Measurement 95 (2017) 317–327

Contents lists available at ScienceDirect

Measurement

journal homepage: www.elsevier.com/locate/measurement

A multi-objective software defined network traffic measurement

Hamid Tahaei [a,*], Rosli Salleh [a,*], Suleman Khan [a], Ryan Izard [b], Kim-Kwang Raymond Choo [c,d], Nor Badrul Anuar [a,*]

[a] Faculty of Computer Science and Information Technology, University of Malaya, Kuala Lumpur, Malaysia
[b] Department of Electrical and Computer Engineering, Clemson University, Clemson, USA
[c] Department of Information Systems and Cyber Security, University of Texas at San Antonio, USA
[d] School of Information Technology & Mathematical Sciences, University of South Australia, Australia

ARTICLE INFO

Article history:
Received 16 April 2016
Received in revised form 17 September 2016
Accepted 11 October 2016
Available online 13 October 2016

Keywords:
Software defined measurement
Software defined networking
Network monitoring

ABSTRACT

Software Defined Networking (SDN) with defining characteristics, such as "separation of data and control plane" and "centralizing network control with decision making", has significantly simplified network management. However, active monitoring techniques used to dynamically measure network traffic introduce additional overheads in the network, while a passive approach lacks accuracy in terms of traffic measurement. As a result, various efforts have been devoted to designing per-flow based network measurement system to address both accuracy and overhead challenges. Existing measurement techniques lack a multi-objective network measurement mechanism to overcome various overheads, like communication cost, controller computation, and accuracy in a real-time environment. Therefore, this paper presents a novel and practical solution to enable accurate real-time traffic matrix for the traffic measurement system in SDN. The solution is proposed to measure fine-grained monitoring task with less controller communication and computational cost with high accuracy. The solution is based on two measurement designs, namely: fixed and elastic schemas. Our experiments demonstrate that both fixed and elastic schemas achieve significant overhead reduction without compromising on accuracy.

1. Introduction

Network monitoring and measurement are crucial to Network Management Systems (NMS). For example, large data center networks (DCN) require accurate measurements of traffic flows to effective monitor the traffic volume in real-time. Similarly, a per-flow traffic measurement system can be used to monitor the micro-details of every flow in different network layers. Such system is also known as a fine-grained monitoring system. The fine-grained traffic measurement system, in turn, needs necessary tasks to have insight into the network traffic, including Traffic Matrix (TM) estimation, elephant flow detection, and link utilization. These measurement tasks are utilized in a wide range of applications, such as network planning, billing, anomaly detection, load-balancing, traffic engineering, security and various others [1]; thus, it is important to ensure Quality of Service (QoS). Traditional flow-based measurement systems, such as NetFlow [2] and sFlow [3], often employ too much resources (e.g. bandwidth, and CPU) and require significant investments in hardware deployment to satisfy the fine-grained measurement requirements. However, traditional flow-based measurement systems have either a low accuracy or a high deployment cost and consume more resources [4]. An example of the latter is the deployment of NetFlow, which requires setting up of collectors, analyzers and other related services. However, enabling NetFlow in the routers may degrade the packet forwarding performance [5]. Furthermore, NetFlow and similar tools such as Sflow, Jflow, IPFIX, and PRTG are hardware based feature and need to be configured to be set for each interface on the physical device (switch/router).

Unlike traditional network architecture, in the relatively recent Software Define Networking (SDN) architecture, the control plane is separated from the data plane [6]. This enables efficient network management by controlling the entire network through logically centralized controller. The network management is then facilitated through network programmable application which executes through the controller to operate the network devices. Despite the benefits (e.g. capability to provide customizable traffic measurement, and flexible and fine-grain visibility of the network traffic), SDN deployment complicates network measurement systems by introducing a new measurement strategy which may impact

* Corresponding authors.
E-mail addresses: hamid.tahaei@siswa.um.edu.my (H. Tahaei), rosli_salleh@um.edu.my (R. Salleh), badrul@um.edu.my (N.B. Anuar).

http://dx.doi.org/10.1016/j.measurement.2016.10.026
0263-2241/

efficiency and performance of both data and control plane resources, for example imposing overhead in controller and network infrastructure.

Network measurement methods are broadly categorized into active and passive measurement [7]. In active measurement, a probe packet is continuously sent over network path as a request to monitor flows and packet/byte counts. Such methods offer a different level of granularity; thus, imposing significant measurement overhead that may disturb the critical traffic flows. These methods are generally used to calculate one-way delay measurement, Round-Trip-Time (RTT), and adjust the forwarding policies based on the load change. The active measurement demands careful planning to cope with the requirements of a centralized control architecture in SDN. Deploying active measurement devices considerably increases the data acquisition and results in the centralized control mechanism of the SDN hitting saturation [8]. Consequently, acquisition of data streams through distributed devices in the network does not inevitably deliver the SDN controller with the information in the time-frames to minimize the impact of traffic disruption [8]. Even applying fast analytical models along with statistical models (e.g. machine learning techniques and Monte Carlo) at the end-to-end QoS measurement, the controller may faces challenges relating to bottlenecks in communications, control and optimization [9]. However, using passive measurement, real-time traffic is captured and analyzed at the predefined points of network. In this method, network is manually captured and its traffic is directed to an analyzer or agent for further processing. Since there is no probe packet in passive measurement; therefore, it does not cause any overhead. Such method allows for the processing of local traffic states, and global behavior of the network traffic flows passing a specific network points. Passive measurement methods are considered non-intrusive and it does not generate extra traffic in the SDN, but these methods need packet-sampling and statistical methods to conclude the state of the network traffic. Two key limitations of these techniques are (1) inaccurate measurement; because small flows being missed or multiple monitoring nodes beside the SDN flow path samples the similar packet [10], and (2) the necessity for a complicated analytical mechanism to process network traffic at high speed in DCNs.

Till now several efforts have been observed that proposed different methods in SDN with having various QoS requirements to overwhelm challenges associated with real-time per-flow basis network traffic measurement. Most methods focused on the trade-off between accuracy and overheads. However, some published methods require additional hardware or software agents, which is not practical for commodity switch with current technologies or implementation in the controller due to their high complexities in analytical models and the calculation overheads imposed by statistical models. For example, Tootoonchian et al. [11] used active measurement based on constantly polling statistics for each active flow in the network. This introduces overhead during the periodical polling of statistical information through switches across the network as well as overhead in the controller load due to the calculation of all active flows and constant updating. Moreover, due to the limited bandwidth between the controller and the switches, the monitoring traffic results with a bandwidth bottleneck in specifically in-bound deployment. In contrast, FlowSense [12] infers link utilization based on passive capturing of flow arrival and expiration messages which does not incur any overhead. However, the link utilization can only be calculated at the discrete points of the time upon the expiry of the flow. Thus, this is unable to fulfill the dynamic requirement, and moreover, accuracy of the results cannot be sure. From the literature, it appears that issues relating to the overhead of central controller (e.g. number of instructions imposed by execution, calculation and comparison of raw data, such as sampling result and statistical information) are understudied. According to the data from Stanford Computer Science and Electrical Engineering and the study in [13] with 10 different DCNs, the number of active flows is well below 10,000 in DCNs with 5500 active hosts and the average number of active flows at a switch in any time of the second is at most 10,000 flows, respectively. Flows in the DCNs examined are generally $\leqslant$10 KB, and a majority last under a few hundreds of milliseconds. However, new flows can arrive within fast sequence (10 μs) of each other, resulting in high rapid arrival rates. Hence, sampling or actively measuring a huge amount of information in a real-time manner requires a huge computational power (CPU and memory footprint) to lead the CPU/RAM of the central controller saturated. Therefore, it is a promising way to reduce the number of instruction in the controller.

In this paper, we propose a design and a practical solution for accurate real-time TM system which runs on commodity network elements for traffic measurement systems in DCNs. The term "real-time" in practice does not imply the actual time during which a process or event occurs. Rather, it emphasizes the earliest moment in which it crosses the actual time. We also propose a real-time elastic polling schema to adaptively adjust polling frequencies for the statistical request. Our solution is proposed to measure fine-grained monitoring task with less overhead and high accuracy. To the best of our knowledge, this is the first attempt in SDN that proposes fine-grained traffic measurement using aggregated statistics integrated with the group table feature of the OpenFlow 1.3. We then evaluate the utility of our solution using experiments, whose results show that in fix polling approach, our design significantly reduces messaging overhead, controller overhead, and communication cost without compromising on accuracy. Also, findings from the evaluation of our elastic schema demonstrate a considerable improvement in capturing traffic spikes and communication cost.

The remainder of this paper is organized as follows. Sections 2 and 3 present related work and system design, respectively. Section 4 outlines our proposed adaptive polling algorithm, which is used to support our measurement tasks with elastic polling frequency. Findings from our evaluation are presented in Section 5. Finally, we conclude the paper in Section 6.

2. Related work

Basically, flow-based network measurement tools have been introduced in traditional IP networks. NetFlow [2] from Cisco is the premier and the most prevalent, uses a central collector to analyze the sampled or complete traffic statistic. It supports various technologies such as Multi-cast IPSEC and MPLS. Later by releasing version 9, it became a universal standard by IP Flow Information Export (IPFIX) IETF working group. Using this standard, Cisco NetFlow collector can be used by non-Cisco devises. InMon introduced sFlow [3] which uses time-based packet sampling for capturing flow-based IP traffic. Similar to NetFlow and sFlow, Jflow [14] proposed by Juniper Networks, also exploits statistical sampling to analyze flows and monitor detail information about flows. However, all of these monitoring tools mentioned above are commercialized and incur licensing. Moreover, these tools require investment and cost for deployment in the network.

Likewise, network measurement is the subject of recent research focus, partly due to the rapid development of SDNs. There have been some studies which present active and passive measurement methods designed to reduce overhead and increase the accuracy (e.g. solutions for per-flow basis and accurate measurement system with low overhead in SDN-based networks). To reduce the overhead in the controller [15], proposed a measurement framework that uses switches to match packets with a small col-

lection of wildcard rules present in the Ternary Content Addressable Memory (TCAM). However, updating the matching rules, is considered to be a big problem in this approach. In [16] the author proposed a monitoring design to compute appropriate paths for Traffic Engineering (TE) purpose, where network flows are constantly monitored between predefined endpoints for different metrics including throughput, packet loss, and delay. This approach adopts an adaptive fetching approach to pull data from switches, where the rate of statistical queries increases when flow rates differ between samples and decreases when flows are stabilized. In the intelligent traffic (de)aggregation and measurement model presented in [17], TCAM entries of switches/routers are partitioned into aggregated and de-aggregated for fine-grained or course measurement tasks, and important flows are then stamped for direct measurement in the de-aggregate task. The framework proposed in [18] supports different measurement task, where the central controller instructs hash-based switches to gather traffic data, along with the Hierarchical Heavy Hitters (HHH) algorithm for defining significant traffic (large flows). However, the rules are required to be prudently delegated across the network.

The work in [19], applied a prediction based algorithm to count the flows for detecting anomalies due the granularity of measurement along the spatial and temporal dimensions which change time to time. Such an approach allows the anomaly detectors to teach the flow collection module to deliver fine-grained measurement data. In [11], a traffic matrix estimation system is proposed to get flow statistics using simple logic for querying flow table counters with different querying strategies. Such a mechanism gathers active flow statistics on a one-by-one which is considered not to be cost-effective. The work in [12], presented a push-based technique which output with no overhead while measuring the network link utilization. However, this method gets the link utilization at discrete points in time with a lengthy delay; therefore, the scheme does not encounter the real-time monitoring requisite and cannot be extended to other general measurement responsibilities. In [20], authors proposed an adaptive statistical collection algorithm, which emphases on the tradeoff between accuracy and network. This approach has a low overhead and achieves a higher accuracy of statistical collection by capturing traffic spikes. In a similar vein, CeMon [21] proposed a low-cost and accurate monitoring system with three algorithms to adaptively poll switches for statistical collection to optimize the polling cost for all active flows. However, the proposed threshold value in this work is not clearly defined. In addition, the proposed method may result in loss of accuracy in different topology as the greedy switch selection algorithm is highly reliant on the behavior of flows in the network.

Other methods propose different approaches for real-time accurate traffic monitoring. The work in [22], for example, proposed a passive approach to monitor near real-time traffic network by employing the capability of port mirroring that occurs in utmost commodity switches. Port mirroring is an approach to monitor traffic passing through the mirror using a variety of network analyzers and security applications. However, traffic volume can exceed the size of the ports which results the switch to start the process of dropping packets. The OpenSample solution presented in [23] provides a real-time measurement of the network load and individual flows. However, instead of using the OpenFlow measurement mechanisms, this solution influences the sFlow [3] packet sampling functionality present in most of the switches. In [4], authors presented a generic and efficient measurement solution by designing a three-stage data plane pipeline that supports different measurement tasks. However, it is not adapted for OpenFlow, which uses a different flow entity in flow structure in TCAM and thus, requires a special type of switch rather than commodity switches.

3. System design

As mentioned earlier, this paper aims to estimate traffic matrix in datacenter environment. Although, our formulation is mainly proposed for the network topologies with multiple stages of switches, it is capable of generalizing in other network types where the topology applied is Clos [24] like topology. We propose our design for a K-pod fat-tree topology which is a the most popular type of Clos topology that is organized in a tree-like structure (interested readers refer to [24]). In this section, we first provide a general background of OpenFlow and then explain our proposed design with further formulating our research problem.

3.1. Background

In OpenFlow [25], the monitoring task is accredited by the controller which is connected to all the switches via a secure channel interface called southbound interface. The secure channel is established over a TCP connection between the controller and the switch. The controller accumulates the real-time flow statistics from the corresponding switches, and combines the raw data to deliver interfaces for upper-layer applications. When a switch receives the first packet of a new flow in the network, it first checks its flow table to find a match[1] for the flow. Then the flow is forwarded according to the corresponding flow entry in the flow table. In the case of table miss (when there is no match for flow); the switch forwards the first packet header to the OpenFlow controller by a packet-In message. The controller processes the packet header and takes further actions such as setting up the routing path. The controller instructs the corresponding switches along the path for a flow by a packet_out message.

According to OpenFlow specification 1.0 [26], a naive approach to obtain a specific flow statistics in the network is to query it from the switch through the controller using single stat request. In this way, fine-grained per-flow information about a predefined individual active flow is requested with the "ofp_flow_stats_request" stats request type. The predefined active flow is queried based on the exact match of several fields such as input port, source/ destination address, and so on. In OpenFlow specification 1.0, there are twelve flow match fields (Fig. 1). Though, the number of flow match field in OpenFlow specification 1.3 [27] is forty. To query each active flow every time, two messages are transferred in the network (one for the request message from controller to switch and another is the replied message from switch to the controller).

3.2. Architecture

Fig. 2 describes the architecture of our proposed design. Unlike the existing methods that query a flow for one specific link or path, we use a dynamic architecture whose objective is to measure the utilization of all links or paths in a network. In general our measurement system is designed into two steps. The first step is responsible for assigning a set of wildcard rules for polling all the edge switches. In the second step, all the collected stats are aggregated and sent to the analyzer to shape a corresponding TM. There are four main reasons that we poll edge switches in the network: (1) Traffic leaving the edge switches is bursty in nature and the ON/OFF intervals can be characterized by heavy-tailed distributions [28]. (2) Number of flows and their utilizations within the core/aggregation layers are higher than those at the edge [28]. Therefore, we avoid unwanted and overlapping flows, thus reducing memory footprint in the controller. (3) Due to high number of

[1] OpenFlow specification version 1.0 proposes twelve fields for match (Fig. 1). However, the newest specification (version 1.5.1, at the time of writing this article) introduces 44 match fields.

Ingress port	Ether src	Ether dst	Ethern type	VLAN id	VLAN priority	IP src	IP dest	IP proto	IP ToS bits	TCP/UDP src port	TCP/UD P dst port

Fig. 1. OpenFlow flow match table. *Source:* [26].

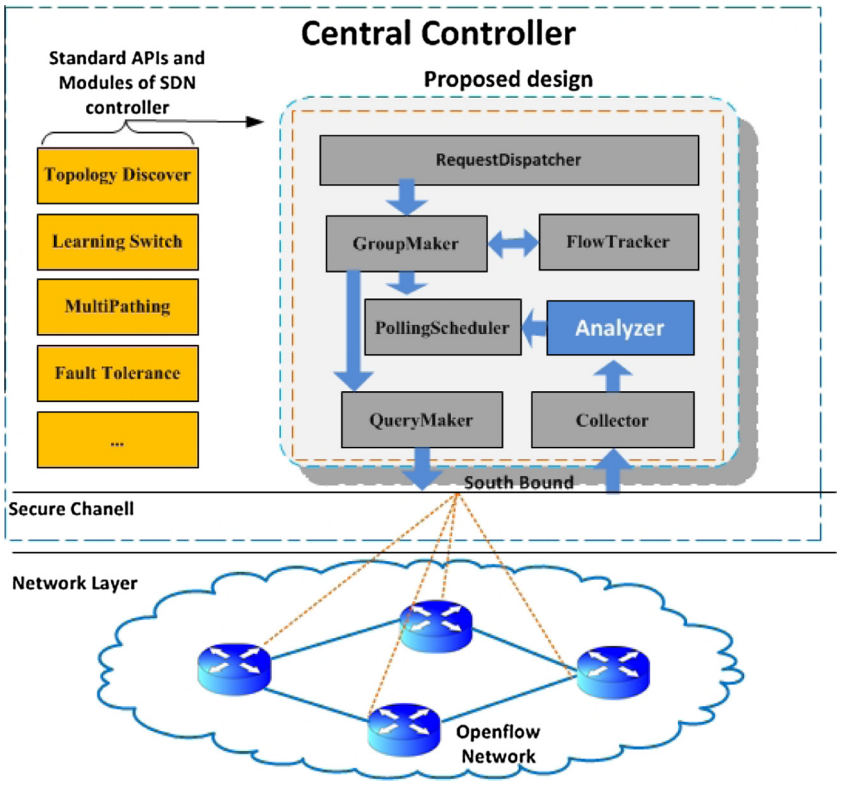

Fig. 2. System architecture.

Table 1
Summary of the main reasons and advantages for selecting edge switches.

Reason	Advantage
More bursty traffic in the edge	Higher probability of detecting large number of flows
High number of flows in core/aggregation layers	Avoiding unwanted and overlapping flows Reducing memory footprint in the controller
Forwarding rules in core layers are often wildcared	Avoiding coarse-grained measurement
Very few hot-spots links are in the edge layer	Estimating utilization before congestion happens

flows in core switches, forwarding rules in these layers are often wild carded. So, the promising strategy to get fine-grained statistics is to look for the finest grained rules which can be found in the edge layer. (4) Last but not least, very few links (only 3% for more than 0.1% of time intervals) in the edge layer are hot-spots [28]. Thus, the utilization is considered before congestion occurs in any point of the network. Table 1 presents a summary of the main reasons and benefits for polling edge switches. Note that, utilization of network paths up to the core layer is calculated on the fly. Therefore, congestion can be detected faster in compare when calculating flow utilization toward down layers (aggregation/edge). Note that we ignore transmission delay as in 10 Mbps Ethernet links are typically on the order of microsecond. Table 2 explains influential components involved in End-to-End delay. End-to-End delay for every source and destination node is calculated by in Eq. (1).

$$Delay_{nodal} = d_{procssing} + d_{queuing} + d_{transsmision} + d_{propagation} \quad [29] \qquad (1)$$

In SDN context, the processing delay is considered to be zero as the process for every flow in a none-edge switch is taken before flow arrival. Moreover, the propagation delay on a physical medium in a network is equal, or a little less than the speed of light which is $2 \cdot 10^8$–$3 \cdot 10^8$ m/s [29]. We ignore queuing delay similarly like previous approaches as for a two or three layer topology design; end-to-end delay is ignorable. Note that the proposed design is provided as a standard northbound application on top of the central controller. Similar to all other northbound application, our design uses the default SDN standard API provided by the central controller. Detail steps of our designed are accomplished by the following modules:

Table 2
Influential components involved in End-to-End delay.

Delay component	Summary
Processing	The time to process packet header by router/switch
Queuing	The time a packet waits in a queue until it can be executed
Transmission	The time taken for the packet's bit pushed into wire or network transmission medium
Propagation	The amount of time for a signal to reach destination from sender

RequestDispatcher: This module translates user level request to machine level commands (e.g., Link utilization, TM matrix, etc.). Then it sends the mentioned information to the GroupMaker module for further actions.

FlowTracker: is responsible to keep all the active flows along with their routing information. Flows and routing information are obtained by standard SDN API such as topology discovery, multipathing (e.g. ECMP), Routing and so on. All the information in this module is then sent to the GroupMaker module.

GroupMaker: This module receives the required command and creates corresponding groups based on the given information provided by the FlowTracker. As we mentioned in Section 3.1, flow is queried based on a predefined match. Therefore, for the sake of reducing the stat reply messages and avoiding overlapping flows, it is promising to query flows based on wildcarding a set of arbitrary fields. Accordingly, the GroupMaker generates stat request by wildcarding all fields whose next hops are a switch rather than end host. Similarly, we query those flows which are going towards aggregation layer and their destinations are not in the same switch source. Thereby, we wildcard only outgoing flows received from the end-hosts.

PollingScheduler defines the frequency of polling behavior for the group(s). We propose an elastic polling scheme in Section 4, which can adaptively adjust the polling frequency based on the ratio of change in utilization. We provide our design in Section 4.

QueryMaker receives all information about switches' groups and polls all the access switches (ToR switches) with exact match on groups defined in the "GroupMaker" module at the predefined intervals.

Collector receives reply messages in each time and forwards them to the Analyzer module. We assign a single module only for collecting the reply message for easy use and configuration in distributed controller. In this case, the collected statistics are sent to the analyzer in another server of controller.

Analyzer: receives all the statistics and calculates the aggregated flow utilization based on their next hop group. We assign each flow in either of pod or core groups. Pod group consists of those flows which come from an attached end host and never go out from the source pod to reach their destination. In the other word, pod group is related to the flows whose destination pods are the same as their source pod. The second group is for those flows which come from an attached host and go up to the core level to reach their destination (different pod). Therefore, the utilization of paths in each pod is the aggregated utilization of its own pod and incoming traffic utilization destined to its pod end-hosts. Recall that all the information about topology such as links, paths, and switches are provided by the standard APIs of the central controller. Following explains pod and core group in more detail:

Pod group: As we mentioned, we consider all flows from all attached end hosts with the same pod destination as their source pod. To calculate the utilization of each link in a pod, flow statistics are grouped by their next hops. For instance, consider a k-pod fat-tree topology as undirected graph G (V, E) where V = {v1, v2, v3, . . ., vn} is the set of switch with $n = |V|$ be the number of switch, and E represent the set of link between switches.

There are $\frac{k}{2}$ flow paths for each source and destination inside a pod where k is the number of pod in our fat-tree network topology. Assume $P = \{p_1, p_2, \ldots, p_i\}$ is a set of flow path where $p_i = \{v_j : v_j \in V,\ j = n\}$ and $P_p = \{p_{p1}, p_{p2}, \ldots, p_{pi}\}$ is the subset of P which are allocated only to the pod. Let $F = \{f_1, f_2, \ldots, f_m\}$ be the set of active flows (universe) and m be the number of flows in the set F. let $F_p = \{f_{p1}, f_{p2}, \ldots, f_{pn}\}$ be a subset of F only streaming in the pod. Therefore, $F_{pe} = \{f_{pe1}, f_{pe2}, \ldots, f_{pen}\}$ represents all incoming flows in a pod where e is the number of edge switches. Consider Uf as utilization of each flow. Thus, utilization of each path *Up* in the pod is a linear function in Eqs. (2) and (3) as follow:

$$U_{pk/2} = \sum_{n=0} Uf_{pen} \tag{2}$$

$$U_{p\acute{k}/2} = \sum_{n=0} U\bar{f}_{pen} \tag{3}$$

Core group: we categorize those flows with different source and destination pods in this group. There are k different paths for a flow going out from a pod to reach a different destination pod in a k-pod fat tree. Thus, we can group the flows based on the k number of core switches and estimate the utilization of all flows which are passing through a specific core switch and destine to a same pod. Therefore, the utilization of each path *Uc* from a source pod to a destination pod becomes the utilization of its corresponding core from the source to destination pod which is mentioned in Eqs. (4) and (5) as follow:

$$U_{ck/2} = \sum_{n=0} Uf_{ken} \tag{4}$$

$$U_{c\acute{k}/2} = \sum_{n=0} U\bar{f}_{ken} \tag{5}$$

The proposed design is placed on top of a logically centralized controller in the network. Moreover, it is capable to run on networks with distributed or a cluster of controllers. In such a scenario, the switch GroupMaker is placed in the root or master controller to instruct other controllers to poll a set of switches.

3.3. Problem formulation

As mentioned in Section 3.1, we can poll flow statistics from switches by "ofp_flow_stats_request" message. The corresponding switch replies flow's information using "ofp_flow_stats" message. Therefore, two messages are generated to measure one flow. Basically, there are two approaches for collecting traffic stats such as exact match of one flow and wildcarding all fields to collect all flows. The former approach generates two messages each time to poll one specific flow. In the later approach, we can poll all the flows information passing through the switch by the same number of messages as in the former approach. Therefore, we can minimize the number of generated messages as well as repeated reply headers. However, excessively applying the second approach causes flow statistics overlapping which imposes extra message interaction overhead and communication cost as well as an overhead in the controller. In this section, we first clarify the problem of objectives above then the proposed solution is described.

3.3.1. Message interaction overhead

Consider the mentioned undirected graph G (V, E) where the number of switches in p_i is s_n. Therefore, the number of message interaction $N_{message}(m)$ for the set F in path p_i in single query approach is a linear function of m in (6).

$$N_{message}(m) = 2 \times m \times s_n \tag{6}$$

As an example, given five flows in Fig. 3 from source A to destination Z, the total number of messages for all switches along the path is $2 \times (5) \times 5 = 50$. Also, $N_{message}(m)$ for polling only edge switch is $2 \times (5) = 10$. On the contrary, we use only one request for all the active outgoing flows from the attached source end-hosts. Thus, the total number of messages for the same example in Fig. 3 is $1 \times (1) = 1$. Now, we can formulate the message interaction (communication) overhead in our design. Given a fat-tree topology as undirected graph, Let K be the number of pods

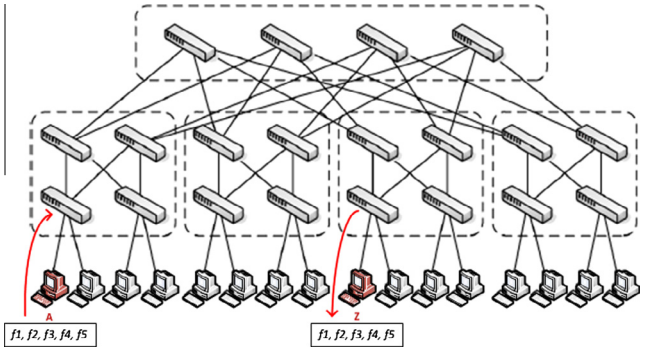

Fig. 3. Fat-Tree Topology (K = 4). The network consists of five flows: f1, f2, f3, f4, f5: A → Z. In general, the number of switches along the path from a source to a destination in different pods is at list five.

in our fat-tree topology and $F_k = \{f_{k1}, f_{k2}, \ldots, f_{km}\}$ represents all active flows in pod K and m be the number of flows. Let e be the number of flows coming from each one of $\frac{k}{2}$ attached hosts to each of $\frac{k}{4}$ switches in a pod. Thereby, the number of communication $N_{message}$ for the graph G and the corresponding match for request message N_{match} is a linear function as shown in Eqs. (7) and (8) respectively. Algorithm 1 describes the steps involve in the construction of query message. The complexity of our GroupMaker module is $O(n)$ in all cases.

$$N_{message}(k) = \frac{k}{2} \times k \tag{7}$$

$$N_{match} = \frac{k}{4} \times f_{kme} \tag{8}$$

3.3.2. Communication cost

According to the OpenFlow specification 1.3 [27], the length of stat request and reply message header in aggregated approach in wire is 122 and 84 bytes. Also, the length of each single flow entry stat in aggregated approach is 144 bytes. Therefore, to measure n specific flows from set F (universe with m number of flows) in a switch using aggregated approach, the communication cost $N_{com}(m)$ (adopted from [21]) in each interval for a switch can be formulated as a linear function in Eq. (9). However, the mention equation in (9) cannot generate optimal solution as the length of reply message in every interval equals to the sum of all active flows in a switch and reply message header. Eq. (10) shows the optimal length of aggregated reply messages for n specific flows.

$$C_{com}(m) = l_{replyheader} + m \times l_{singleflowentry} \tag{9}$$

$$C_{com}(n) = l_{replyheader} + n \times l_{singleflowentry} \tag{10}$$

Algorithm 1: QueryMaker algorithm

Input: $G(V, E)$, $F_k = \{f_{k1}, f_{k2}, \ldots, f_{km}\}$, Group
Output: a set of flow match
function QueryMaker (Input (graph G), array (F_k))
  match ← array [flow match] = null;
  c = 0;
    for each $f \in F_k$ **do**
      if (($f = f_{kme}$) **and** (*fkme* is in the group))
        match [c] = f_{kme}.match();
        c ++;
end for
end function

3.3.3. Controller overhead

As mentioned in Section 1, by the controller's overhead we mean the numbers of instructions imposed by execution, calculation and comparison of raw data to process information. The performance and throughput of CPU is measured from different perspectives such as Cycles Per Instruction (CPI), Million Instruction Per Second (MIPS), and Transaction Per Second (TPS). Thereby, the CPU instruction rate is calculated by dividing the observed CPU cycle speed by the observed CPI [9]. However, determining the number of instruction applied by CPU requires to obtain the job's information (calculation of statistic reply) comprised of multiple tasks each of which consists of multiple threads, which is out of the scope of this paper. A simpler criterion to observe the imposed overhead is to presume a constant value x indicating the number of instruction taken by a single flow stat reply in CPU. Therefore, the controller overhead $C_{overhead}(n)$ analyzing n specific flows from set F (universe with m number of flows) in each interval is formulated as a linear functioned of m in Eq. (11). Whereas, the optimal solution applies n as a function for calculating the mentioned overhead as shown in Eq. (12).

$$C_{overhead}(m) = Number\ of\ switch \times number\ of\ reply\ message \times m \times x \tag{11}$$

$$C_{overhead}(n) = Number\ of\ switch \times number\ of\ reply\ message \times n \times x \tag{12}$$

3.3.4. Solution

The optimal solution for all the objectives above is to wildcard all the specific flows n out of the universe flow set m. This is quite infeasible except aggregating all desired flows and wildcarding them based on one specific feature. To this end, we use "group table" feature available in OpenFlow 1.3. The OpenFlow 1.3 introduces "group table" in which every flow entry can point out to a group as its action. A group can either have a single or a list of action/bucket. In general, there are four types of group such as "All", "SELECT", "INDIRECT", and "FAST-FAILOVER", each of which has specific features (interested readers are refer to [27] for more information). In our design, we use "SELECT" type in which each packet entering the group is sent to a single bucket associated with its action. Thus, for this group in a switch we define all potential output actions related to a packet. In such a case, all the incoming flows from the end-hosts are grouped without any intervention to the forwarding decision and central policy enforcement. Therefore, the action of a flow entry is set to the action or a list of actions for that group. Fig. 4 shows the SELECT group type in OpenFlow 1.3. As our design polls only Edge/ToR switch, it applies group on only edge/ToR switches for decreasing complexity. This approach significantly reduces overhead in the analyzer and reply message length due to two main reasons: (1) The TCP acknowledgment replies

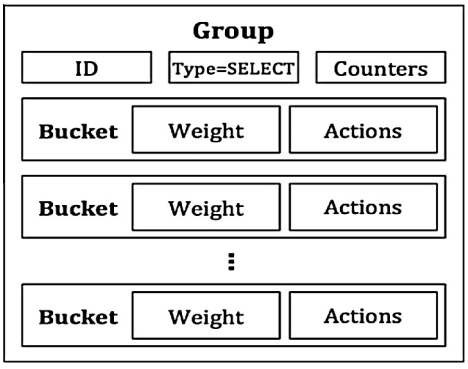

Fig. 4. The SELECT group (adopted from [30]).

flows are not collected at the source node. Therefore, we do not calculate 50% of statistics in source nodes (sender switch) for TCP flows. (2) We calculate all flows only at source node not at both source and destination nodes. Therefore, for all the flows in the network we are saving 50% of the calculation.

Integrating group table with aggregating desired active flows in the network provides the feasibility to wildcard a set of specific flows for the purpose of fine-grained aggregated measurement. In this case, the optimal number of flows in every interval is captured. This capability accomplishes the requirement to meet optimal solution formulated in Eqs. (7), (8), (10), and (12).

4. Elastic polling scheme

In this section, we propose an elastic polling schema to adaptively adjust polling intervals. Our goal is to achieve accurate and timely link utilization, while incurring little network overhead.

4.1. Overview and background

In normal flow statistics collection, the switch is polled after a pre-defined fixed rate of samples which is known as fixed sampling [21]. In this extend, acquiring a highly accurate flow statistic requires a high frequency (low polling interval) polling system. Using fixed sampling approach results in low controller overhead. However, the fixed sampling waste resources in a situation where the traffic is slow and cannot cope up with the traffic spikes in a timely fashion. On the other hand, high frequency polling will induce significant monitoring overhead in the network though it is capable to track instant traffic changes and identify spikes with more probabilities.

To strike a better trade-off between the flow statistic collection accuracy and incurred network overhead, we propose an adaptive flow statistical collection with a variable frequency algorithm to adjust the polling frequencies. The rationale behind this idea is to poll more frequently in those flows which have more fluctuations in bandwidth usage (frequent changes in utilization volume). On the contrary, we poll less frequently for flows which have stable utilization.

Prior works made a close consideration towards an adaptive polling architecture to adjust polling frequencies in different scenarios. The work in [20] proposed an adaptive monitoring scheme, in which a threshold value (100 MB) is compared with the difference between the previous and current byte count of the flow. If the byte count difference is above the threshold, the polling scheduler is divided by the constant α (6 s).

Otherwise, the scheduler is multiplied by small constant β (2 s). Therefore, it maintains a higher polling frequency for flows that contribute to the link utilization, and it preserves a lower polling frequency for flows that do not significantly contribute towards the link utilization at that moment.

Similar to [20], the work in [21] proposed three different algorithms. The first two algorithms "Tuning Sampling Frequency" and "Proportional tuning" take more consideration to the traffic factor and the smoothing factors to adjust the sampling frequency based on historical data (data collected from previous intervals). However, the author ended up with "Sliding windows based Tuning" where the byte count difference of a flow is compared solely against itself. Therefore, when the traffic does not change a lot, the window size should be expanded to keep the recent data stable. Otherwise, the windows size should be decreased quickly to be responsive to instant traffic spikes.

For our research work evaluation, we have implemented all aforementioned algorithms. As a result, we found that the existing adaptive polling schemes are sub-optimal as flows' behavior in DCNs change time to time, specifically TCP flows which are very bursty in nature. Also, selected existing algorithms poll all the flows individually for a specific destination to calculate the corresponding link utilization. This brings a significant overhead in the network when the number of flows increases and their utilization have a significant fluctuation.

4.2. Design

Our elastic polling scheme is designed into two layers. In the first layer, we apply aggregated stat request for all the active flows passing through a specific link. Our design is able to create any aggregate request for every destination or node/port in the network by setting a group on the match request. In the second layer, we apply our adaptive algorithm to adjust the polling intervals to achieve high accurate link utilization with a minimum overhead.

4.2.1. Applying aggregated stat request for link utilization

As we explained in Section 3.2, all the routing information about all flows are kept in flowTracker module. Therefore, we assign a group for all flows which contain a specific link (port on a switch). Then we set the exact match of stat-request to a specific selected group. Note that, our design does not intervene or manipulate the forwarding policy of the network. Accordingly, original output actions for all flows remain intact. So, upon receiving a request for monitoring the utilization of specific link, following steps are applied:

- All the active flows in FlowTracker are checked out to see whether they are contributing to the link or not.
- The GroupMaker module creates a group on every edge/ToR switch that represents a source (attached to end-host) for the aforementioned flows.
- When a new flow arrives, step 1 is applied and added to the group.
- Expired flows are removed from the group upon receiving an expiry signal by flow removal message.

Using this approach, our design requires creating the stat request message on all Edge/Tor switches based on only one exact match. Thus, in the worst case, for all flows in the network, it generates flow stat request message the same as the number of edge/ToR Switch. This is a considerable reduction in network overhead where the number of stat request message in other methods [11,20] is the same as number of flows.

Algorithm 2: Elastic polling scheme

Input: Utilization of flows' group in time t_n, and $_{tn-1}$, sw_{n-1}
Output: Next polling time (τ)
function ElasticPolling (Input (Utilization t_n and t_{n-1}), sw_{n-1})
    dn = Calculate $Pd_{(Utn,Utn)}$ // Eq. (8)
  sw = dn
if ($sw_n - sw_{n-1} > 20$ %) {
    $\tau_{n+1} = \min(\tau Max, \tau_{n\times 3})$;
    }
Else{
    $\tau_{n+1} = \max(\tau Min, \tau_{n\times 0.5})$;
}
end function

4.2.2. Elastic polling scheme

The main objective in an adaptive polling schema is to capture traffic spike in real time with a minimum overhead. We adhere that the adaptive algorithm should decrease the polling frequency

for stable flow groups while it increase the frequency for high fluctuated groups with significant contribution in bandwidth usage (not stable utilization). Our proposed algorithm is similar to SWT in [21]. We argue that, it is necessary to take into consideration the ratio of the flows' group utilization on the link utilization with its previous interval. Thereby, we develop a sliding window which regulates the polling intervals according to the flow group behavior. We keep tracking of the percentage of the difference between byte count of flows' group in the current interval and the previous one. We update the sliding window with the mentioned percentage value in each time interval. We also keep updating the last difference value in the sliding window. For every new interval the following steps are applied for its next interval:

- The value of sliding window (percentage of difference) is compared with the new value. If the difference between these two values is less (less than 20%), it means the utilization does not have a high fluctuation. Therefore, we reduce the polling frequency by 50% of the last time.
- If the difference is more visible (above 20%), we double the frequency by 600% of the current interval. Thus, when there is a relatively high fluctuation, we poll statistics in shorter intervals.

The linear function of percentage ratio of a flow group is shown in the Eq. (13). The rationale behind is to increase polling frequency according to the ratio of fluctuation rather than byte count difference. Algorithm 2 shows the pseudo-code related to our elastic polling scheme.

$$Pd_{(Utn,Utn-1)} = \frac{(|U_{tn} - U_{tn-1}|)}{(tn + U_{tn-1}) \div 2} \times 100 \quad (13)$$

5. Evaluation

We evaluate the performance of our proposed design from different perspectives such as accuracy of TM estimation, communication cost, message overhead, and the controller overhead. All the experiments are conducted using Mininet (SDN emulator) on an Intel i5-2400 3.10 GHz and 16G RAM. We use the Mininet version 2.2.1 [31] network emulator to evaluate our design in a controlled and repeatable environment. Mininet is using a Linux containers to emulate hosts and Open vSwitch (OVS), which allows the entire network to be emulated in a single computer. Mininet uses Linux traffic shaping to emulate fixed-speed links by facilitating the emulated network realistic congestion and queueing delays. We set the speed of the link to 10 Mbps to have faster experimental of our experimental emulation. We implement a prototype of our design as a module in the Floodlight controller. The controller uses Equal Cost Multiple Path (ECMP) routing for forwarding decisions. We replicate the testbed benchmark of Hedera [32] with identical topology. In our proposed design emulation, we generate flows by Iperf version 2.0.5 [33]. All end hosts are set to server and client at the same time. Traffic source, destination, duration, volume, intervals, and type (UDP/TCP) are all set to a uniformly random fashion respectively. A ratio of 49–90% (adopted from [28]) is defined to generate traffic that traverses the network interconnection. We purposely, defined the mentioned ratio to emulate DCN traffic behavior in our testbed. We used different TCP and UDP ports to impose abundant active flows. In this case, we run our experiment with different active flow's number. We evaluate our design with two polling interval frequency: N = 500 and 1000 ms respectively. Our goal is to compare the accuracy, communication cost, message and the controller overhead in fixed polling interval. We then compare the accuracy, and message overhead in adaptive polling interval against the previous counter-polling–based approaches OpenTM in [11], Payless in [20], and CeMon in [21]. For the purpose of capturing traffic spikes, Payless [20] and CeMon [21] proposes their adaptive polling system with the minimum and maximum polling intervals of 0.5 and 1 s respectively. We present the result of fixed and elastic polling schema in the subsequent sections as follows. Table 3 summarizes the detail of specifications involved in our experiment.

5.1. Fixed polling

5.1.1. Accuracy

We define the accuracy, as the real-time accuracy of traffic matrix (TM) estimation and accurate flow ratio (AFR) which is obtained by the number of accurately measured flows over the total number of flows. We then discuss about our result with MCPS method proposed by CeMon, an aggregated-flow polling method proposed in [21]. CeMon uses a greedy algorithm for selecting switch to poll a set of switches. If the greedy algorithm takes to consideration, a slight fault proportion is imposed on the AFR ratio. On the other hand, if original algorithm proposed in [21] is considered without its greedy strategy the AFR ratio is not affected as a result of choosing all switches to poll in a fault-free manner.

Fig. 5 shows the accurate flow ratio and TM estimation accuracy in a fat tree k = 4. Unsurprisingly, we achieved the same amount of flow and byte counts in traffic matrix (TM) estimation which represents the error between the measured and the real TM. In AFR ratio, we did not observe flow or byte loss in our experiment. However, it worth mentioning that this ration requires more experiments with a higher number of flows. We observed the average accuracy of TM always stands above 98.4% in all the scenarios with a different number of active flows. However, accuracy is highly associated with the polling intervals (initial time of polling and actual time when flows are started). We also manually set the initial polling time to the flow arrival time; our design proved that the accuracy of TM with 99.9% which is closed to optimal solution is feasible. However, setting the initial polling interval in aggregated approach is infeasible in real scenarios. The result of accuracy in TM estimation is similar to MCPS in CeMon. However, for AFR in MCPS there is a loss ratio imposed by the greedy algorithm which

Table 3
Detail specifications of experiment.

Specification	Detail
SDN emulator	Mininet
Switch type	Open vSwitch(OVS) OpenFlow 1.3 enabled
Traffic generator	Iperf version 2.0.5
Traffic Type	Randomized TCP/UDP
Network topology	Three layered fat-tree K = 4 pod
Polling intervals	For fix polling 500–1000 ms, for elastic polling a range between 3000 and 5000 ms

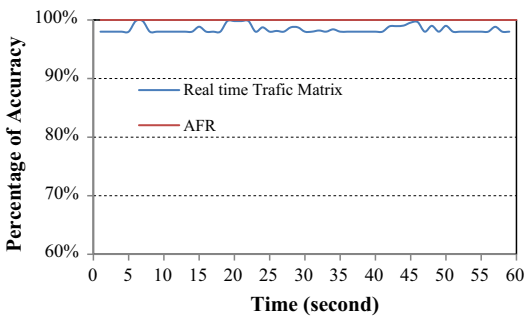

Fig. 5. Accurate of flow ratio and TM estimation accuracy in a fat tree k = 4.

chooses switches for polling. According to the results, if the loss ratio is taken into account, AFR drops and fluctuates around 90%. For the purpose of realization, the loss switches in MCPS is generated in a uniformly random manner. These experiments demonstrate that our design is able to saves the accuracy of flow ratio without loss of accuracy in TM estimation.

5.1.2. Communication cost

We define the communication cost by the total bandwidth usage for flow stat reply message in every interval. This parameter becomes essential for in-band SDN deployment where the network bandwidth is shared between management and forwarding paths. In this section, we elaborate the reduction of communication cost by our design. Then, we compare it with a basic per-flow polling method proposed in [11] and MCPS in [21]. We set the polling interval to 5 s and a maximum number of flow is 3200 in number. The flow arrival rate is 8% of total flows in every 5 s (there is 92% similarity for two consecutive polling). We measure the total communication cost in 60 s time using Wireshark network analyzer tool. Fig. 6 shows the total communication cost for each 5 s. As it can be seen in the figure, the total cost of OpenTM sharply grows and continues to the end. The growth of OpenTM's cost has a direct relation with the number of new flows and their distribution in the network. We observe that although CeMon improves the total communication cost over OpenTM, these two methods have almost similar behavior. CeMon, takes the current active flows in an aggregated reply but it generates a single reply message for every new flow. Thus, it improves the cost for polling those flows which are already exists in the network. On the other hand, our design applies aggregated polling for all flows (current and new flows) in the network. Therefore, it generates one statistic request in every interval for all the flows in a switch which imposes 0% cost for a new flow in a single reply message. In total communication cost, our design saves total cost roughly 8% and 44% over CeMon and OpenTM respectively. However, in higher flow distribution with bigger granularity (e.g. new flow arrival is 50% and 80% of total flows), our design improves more than 31% and 53% over CeMon respectively.

5.1.3. Message overhead

In this section, we evaluate the overhead of our design in terms of the number of request messages sent from the central controller for polling the switches. We compute the overhead at every 1 s of the time interval. Fig. 7 shows the total number of request messages. OpenTM, starts by 240 request messages and continues with a sharp and steady increase until the end with 3200. This sharp increase has a direct relation to the number of active flows which are being contributed to the TM. Thus, the number of request messages is increased at the expense of active flows number. On the other hand, CeMon takes the increment of request message slightly

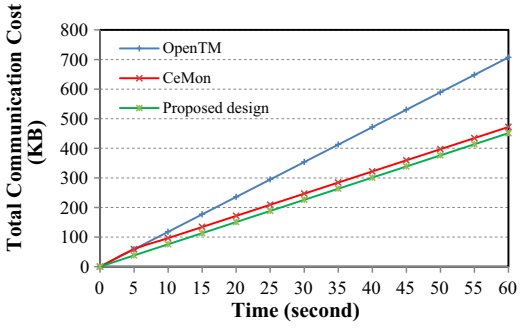

Fig. 6. Total communication cost with 3200 flow number.

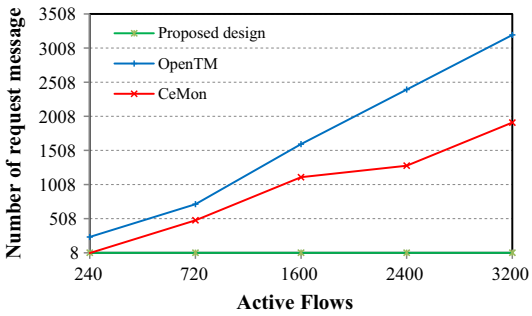

Fig. 7. Number of request message in 60 s.

Table 4
Number of switches for polling.

Number of switch	Proposed design	CeMon [21]	OpenTM [11]
Worst case	8	4	1
Best case	1	1	1

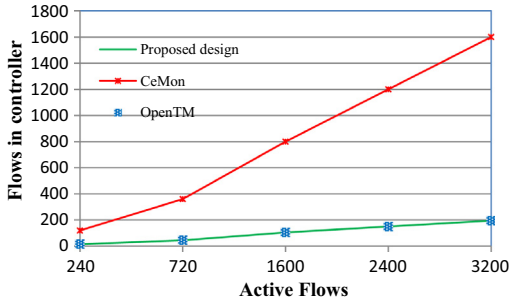

Fig. 8. Total numbers of flows processed by the controller.

slower. It starts by 4 request messages with a total number of 240 active flows until it finishes by 1682 request with 3200 active flows.

This is due to using aggregate request in one message. It aggregates flows in every interval which is similar to the previous interval. We observed that CeMon improve the number of request messages by roughly 46% over OpenTM. Different from OpenTM and CeMon which the number of request messages is depended on the number of flows, in our design, the number of request message is identical to the number of edge switches in the network (8 in this evaluation). This number always remains the same from time to time with different flow number. The experiments above demonstrate that our design can save the number of request messages more than 90% over CeMon. Table 4 explains a general view of the total number of switches which is polled by the central controller. As it can be seen in the table, all three approaches take only one switch for the polling purpose in the best case scenario. On the other hand in the worst case scenario, the switch number is 8, 4, and 1 for our design, CeMon and OpenTM respectively. It can be inferred that all the traffics imposed by request and reply messages (communication cost) are generated from one switch in OpenTM. In contrast, our frame work and CeMon distribute generated traffic from the mentioned metrics over several switches.

5.1.4. Controller overhead

We evaluate the total number of flows which is processed in the controller in every specified interval. Fig. 8 explains the total number of flows processed by the controller in every interval (1 s). The native approach in OpenFlow (single stat request) is considered as

the optimal number of flows for the TM construction. The reason is that the controller requests the exact number of flows which contribute to the construction of TM. OpenTM similarly applies this approach to measure flows. Therefore, OpenTM's controller overhead is the most optimal solution in terms of flow's number being processed by the controller. In the proposed design in this paper, due to grouping desired flows which contribute to TM construction, the controller is given the exact flows in every response message which is the optimal number of flows resemble OpenTM. However, the proposed design applies polling aggregated flow stats rather than polling strategy per-flow used in OpenTM. On the other hand in CeMon, the number of flows processed by the controller in every interval is started by 120 and finished with a sharp increase by 1600 in 3200 active flows. This experiment shows that our design reduces the number of flows sent to the central controller is reduced by more than 87% over CeMon. Thus, reducing controller utilization more than 87% results to minimize calculation in the controller.

5.2. Elastic polling scheme

To evaluate the effectiveness of the elastic polling scheme, we analyze the accuracy of link utilization, controller overhead, communication cost and the number of request message interaction for 60 s time with flow dataset in [21]. By accuracy we mean, the actual utilization of link to capture the traffic spikes. We measure the actual traffic utilization and the number of message using Wireshark. The idle_timeout for flows are set to the original value in the Floodlight controller which is 5 s. The link utilization measurement interval is set to 1 s. The initial sampling intervals for all algorithms are set to 1 s. The minimum and maximum polling interval for our scheduling algorithm is set to 500 and 3000 ms respectively. Table 5 summarizes the detail of specifications involved in our experiment.

5.2.1. Accuracy

In this section, we evaluate the effectiveness of our adaptive polling system in terms of accuracy of link utilization in 60 s of the time. We compared our result with two similar adaptive techniques SWT in CeMon [21], Payless [20] and a baseline scenario (periodic polling). Fig. 9 shows the result of our design is compare against the actual link utilization capture through Wireshark. The accuracy result obtained by our design shows that it could capture most of the traffic spike except a short duration from 35 to 40 and 50 to 60 s of time. CeMon and Payless also miss several traffic spikes and shift them from current interval into the next one which leads these methods to issue inaccurate result. In a total duration of the 60 s time, our design improves the accuracy to capture traffic spikes at least by 35% over Payless. Surprisingly we observed that payless achieves 12% better efficiency in capturing traffic spikes.

5.2.2. Message overhead

Fig. 10 shows the number of request message sent by the controller in 60 s time. The fixed polling system (periodic polling) takes the highest number of messages. In this experiment, the average number of message interaction for periodic polling is roughly 9.5 per second. Since Payless applies a threshold (10% of NIC), it sends averagely 6 messages per second as the threshold is continuously triggered for most of the flows. On the other hand, CeMon and our elastic take the difference of flow's fraction in every interval with its previous interval. Both of these methods obtain lower message number. In CeMon, the ratio of utilization in two consecutive intervals is mostly taken into consideration while we apply the percentage difference for two intervals against their previous value. We observed that our elastic algorithm reduces the average number of message by 109% and 130% over CeMon and periodic polling respectively.

Table 5
Detail specifications of experiment.

Specification	Time (s)
Idle_timeout	5
Measurement interval	1
Initial sampling intervals	1
Minimum polling interval	0.5
Maximum polling interval	3

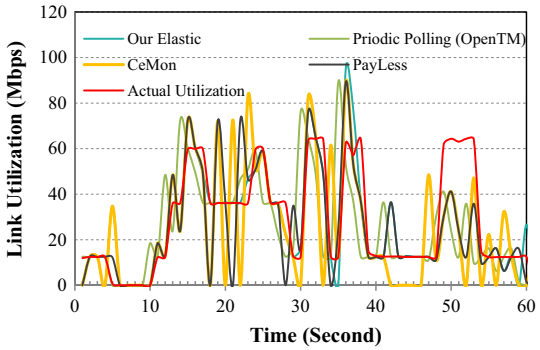

Fig. 9. The measured link utilization by elastic polling schema.

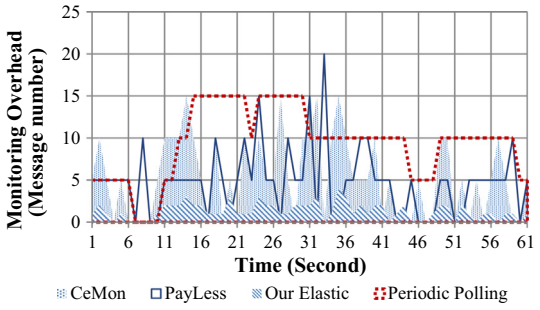

Fig. 10. Number of request message sent by the controller.

6. Conclusion

In this paper, we presented a real-time measurement system for SDN-enabled datacenters. Specifically, we designed a comprehensive monitoring system that actively polls the switches to collect real time statistics for high accurate and low overhead traffic measurement. We also proposed two novel designs for fixed elastic polling systems to accommodate various monitoring purposes. We then applied the group table feature of OpenFlow 1.3 with active measurement approach to aggregate a set of desired flows into one stat reply messages. Our design focuses on DCN traffic measurement using aggregated polling strategy with the support of group feature in OpenFlow 1.3. To demonstrate the practicality of our proposed system, we conducted extensive experiments whose findings indicated that our elastic method can capture traffic spike with a 12% improvement and at least a 109% reduction in message overhead in comparison to previously published methods. The finding also shows an 87% save in controller overhead by polling aggregated statistics. Also, we observed 10% and 31% improvement in message overhead and communication cost over similar methods. The results obtained from the fix polling system showed that our design achieves a significant edge over previous approaches.

The focus of this paper has been on TM estimation, link utilization and capturing traffic spikes. Future work will include extending the system to cater for distributed controller platforms in large DCNs using SDN.

Acknowledgment

The work of the last author (i.e. Nor Badrul Anuar) is supported by University Malaya Research Grant Programme (Equitable Society) under grant RP032B-16SBS.

References

[1] C.-W. Chang, G. Huang, B. Lin, C.-N. Chuah, Leisure: load-balanced network-wide traffic measurement and monitor placement, IEEE Trans. Parallel Distrib. Syst. 26 (2015) 1059–1070.
[2] B. Claise, Cisco systems NetFlow services export version 9, 2004.
[3] P. Phaal, M. Lavine, Sflow version 5, Specification.sFlow.org, 2004.
[4] M. Yu, L. Jose, R. Miao, Software defined traffic measurement with OpenSketch, in: Presented as Part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13), 2013, pp. 29–42.
[5] G.R. Cantieni, G. Iannaccone, C. Barakat, C. Diot, P. Thiran, Reformulating the monitor placement problem: optimal network-wide sampling, in: Proceedings of the 2006 ACM CoNEXT Conference, 2006, p. 5.
[6] S. Khan, A. Gani, A.A. Wahab, M. Guizani, M.K. Khan, Topology discovery in software defined networks: threats, taxonomy, and state-of-the-art, IEEE Commun. Surv. Tutor. PP (2016) 1.
[7] V. Mohan, Y.J. Reddy, K. Kalpana, Active and passive network measurements: a survey, Int. J. Comput. Sci. Inf. Technol. 2 (2011) 1372–1385.
[8] S. Sezer, S. Scott-Hayward, P.K. Chouhan, B. Fraser, D. Lake, J. Finnegan, et al., Are we ready for SDN? Implementation challenges for software-defined networks, IEEE Commun. Mag. 51 (2013) 36–43.
[9] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, J. Wilkes, CPI 2: CPU performance isolation for shared compute clusters, in: Proceedings of the 8th ACM European Conference on Computer Systems, 2013, pp. 379–391.
[10] M. Jarschel, T. Zinner, T. Höhn, P. Tran-Gia, On the accuracy of leveraging SDN for passive network measurements, in: Australasian Telecommunication Networks and Applications Conference (ATNAC), 2013, pp. 41–46.
[11] A. Tootoonchian, M. Ghobadi, Y. Ganjali, OpenTM: traffic matrix estimator for OpenFlow networks, in: International Conference on Passive and Active Network Measurement, 2010, pp. 201–210.
[12] C. Yu, C. Lumezanu, Y. Zhang, V. Singh, G. Jiang, H.V. Madhyastha, FlowSense: monitoring network utilization with zero measurement cost, in: International Conference on Passive and Active Network Measurement, 2013, pp. 31–41.
[13] J. Naous, D. Erickson, G.A. Covington, G. Appenzeller, N. McKeown, Implementing an OpenFlow switch on the NetFPGA platform, in: Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems, 2008, pp. 1–9.
[14] A.C. Myers, JFlow: practical mostly-static information flow control, in: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1999, pp. 228–241.
[15] L. Jose, M. Yu, J. Rexford, Online measurement of large traffic aggregates on commodity switches, in: Hot-ICE, 2011.
[16] N.L. Van Adrichem, C. Doerr, F.A. Kuipers, Opennetmon: network monitoring in openflow software-defined networks, in: Network Operations and Management Symposium (NOMS), IEEE, 2014, pp. 1–8.
[17] M. Malboubi, L. Wang, C.-N. Chuah, P. Sharma, Intelligent sdn based traffic (de) aggregation and measurement paradigm (istamp), in: IEEE INFOCOM 2014 – IEEE Conference on Computer Communications, 2014, pp. 934–942.
[18] M. Moshref, M. Yu, R. Govindan, Resource/accuracy tradeoffs in software-defined measurement, in: Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, 2013, pp. 73–78.
[19] Y. Zhang, An adaptive flow counting method for anomaly detection in SDN, in: Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies, 2013, pp. 25–30.
[20] S.R. Chowdhury, M.F. Bari, R. Ahmed, R. Boutaba, Payless: a low cost network monitoring framework for software defined networks, in: Network Operations and Management Symposium (NOMS), IEEE, 2014, pp. 1–9.
[21] Z. Su, T. Wang, Y. Xia, M. Hamdi, CeMon: a cost-effective flow monitoring system in software defined networks, Comput. Netw. 92 (2015) 101–115.
[22] J. Rasley, B. Stephens, C. Dixon, E. Rozner, W. Felter, K. Agarwal, et al., Planck: millisecond-scale monitoring and control for commodity networks, ACM SIGCOMM Comput. Commun. Rev. 44 (2015) 407–418.
[23] J. Suh, T.T. Kwon, C. Dixon, W. Felter, J. Carter, OpenSample: a low-latency, sampling-based measurement platform for commodity SDN, in: 34th International Conference on Distributed Computing Systems (ICDCS), IEEE, 2014, pp. 228–237.
[24] M.F. Bari, R. Boutaba, R. Esteves, L.Z. Granville, M. Podlesny, M.G. Rabbani, et al., Data center network virtualization: a survey, IEEE Commun. Surv. Tutor. 15 (2013) 909–928.
[25] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, et al., OpenFlow: enabling innovation in campus networks, ACM SIGCOMM Comput. Commun. Rev. 38 (2008) 69–74.
[26] O.S. Consortium, OpenFlow switch specification version 1.0.0, December, 2009.
[27] O.S. Consortium, OpenFlow switch specification version 1.3.0, December, 2012.
[28] T. Benson, A. Akella, D.A. Maltz, Network traffic characteristics of data centers in the wild, in: Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement, 2010, pp. 267–280.
[29] J. Kurose, K. Ross, Computer Networking: A Top-Down Approach, sixth ed., Pearson, New Jersey, USA, 2012.
[30] R. Izard, Fast-Failover OpenFlow Groups Available: <https://floodlight.atlassian.net/wiki/display/floodlightcontroller/How+to+Work+with+Fast-Failover+OpenFlow+Groups>2016.
[31] Mininet version 2.2.1, <http://mininet.org/overview/> (2.2.1 ed.).
[32] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, A. Vahdatc, Hedera: dynamic flow scheduling for data center networks, in: NSDI, 2010, 19–19.
[33] A. Tirumala, F. Qin, J. Dugan, J. Ferguson, K. Gibbs, Iperf: The TCP/UDP bandwidth measurement tool, 2005, <http://dast.nlanr.net/Projects>.