify the bottleneck link and to estimate the packet loss in a given queue and link. In this section, we present the results for the use cases of GlobeSnap. The existing works OpenNetMon [5], CeMon [4], Simple polling does not provide

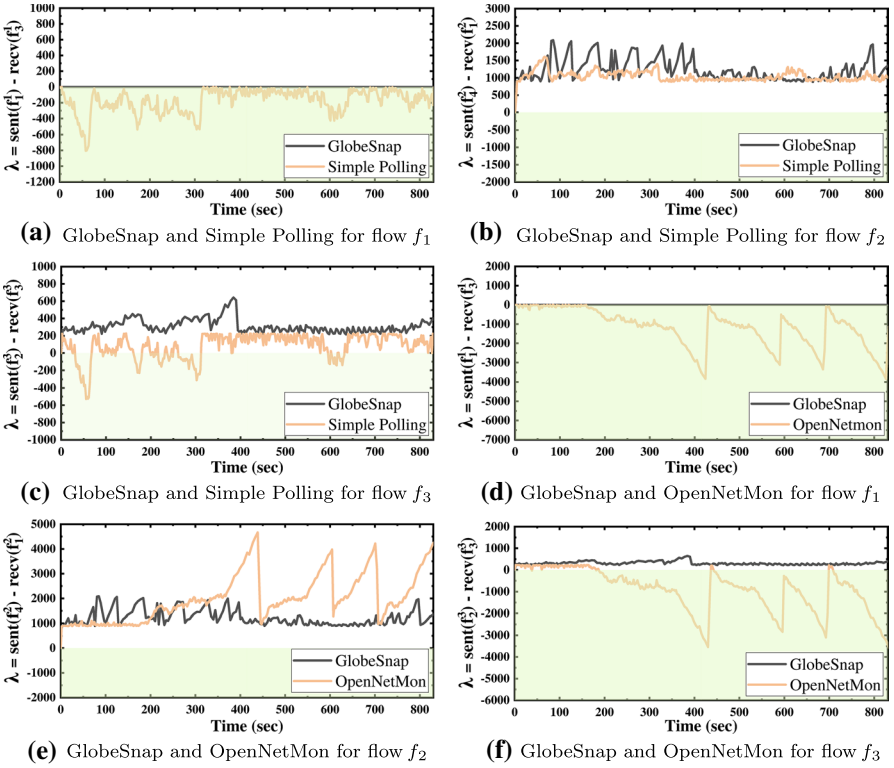
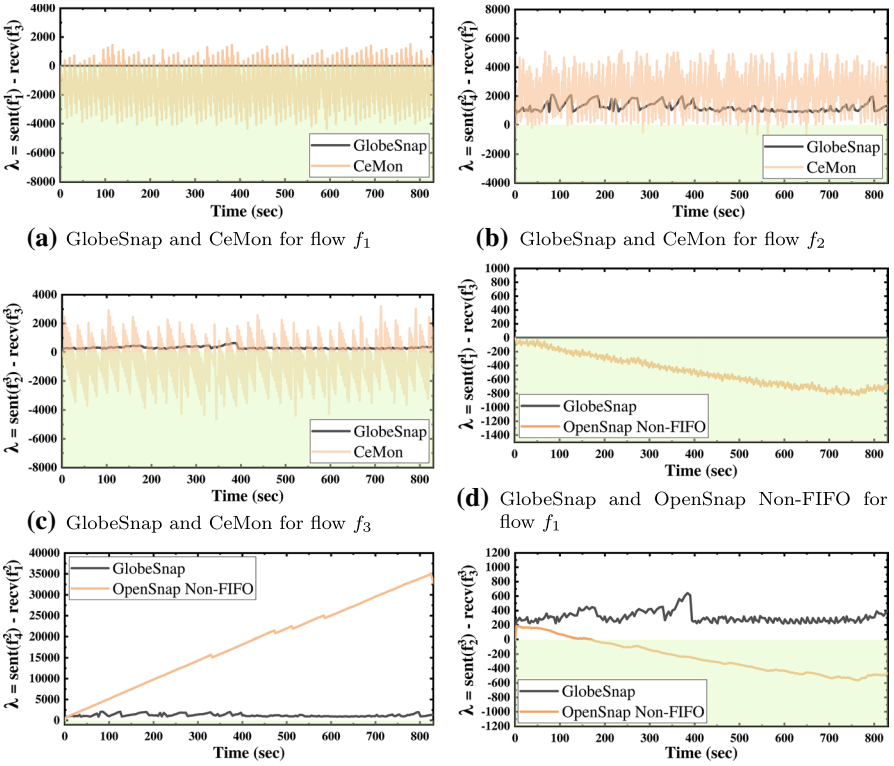


Fig. 9 Comparison of GlobeSnap with simple polling, and OpenNetMon [5] with Non-FIFO channels. The shaded area represents the inconsistent statistics

consistent statistics. Thus, they can not be used to identify the bottleneck link and packet losses. As shown in the previous section OpenSnap provides consistent statistics for FIFO networks only. Thus, it can not identify the bottleneck links in Non-FIFO networks. For the experiments, we have used the topology as shown in Fig. 13 and network configurations as given in Table 6.

Initially, there are only two flows in the network f_1 and f_2. Flow f_1 is forwarded through queue q_2, whereas flow f_2 is forwarded through queue q_1. After 120 s, two more flows (f_3 and f_4) are admitted in the network, where the flow f_3 is forwarded through queue q_2 and flow f_4 is forwarded through queue q_1. After next 120 s, one more flow (i.e., f_5) is admitted to the network, which is forwarded through queue q_2. After 60 s flow f_6 is admitted in the network, which is forwarded through queue q_1. There are six flows in the network, and all six flows go through link L_2 : $S_3 \rightarrow S_2$. Whereas, only three flows go through link L_1 : $S_2 \rightarrow S_1$ and L_3 : $S_4 \rightarrow S_3$.



(a) GlobeSnap and CeMon for flow f_1

(b) GlobeSnap and CeMon for flow f_2

(c) GlobeSnap and CeMon for flow f_3

(d) GlobeSnap and OpenSnap Non-FIFO for flow f_1

(e) GlobeSnap and OpenSnap Non-FIFO for flow f_2

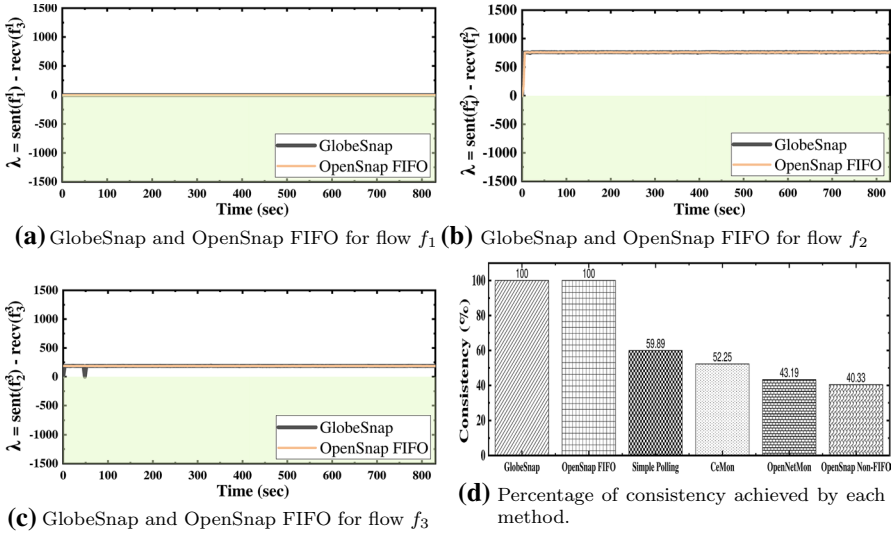
(f) GlobeSnap and OpenSnap Non-FIFO for flow f_3

Fig. 10 Comparison of GlobeSnap with CeMon [4], and OpenSnap [6] with Non-FIFO channels for consistency. The shaded area represents the inconsistent statistics

5.3.1 Identifying Bottleneck Links

The statistics collected using GlobeSnap can be used to identify the bottleneck links correctly. We consider a link to be a bottleneck link when it reaches 70% of its capacity. We estimate the arrival rate of a link by taking the difference of the number of bytes sent from source switch and the number of bytes received at the destination switch of a link for all the flows going through the link and divide it by the link delay.

As shown in Fig. 14a and c, links L_1 : $S_2 \rightarrow S_1$, and L_3 : $S_4 \rightarrow S_3$ are not bottleneck links. As all six flows are going through link L_2 : $S_3 \rightarrow S_2$, the arrival rate increases every time when a new flow joins the link. Between 266th and 269th second the traffic arrival rate increases which results in 70% link utilization as shown in Fig. 14b. Thus, L_2 link is identified as a bottleneck link. Between 269th and 272th second the link state of L_2 goes above the link capacity.



(a) GlobeSnap and OpenSnap FIFO for flow f_1 **(b)** GlobeSnap and OpenSnap FIFO for flow f_2

(c) GlobeSnap and OpenSnap FIFO for flow f_3 **(d)** Percentage of consistency achieved by each method.

Fig. 11 **a**–**c** Show the comparison of GlobeSnap with OpenSnap [6] with FIFO channels for consistency. The shaded area represents the inconsistent statistics. **d** Shows the percentage of consistency achieved by each method

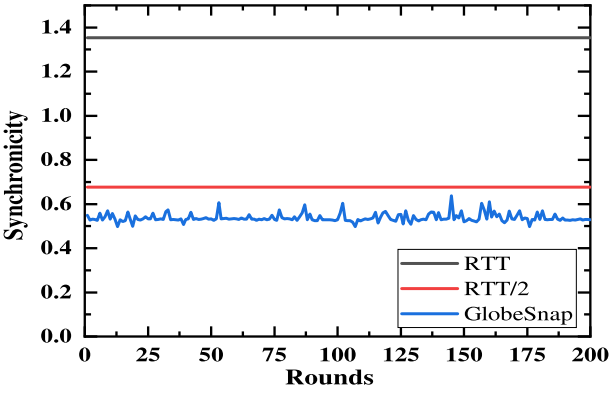


Fig. 12 Synchronicity of GlobeSnap

5.3.2 Computing Packet Loss

For packet loss evaluation, we have considered the same topology as shown in Fig. 13 and network configurations as given in Table 6. We compare the packet loss results of GlobeSnap with the actual number of packet loss provided by Open vSwitch (OVS) queue statistics (i.e., NetEm [32] statistics).

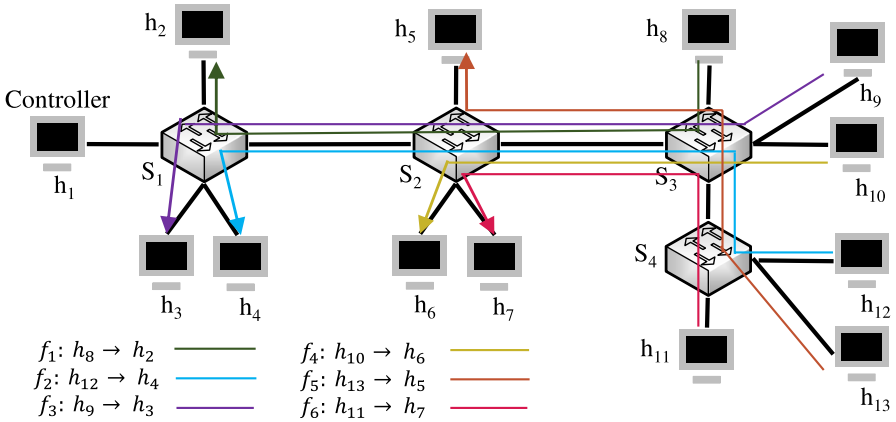
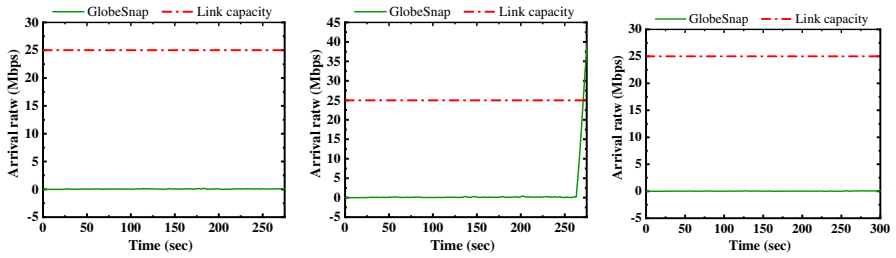


Fig. 13 Topology for bottleneck link and packet loss evaluations

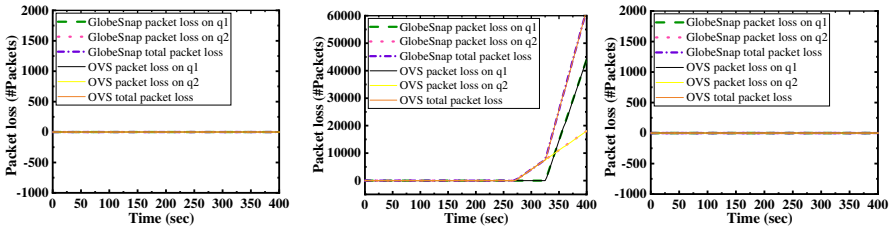
Table 6 Network configurations for experiments to identify the bottleneck link and number of packets lost on a link

Topology	Given in Fig. 13
No. of queues per port	3, namely q_0, q_1, and q_2
Bandwidth of link	25 Mbps
Queue bandwidth	10 Mbps for queue q_1, 12 Mbps for queue q_2 and remaining 3 Mbps for q_0.
Host to switch delay	5 µs
Switch to switch delay	176 µs
Number of flows	6 (f_1 : $h_8 \rightarrow h_2$, f_2 : $h_{12} \rightarrow h_4$, f_3 : $h_9 \rightarrow h_3$, f_4 : $h_{10} \rightarrow h_6$, f_5 : $h_{13} \rightarrow h_5$, and f_6 : $h_{11} \rightarrow h_7$)
Traffic generator	D-ITG [31] (4 Mbps per flow)
Controller	In-band



(a) Link L_1 : $S_2 \rightarrow S_1$ **(b)** Link L_2 : $S_3 \rightarrow S_2$ **(c)** Link L_3 : $S_4 \rightarrow S_3$

Fig. 14 Bottleneck link identification in the network using GlobeSnap



(a) Packet loss on the link L_1 : $S_2 \rightarrow S_1$. **(b)** Packet loss on the link L_2 : $S_3 \rightarrow S_2$. **(c)** Packet loss on the link L_3: $S_4 \rightarrow S_3$.

Fig. 15 Packet loss measurements on each link of the network by GlobeSnap and actual packet loss provided by queues statistics

Figure 15a and c show that the results of GlobeSnap for packet loss through individual queues and for total packet loss on the links $L_1 : S_2 \rightarrow S_1$, and $L_3 : S_4 \rightarrow S_3$ are overlapping with the results given by OVS and the packet loss on both the links is 0. All six flows are going through link $L_2 : S_3 \rightarrow S_2$, Fig. 15b shows that packets start dropping at 272th second on queue q_2 and 327th second on queue q_1 of link $L_2 : S_3 \rightarrow S_2$ when fifth and sixth flows joined the link respectively. The total packet loss on the link is sum of packet loss on both the queues. The results of GlobeSnap for packet loss through individual queues and total packet loss on the link are overlapping with the results given by OVS.

6 Conclusion

In this paper, we proposed an efficient and robust method to collect globally consistent statistics in OpenFlow based SDN networks. GlobeSnap uses a coloring mechanism to collect consistent statistics. GlobeSnap outperforms the state-of-the-art approach OpenSnap [6] and other polling-based methods in consistency evaluation. Also, in CeMon [4] and OpenNetMon [5], the overhead of the number of control packets increases if the number of flows in the network increases. Whereas, the control packet overhead in GlobeSnap is independent of the number of flows in the network. We also demonstrated that consistent statistics can be used to identify the bottleneck links accurately and to estimate the number of packet loss in the links.

A Appendix for Correctness

In this section, we show the correctness of the proposed solution to collect consistent statistics in OpenFlow networks with Non-FIFO channels.

Consider the network segment given in Fig. 16. There are two flows f_1 and f_2, both are going from switch S_1 to switch S_2. The controller is connected to switch S_1.

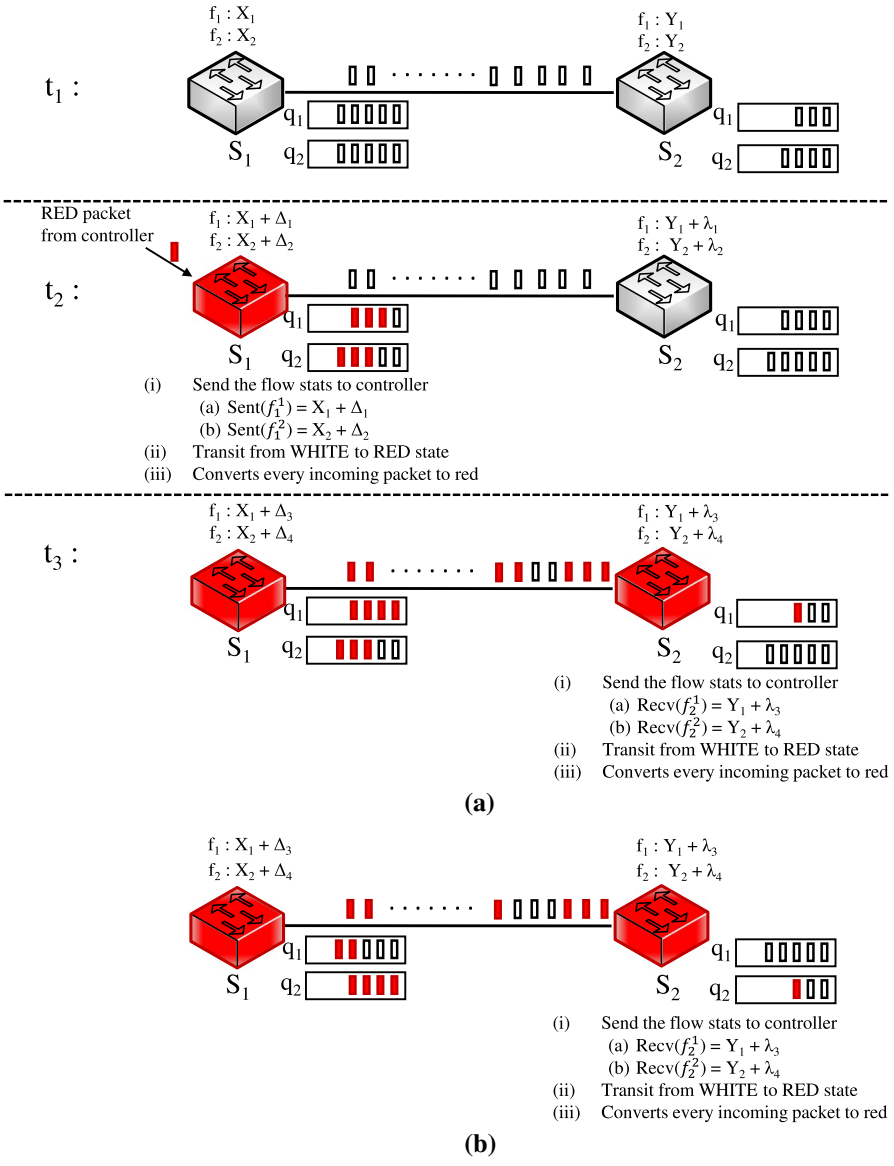


Fig. 16 Illustrating the correctness of GlobeSnap for OpenFlow based networks with Non-FIFO channels

Both switches have two queues, q_1 and q_2, configured on each port. For simplicity, let's assume flow f_1 is forwarded through queue q_1 and flow f_2 is forwarded through queue q_2. As shown in Fig. 16, at time t_1 the packet count corresponding to the flows f_1 and f_2 is X_1 and X_2 respectively at switch S_1. The count for the flows f_1 and f_2 is Y_1, Y_2 respectively at switch S_2 and the whole network is in WHITE state. Considering X_1 to be equal to Y_1 and X_2 to be equal to Y_2. Now at time t_2, the controller initiates

the statistics collection process by sending a red packet to switch S_1. On receipt of a red packet, switch S_1 sends the statistics to the controller as X_1+Δ_1[6] and X_2+Δ_2 for flows f_1 and f_2 respectively. Which are recorded as sent statistics for both the flows w.r.t to switch S_1, that is,

$$sent(f_1^1) = X_1 + \Delta_1, \tag{4}$$

$$sent(f_1^2) = X_2 + \Delta_2. \tag{5}$$

After sending the statistics to the controller the switch S_1 changes its state from WHITE to RED. Any further transmission of packets from switch S_1 are colored red. At time t_3, when the first red packet from switch S_1 hits the switch S_2, it triggers the statistics collection at switch S_2. This ensures that any packet which is transmitted after the statistics collection from switch S_1 will not be recorded in the received statistics at switch S_2. At time t_3, there are two possibilities,

Case1: First red packet scheduled on data channel is from the queue q_1 of switch S_1. At time t_3, let queue q_1 be scheduled first for packet transmission on the data channel as shown in Fig. 16a. The packets from queue q_1 are transmitted in FIFO order. When the first red packet which is transmitted from switch S_1 through queue q_1 hits the switch S_2, switch S_2 sends the statistics to controller as Y_1+λ_3[7] and Y_2+λ_4 for flows f_1 and f_2 respectively. Which are recorded as received statistics for flows f_1 and f_2 at switch S_2, that is,

$$recv(f_2^1) = Y_1 + \lambda_3, \tag{6}$$

$$recv(f_2^2) = Y_2 + \lambda_4. \tag{7}$$

Since the first red packet which triggers the statistics collection at switch S_2 was sent through queue q_1 of switch S_1. Thus it belongs to flow f_1. Any packet which is received at switch S_2 before the red packet was a white packet. If there is no packet loss on the link which connects switch S_1 and switch S_2 then,

$$\Delta_1 = \lambda_3. \tag{8}$$

Using Eqs. 4, 6 and 8

$$sent(f_1^1) = recv(f_2^1). \tag{9}$$

If there is a packet loss for flow f_1 on the link which connects switch S_1 and switch S_2 then,

[6] Δ_i, where i=1,2,3,…, is the number of packets sent from source switch to destination after time t_1

[7] λ_i, where i=1,2,3,…, is the number of packets received at the destination switch after time t_1

$$\Delta_1 > \lambda_3. \tag{10}$$

Using Eqs. 4, 6 and 10,

$$sent(f_1^1) > recv(f_2^1). \tag{11}$$

Using Eqs. 9 and 11,

$$sent(f_1^1) \geq recv(f_2^1). \tag{12}$$

Thus, it satisfies the consistency condition given in Eq. 2.

For flow f_2, the sent statistics can be greater than or equal to the received statistics. The sent statistics will be equal to received statistics, if there is no white packet in queue q_2 at switch S_1 when the first red packet from queue q_1 at switch S_1 is scheduled on data channel and there is no packet loss on the link which connects switch S_1 and switch S_2. That is,

$$\Delta_2 = \lambda_4. \tag{13}$$

Using Eqs. 5, 7 and 13,

$$sent(f_1^2) = recv(f_2^2). \tag{14}$$

The sent statistics will be greater than received statistics for flow f_2, if there is at least one white packet in queue q_2 at switch S_1 when the first red packet from queue q_1 of switch S_1 is scheduled on data channel or there is a packet loss on the link which connects switch S_1 and switch S_2. This gives,

$$\Delta_2 > \lambda_4. \tag{15}$$

Using Eqs. 5, 7 and 15

$$sent(f_1^2) > recv(f_2^2). \tag{16}$$

Using Eqs. 14 and 16

$$sent(f_1^2) \geq recv(f_2^2). \tag{17}$$

Thus, satisfies the consistency conditions given in Eq. 2.

Case2: First red packet scheduled on data channel is from the queue q_2 of switch S_1. At time t_3, let queue q_2 be scheduled first for the packet transmission on the data channel as shown in Fig. 16b. The packets from queue q_2 are transmitted in FIFO order. When the first red packet which is transmitted from switch S_1 through queue q_2 hits the switch S_2, it triggers the statistics collection process. Switch S_2 sends the statistics to the controller as $Y_1 + \lambda_3$ and $Y_2 + \lambda_4$ for flows f_1 and f_2 respectively. Which are recorded as received statistics for the flows w.r.t to switch S_2 as given in Eqs. 6 and 7. The first red packet which hits the switch S_2 is sent through queue q_2 from switch S_1. Any packet received by switch S_2 before the reception of the red packet is counted in the sent statistics at switch S_1. If there is no packet loss on the

link which connects switch S_1 and switch S_2, then it results in Eqs. 13 and 14. That is, sent and received statistics will be equal for flow f_2. If there is a packet loss on the link which connects switch S_1 and switch S_2 then it results in Eqs. 15 and 16. That is, the sent statistics will be greater than received statistics for flow f_2. Using Eqs. 13, 14, 15 and 16,

$$sent(f_1^2) \geq recv(f_2^2) \tag{18}$$

Thus, it satisfies the consistency condition given in Eq. 2. For flow f_1, the sent statistics can be greater than or equal to the received statistics. If there is no white packet in queue q_1 at switch S_1, when the first red packet from queue q_2 at switch S_1 is scheduled on data channel and there is no packet loss on the link which connects switch S_1 and switch S_2 then it results in Eqs. 8 and 9. That is, the sent statistics and received statistics for flow f_1 are equal. The sent statistics will be greater than received statistics if there is at least one white packet in queue q_1 at switch S_1 when the first red packet from queue q_2 at switch S_1 is scheduled on data channel or there is a packet loss on the link which connects switch S_1 and switch S_2. This results in Eqs. 10 and 11. Using Eqs. 8, 9, 10 and 11,

$$sent(f_1^1) \geq recv(f_2^1) \tag{19}$$

Thus, it satisfies the consistency condition given in Eq. 2.

Correctness of Consistent Statistics for End-to-End Path

We proved that GlobeSnap provides consistent statistics on a given link. It can be easily seen that GlobeSnap would also provide end-to-end consistent statistics using transitive relation between the switches for the flow. It can also be observed that even if switches are connected in a mesh topology then also GlobeSnap would provide consistent statistics. This is because every switch has a unique link from which it receives the packets to be forwarded towards the destination for a particular flow. In GlobeSnap, it is important to note that explicit marker packets are not required to collect the statistics. Colored packets themselves act as markers and triggers the statistics collection process. GlobeSnap always provides consistent statistics for all flows because, on a given link all the packets that arrived at source switch after it has sent the statistics to the controller will be colored red before transmission on the data channel. The destination switch of a link sends statistics to the controller only when the first red packet arrives at it. All the packets received at the destination switch before the arrival of the first red packet are white and were recorded in the sent statistics at source switch. Thus, the sent statistics will always be greater than or equal to the received statistics for a given flow on a given link.

B Experiment with Large Number of Packets and Over a Longer Time Window

Table 7 below shows the experimental results with large number of packets for the network topology given in Fig. 1. The controller polls both the switches (S_1 and S_2) by sending flow statistics requests. The results show even with longer time duration the inconsistencies in the collected statistics does not smooth out.

Table 7 Statistics collected at controller without enforcing order of events in statistics collection

Time	Statistics from switch S_1	Statistics from switch S_2	Sent–received	Stats
0	1978	1991	− 13	Inconsistent
2.08432	4325	4325	0	Consistent
4.23353	6696	6788	− 92	Inconsistent
6.36766	9076	9144	− 68	Inconsistent
8.49809	11,434	11,560	− 126	Inconsistent
10.62451	13,798	13,995	− 197	Inconsistent
12.76864	16,169	16,381	− 212	Inconsistent
14.88422	18,555	18,741	− 186	Inconsistent
16.98311	20,923	21,132	− 209	Inconsistent
19.13689	23,321	23,557	− 236	Inconsistent
21.21459	25,707	25,843	− 136	Inconsistent
23.34231	28,068	28,257	− 189	Inconsistent
25.48619	30,441	30,705	− 264	Inconsistent
27.64657	32,809	33,088	− 279	Inconsistent
29.69471	35,171	35,359	− 188	Inconsistent
31.84977	37,536	37,888	− 352	Inconsistent
34.0017	39,913	40,312	− 399	Inconsistent
36.10995	42,284	42,674	− 390	Inconsistent
38.2682	44,653	45,131	− 478	Inconsistent
40.40304	47,028	47,486	− 458	Inconsistent
42.515	49,411	49,932	− 521	Inconsistent
44.67638	51,772	52,364	− 592	Inconsistent
46.79377	54,146	54,771	− 625	Inconsistent
48.88439	56,523	57,076	− 553	Inconsistent
51.05453	58,896	59,510	− 614	Inconsistent
53.12028	61,264	61,870	− 606	Inconsistent

Acknowledgements This research was supported in part by NSF under grants CNS-1618339, CNS-1617729, CNS-1814322, CNS-1831140, CNS-1836772, and CNS-1901103.

References

1. Zhang, Y., Cui, L., Wang, W., Zhang, Y.: A survey on software defined networking with multiple controllers. J. Netw. Comput. Appl. **103**, 101–118 (2018)
2. Megyesi, P., Botta, A., Aceto, G., Pescapé, A., Molnár, S.: Challenges and solution for measuring available bandwidth in software defined networks. Comput. Commun. **99**, 48–61 (2017)
3. Yaseen, N., Sonchack, J., Liu, V.: Synchronized network snapshots. In: Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, pp. 402–416. ACM (2018)
4. Su, Z., Wang, T., Xia, Y., Hamdi, M.: CeMon: a cost-effective flow monitoring system in software defined networks. Comput. Netw. **92**, 101–115 (2015)
5. Van Adrichem, N.L., Doerr, C., Kuipers, F.A.: OpenNetMon: network monitoring in OpenFlow software-defined networks. In: Network Operations and Management Symposium (NOMS), pp. 1–8. IEEE (2014)
6. Rathee, S., Sharma, R., Jain, P.K., Haribabu, K., Bhatia, A., Balasubramaniam, S.: OpenSnap: collection of globally consistent statistics in software defined networks. In: 11th International Conference on Communication Systems & Networks (COMSNETS), pp. 149–156. IEEE (2019)
7. Wundsam, A., Levin, D., Seetharaman, S., Feldmann, A.: Ofrewind: Enabling record and replay troubleshooting for networks. In: USENIX Annual Technical Conference, pp. 327–340. USENIX Association (2011)
8. Foerster, K.T., Schmid, S., Vissicchio, S.: Survey of consistent software-defined network updates. IEEE Commun. Surveys Tutor. **21**(2), 1435–1461 (2018)
9. Yaseen, N., Sonchack, J., Liu, V.: tpprof: a network traffic pattern profiler. In: 17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20), pp. 1015–1030 (2020)
10. Chowdhury, S.R., Bari, M.F., Ahmed, R., Boutaba, R.: PayLess: A Low Cost Network Monitoring Framework for Software Defined Networks. Network Operations and Management Symposium (NOMS), pp. 1–9. IEEE (2014)
11. Kim, C., Sivaraman, A., Katta, N., Bas, A., Dixit, A., Wobker, L.J.: In-band network telemetry via programmable dataplanes. In: ACM SIGCOMM, Vol. 15 (2015)
12. Yu, C., Lumezanu, C., Zhang, Y., Singh, V., Jiang, G., Madhyastha, H.V.: FlowSense: monitoring network utilization with zero measurement cost. In: International Conference on Passive and Active Network Measurement, pp. 31–41. Springer (2013)
13. Chandy, K.M., Lamport, L.: distributed snapshots: determining global states of distributed systems. ACM Trans. Comput. Syst. (TOCS) **3**(1), 63–75 (1985)
14. Kshemkalyani, A.D., Raynal, M., Singhal, M.: An introduction to snapshot algorithms in distributed computing. Distrib. Syst. Eng. **2**(4), 224 (1995)
15. Awan, I.I., Shah, N., Imran, M., Shoaib, M., Saeed, N.: An improved mechanism for flow rule installation in In-band SDN. J. Syst. Archit. **96**, 32–51 (2019)
16. Lantz, B., Heller, B., McKeown, N.: A network in a laptop: rapid prototyping for software-defined networks. In: Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks, p. 19. ACM (2010)
17. Tootoonchian, A., Ghobadi, M., Ganjali, Y.: OpenTM: traffic matrix estimator for OpenFlow networks. International Conference on Passive and Active Network Measurement, pp. 201–210. Springer (2010)
18. Li, Y., Miao, R., Kim, C., Yu, M.: Flowradar: a better netflow for data centers. In: 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI), pp. 311–324 (2016)
19. Claise, B.: Cisco Systems NetFlow Services Export Version, Vol. 9 (2004)
20. Li, Y., Miao, R., Kim, C., Yu, M.: LossRadar: fast detection of lost packets in data center networks. In: Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies, pp. 481–495. ACM (2016)
21. Suh, J., Kwon, T.T., Dixon, C., Felter, W., Carter, J.: OpenSample: a low-latency, sampling-based measurement platform for commodity SDN. In: 34th International Conference on Distributed Computing Systems (ICDCS), pp. 228–237. IEEE (2014)

22. Sherwin, J., Sreenan, C.J.: LogSnap: creating snapshots of OpenFlow data centre networks for offline querying. In: 2019 10th International Conference on Networks of the Future (NoF), pp. 66–73. IEEE (2019)
23. Configuring QoS. https://www.cisco.com/en/US/docs/switches/lan/catalyst3850/software/release/3.2_0_se/multibook/configuration_guide/b_consolidated_config_guide_3850_chapter_010000.html. Accessed 21 Feb 2020
24. OpenFlow Switch Specification. https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.3.0.pdf. Accessed 17 Oct 2020
25. McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., Turner, J.: OpenFlow: enabling innovation in campus networks. ACM SIGCOMM Comput. Commun. Rev. **38**(2), 69–74 (2008)
26. Lai, T.H., Yang, T.H.: On distributed snapshots. Inf. Process. Lett. **25**(3), 153–158 (1987)
27. Pfaff, B., Pettit, J., Koponen, T., Jackson, E., Zhou, A., Rajahalme, J., Gross, J., Wang, A., Stringer, J., Shelar, P., et al.: The design and implementation of open vswitch. In: 12th USENIX Symposium on Networked Systems Design and Implementation, pp. 117–130. NSDI (2015)
28. Awduche, D., Chiu, A., Elwalid, A., Widjaja, I., Xiao, X.: Overview and Principles of Internet Traffic Engineering. Tech. Rep., RFC 3272, May (2002)
29. ryu Documentation. https://buildmedia.readthedocs.org/media/pdf/ryu/latest/ryu.pdf. Accessed 17 Oct 2020
30. Open vSwitch Manual. http://www.openvswitch.org/support/dist-docs/ovs-vswitchd.conf.db.5.html. Accessed 17 Oct 2020
31. Botta, A., Dainotti, A., Pescapè, A.: A tool for the generation of realistic network workload for emerging networking scenarios. Comput. Netw. **56**(15), 3531–3547 (2012)
32. Hemminger, S., et al.: Network Emulation with NetEm, pp. 18–23. Linux conf au (2005)

Authors and Affiliations

Sandhya Rathee[1] · Nitin Varyani[2] · K. Haribabu[1] · Aakash Bajaj[1] · Ashutosh Bhatia[1] · Ram Jashnani[1] · Zhi-Li Zhang[2]

Nitin Varyani
varya001@umn.edu

K. Haribabu
khari@pilani.bits-pilani.ac.in

Aakash Bajaj
f2015586@pilani.bits-pilani.ac.in

Ashutosh Bhatia
ashutosh.bhatia@pilani.bits-pilani.ac.in

Ram Jashnani
f2015099@pilani.bits-pilani.ac.in

Zhi-Li Zhang
zhang089@umn.edu

[1] Birla Institute of Technology and Science Pilani - Pilani Campus, Pilani, Rajasthan, India

[2] University of Minnesota – Twin Cities, Minneapolis, MN, USA