Computer Communications 171 (2021) 99–111

Contents lists available at ScienceDirect

Computer Communications

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

Sieve: A flow scheduling framework in SDN based data center networks

Maiass Zaher [a,*], Aymen Hasan Alawadi [a,b], Sándor Molnár [a]

[a] Department of Telecommunication and Media Informatics, Budapest University of Technology and Economics, Magyar tudósok körútja 2, Budapest, H-1117, Hungary
[b] Department of Computer Science, Faculty of Education, University of Kufa, P.O Box 21, Kufa, Najaf, Iraq

ARTICLE INFO

Keywords:
Data Center Network (DCN)
Flow scheduling
Software Defined Networks (SDN)
Mice flow
Elephant flow

ABSTRACT

Today's data centers act as the primary infrastructure for emerging technologies. QoS imposes requirements for more attentive techniques that can deal with different characteristics of traffic classes and patterns. In this context, network flows can be classified into large and long-lived flows called elephant flows and mice flows, which are small and short-lived flows. According to the characteristics of the emerging technologies, e.g., IoT and Big Data, mice flows are dominant; Hence, it is crucial to improve Flow Completion Time (FCT) for such delay-sensitive flows. This paper presents Sieve, a new distributed Software Defined Networks (SDN) based framework. Sieve initially schedules a portion of the flows based on the available bandwidth despite their classes. We propose a distributed sampling technique which sends a portion of the packets to the controller. Furthermore, Sieve polls the edge switches periodically to get the network information rather than polls all switches in the network, and it reschedules elephant flows only. Mininet emulator and mathematical analysis have been employed to validate the proposed solution in 4-ary Fat-Tree DCN. Sieve provides less FCT up to around 58% for mice flows and maintains throughput of elephant flows compared to Equal Cost MultiPath (ECMP) and Hedera.

1. Introduction

Nowadays, many enterprises employ various services and applications by leveraging data center networks. As a result, the associated flow patterns impose new considerations. In particular, the majority of flows generated by applications like MapReduce send less than 10 KB of data and lasting for less than 10 s, and similar characteristics are obtained in case of web services [1]. In this context, the proliferation of IoT's prospective applications may boost these characteristics since the flow size of such applications very small [2].

Basically, flows in DCN can be classified into mice and elephant flows [3]. Specifically, mice flows are known as the smallest and shortest-lived flows, and they are sensitive to the delay [4]. On the other hand, the elephant flows are long-lived flows and more affected by the available bandwidth [3]. Although, there are a smaller number of elephant flows than mice flows in DCN, but they carry more than 80% of the transferred data [5]. Typically, elephant flows endeavor to utilize the link capacity fully. Therefore, mice flows could suffer from real-time latency [6]. Studies show that any delay in the response time of data center applications profoundly impacts the user experience [7]. For example, in the case of increasing the flow delay with 400 ms, Google found that the number of daily searches reduced by 0.6% [8]. Therefore, enhancing the DCN utilization involves minimizing FCT [9]; Hence, namely, we aim to mitigate FCT of delay-sensitive flows, i.e., mice flows, as well as to maintain the throughput of bandwidth-hungry flows which are elephant flows.

In this context, SDN paradigm provides reliable and effective techniques to handle network resources and information centrally. SDN controller can collect statistical information about various network resources and events, including information about network flows. As a result, many researches applied SDN significantly for improving QoS, e.g., by flow scheduling and traffic load balancing [10,11].

The key contribution of this paper is to propose a flow scheduling algorithm that achieves the following objectives:

1. Propose a heuristic, adaptive, and scalable scheduling algorithm to schedule elephant and mice flows. The proposed algorithm is based on the available bandwidth to mitigate FCT of mice flows and to maintain throughput of elephant flows.
2. Propose a balanced scheduling scheme which divides the flow scheduling burden between Sieve and ECMP.
3. Propose a flow detection technique to provide enough information about a portion of network flows.

Sieve attempts to resolve two dilemmas:

* Corresponding author.
E-mail addresses: zaher@tmit.bme.hu (M. Zaher), aymen@tmit.bme.hu (A. Alawadi), molnar@tmit.bme.hu (S. Molnár).

https://doi.org/10.1016/j.comcom.2021.02.013
Received 2 June 2020; Received in revised form 11 January 2021; Accepted 7 February 2021
Available online 15 February 2021

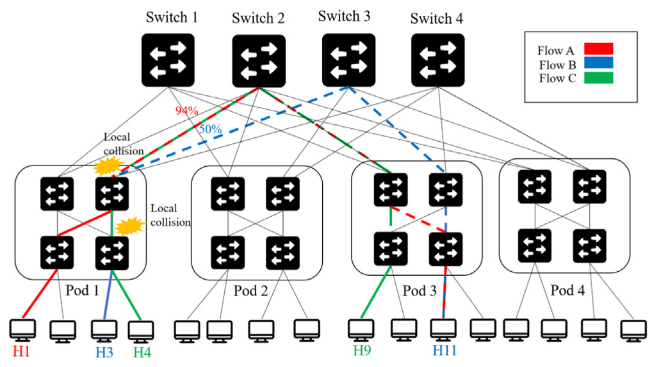


Fig. 1. Motivation example on flow collisions inside 4-ary Fat-tree DCN in which packet collisions yielded as a result of static hashing based routing employed by ECMP.

1. Polling information about every flow provides full flow visibility on the controller, but overwhelms the control plane.
2. Flow scheduling based on ECMP only provides rapid scheduling, but prevents any scheduling optimization by control plane.

This paper is structured as follows: Section 2 gives background information about the addressed problem. Section 3 surveys related works. We present the Sieve framework and its design aspects in Sections 4 and 5 respectively, and we evaluate and discuss its performance in Sections 6 and 7. We conclude our work in Section 8.

2. Preliminary

The typical design of DCN topology includes multi-rooted layers that it has many paths between each pair of hosts [12]. Therefore, finding a suitable path is challenging since congestion avoidance requires considering the current load and potential conflicts. The motivation example, depicted in Fig. 1, shows 4-ary Fat-tree DCN, which contains many-to-one communication pattern regardless of the flow size. Two senders, H1 and H3, connected to pod 1 initiate two mice flows to H11 in pod 3. Assuming that there was a background elephant flow transferring data from H4 to H9. Employing ECMP only in such scenario probably incurs many collisions, as shown in Fig. 1, since it adopts scheduling based on static hashing of packet header where congestion occurs due to overwhelming some links. This kind of congestion will profoundly reduce QoS [8] of mice flows. Hence, deploying applications in data centers leveraging ECMP does not guarantee QoS due to the probability of data loss and retransmissions. However, flow contentions and bottlenecks are inevitable [13]. Therefore, rescheduling flows from a path to another based on flow's bandwidth consumption only, such as in Hedera [12], might produce another kind of congestion and flow completion delay. Consequently, improving QoS in DCN requires considering the characteristics of both flow classes and the network situation.

3. Related work

We classify the studies into two categories. One for the central solutions reside in the control plane, while the other for the studies assign a portion of the operations to the data plane.

3.1. Central solutions

Several studies have emerged dealing with flow scheduling in DCN. Al-Fares et al. [12] proposed Hedera, and they assumed a flow which consumes more than 10% of link capacity as an elephant flow. Then, Global First Fit algorithm determines the best available path for this flow. On the other hand, Hedera employs ECMP to schedule all undetermined flows, so FCT of mice flows is not considered as an evaluation metric. Curtis et al. in [10] proposed Mahout to detect elephant flows in DCN on end-hosts by inspecting socket layer. Mahout periodically polls switch counters to optimize elephant flows scheduling using first-fit algorithm. Although the authors tried to mitigate the detection overhead on the controller, deploying such method requires modifications in the end-host kernel. DevoFlow [14] schedules elephant flows only, and it provides no FCT measurements of mice flows. MiceTrap [15] is a mice flows detection and scheduling algorithm based on finding underutilized paths. Nevertheless, the authors did not evaluate their solution. Yazidi et al. [16] classified links into hot and cold links to reschedule the detected elephant flows to the least congested links. However, the mechanism monitors the demand of all flows, and this solution results in processing overhead. L2RM [17] balances the load in fat-tree DCN. L2RM reschedules flows after two consecutive predefined threshold hits to maintain the balance between DNC links. However, L2RM treats both flows classes similarly.

3.2. Distributed solutions

Distributed solutions aim to diminish the overhead on the control plane. In this context, a fine granularity has been proposed by flowlet, which is a data unit defined in [18] as a packet burst. Flowlet is leveraged in CONGA [19] to achieve optimal flow scheduling in leaf-spine DCN by employing leaf switches to feedback on congestion metrics. However, CONGA requires custom switching ASICs. Wang et al. [20] employed end-hosts to detect elephant flows to be scheduled by the controller. On the other hand, EMCP schedules the remaining flows on the switches. BLEND [21] uses end-hosts to track all outgoing elephant flows and to select paths for the remaining flows locally. However, modifying end-hosts kernel is not a feasible solution. The work presented in [22] provides different QoS for mice and elephant flows by creating specific queues for each traffic class. Hence, It involves creating and monitoring many queues. Afek et al. [23] proposed a sampling technique using OpenFlow group feature. They preserved the flows information in a data structure into the controller to track the total counter of each sampled flow. Besides, a unique flow entry for a candidate elephant flow is installed after passing a predefined threshold. Then, it polls the candidate flow entries to detect if it is an elephant flow indeed. As a result, the controller will be involved in more processing. Similarly, Tang et al. [13] proposed a flow classification model for both mice and elephant flows by sampling *packet-in* packet of each flow to identify aggregated flow category; Therefore, this solution is exposed to misclassification. Hermes [24] is a congestion-aware load balancing technique. Hermes proactively schedules flows when congestion or failure occurs. The method depends on ECN (Explicit Congestion Notification) and RTT (Round-Trip Time) for congestion detection. Although Hermes is deployable since it does not require hardware changing, all data center end-hosts need to take part in the sensing method. Therefore, without end-host visibility, the sensing approach cannot be achieved. CAPS [25] is a congestion aware technique based on end-hosts. The solution contains three modules; packet encoder and decoder on each host of the data center besides packet spraying module based on Random Packet Spraying (RPS) on the ToR switch. Traffic flows are divided into mice and elephant flows, where the elephants are scheduled using ECMP to specific paths, and mice flow scattered to all available paths based on RPS. However, this technique requires modifications in end-hosts and the availability of RPS capable switches. Luopan [26] is a distributed congestion aware approach implemented based on sampling routing paths between ToR switches to direct the flowcels to less congestion path. However, the method operates at flowcell granularity at a threshold of 64 KB. There-

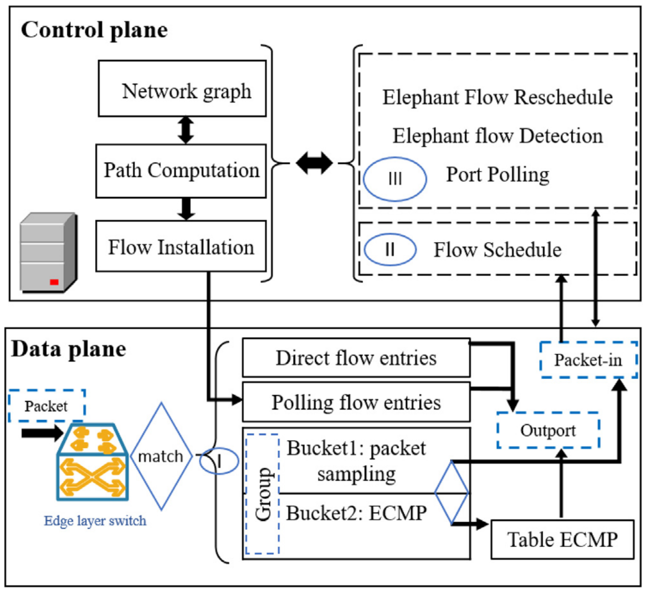
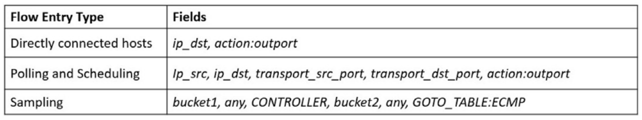


Fig. 2. The proposed framework architecture.



Flow Entry Type	Fields
Directly connected hosts	*ip_dst, action:outport*
Polling and Scheduling	*Ip_src, ip_dst, transport_src_port, transport_dst_port, action:outport*
Sampling	*bucket1, any, CONTROLLER, bucket2, any, GOTO_TABLE:ECMP*

Fig. 3. Flow entry types in Table 0 on edge switch.

fore, end-host NIC needs to be managed to generate flowcells. On the other hand, the process of path sampling is performed by packet propping, which requires TCP/IP modification.

Therefore, based on the presented related works, our solution is different from others since it is distributed, but it does not incur the overhead resulted from end-hosts contacting. Furthermore, it does not require any modifications in kernel nor hardware. In addition, Sieve mitigates the overhead due to its functions are applied on a portion of flows only.

4. The proposed Sieve framework

The framework architecture is depicted in Fig. 2. Basically, the proposed framework aims to guarantee QoS of mice flows by classifying network flows so that mice flows can be served with less delay. We propose three layers framework as depicted in Fig. 2. The first layer resides in the data plane. It employs OpenFlow bucket group feature to provide packet sampling and ECMP-based scheduling. The second layer resides in the control plane. It contains the required functionalities to schedule the sampled flows using *shortest-path available-bandwidth* mechanism. The last layer resides in the control plane also. It contains the polling technique and elephant flows rescheduling functions.

In the following, we describe the proposed framework modules in more details.

4.1. Flow sampling

Sieve employs weighted bucket group feature of OpenFlow to sample flows as well as to mitigate the overhead on the control plane. In practice, the data plane does not send every packet to the controller,

Table 1
Sampling group entry.

group_id	group_type	bucket_action
group_id=1	select	bucket=weight:50,actions=CONTROLLER, bucket=weight:50,actions=GOTO_TABLE:ECMP

Table 2
ECMP group entry.

group_id	group_type	bucket_action
group_id=1	select	bucket=weight:50,actions=OUTPORT:1, bucket=weight:50,actions=OUTPORT:2

but rather the first packet of a flow, i.e., *packet-in*, is either scheduled based on ECMP or sent to the controller. We implement this mechanism using a group entry consists of four fields:

1. *Group identifier:* It is 32 bits integer value used as a unique identity.
2. *Group type:* We use "select" group type, so the switch executes one bucket's action based on the value of packet header hashing and bucket's weight value.
3. *Counter:* Represents the number of packets matched by the group.
4. *Action bucket:* An action associated with a specific bucket and applied to the matched packets.

Recall that the first packet of a flow did not match the direct nor polling flow entries on an edge switch, as shown in Fig. 3, so it will be handled according to the sampling group entry presented in Table 1. Since the type of group is select, the switch chooses one bucket of actions based on its weight. In particular, the switch will hash the packet header information; then, based on the hash value and the values of buckets weights, one bucket will be selected to apply its associated action. Specifically, the switch will either forward the packet to the controller or to ECMP table with probabilities of 0.5 and 0.5, respectively. Sieve samples flows only from edge switches, so only edge switches contain the sampling group entry presented in Table 1. Specifically, edge layer switches contain two flow tables which are Table 0 shown in Fig. 3 and Table ECMP contained ECMP-based scheduling group entries as shown in Fig. 2. This partial sampling saves the controller from overwhelming samples load. Our framework treats the sampled packets equally; regardless they belong to elephant or mice flows.

4.2. ECMP-based scheduling

We utilize ECMP in scheduling a portion of flows since ECMP represents a fast scheduling technique. Therefore, we can avoid flow collisions which could happen in case of sole dependence on ECMP. In particular, we implement ECMP-based scheduling by proactively defining bucket group with equal weights into all switches of edge and aggregate layers. ECMP group entry is presented in Table 2. Specifically, Sieve employs ECMP to schedule the flows forwarded out switches ports 1 or 2, on edge layer connected to the aggregate layer and on aggregate layer connected to the core layer only, based on packet header hash value and buckets weights values. On the other hand, Sieve uses proactively defined flow entries for directly connected hosts and subnets for forwarding the flows from an upper layer toward a lower layer. Besides, polling flow entries are used for scheduling flows in either direction. Fig. 4 presents the flow entries effective directions. However, different priority values are used to enable preferential scheduling. As a result, edge layer switches contain two flow entry tables, whereas one flow entry table exists into all other switches. The

Table 3
Flow tables in different switch layers and flow entry types with an indication whether they are proactively predefined or reactively defined.

Flow entry type	Table_id	Switch_layer	Proactive/Reactive
Directly connected host	0	edge	proactive
Polling and Scheduling	0	edge/agg/core	reactive
Directly connected subnet	0	agg/core	proactive
ECMP-based scheduling	0	agg	proactive
Sampling	0	edge	proactive
ECMP-based scheduling	ECMP	edge	proactive

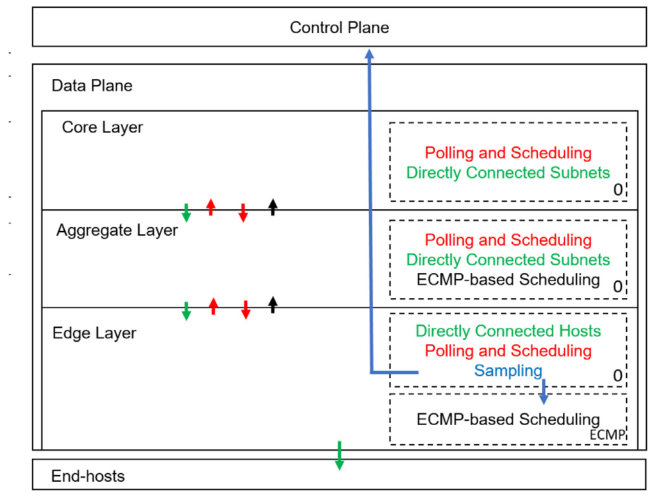


Fig. 4. Flow entry effective forwarding directions where each flow entry type is created to forward traffic in dedicated directions across the Fat-tree DCN layers and end-hosts as well.

proactively and re-actively installed flow entries are depicted in Table 3 along with the table id, location and their types. Table 3 is descendingly sorted based on priority values of the flow entry types according to each table_id.

4.3. Flow schedule

Upon receiving *packet-in* at the control plane, Sieve parses the packet header to retrieve the connection information. Then, *Path Computation* module will be invoked to find the four shortest paths between the source and destination, which has the best available bandwidth as well. As shown in Algorithm 1, *bottleneck_of_path* function will be invoked for each shortest path in *shortest_p*, which contains information of the four shortest paths, to compute its bottleneck link, which is the highest loaded link. Then, the path with maximum bottleneck bandwidth will be the best one, *best_p*, to schedule the new flow onto. Finally, this module sends the connection information and the best path to *Flow Installation* module, which installs a new polling flow entry into all switches along the chosen path. In this layer, Sieve serves new flows similarly regardless they are mice or elephant flows.

4.4. Port polling

Polling_stat function in Algorithm 2 periodically polls port statistics every *PR* from all switches in the network by sending OpenFlow *OFPPORTStats* message. Therefore, the framework has full available-bandwidth visibility. Sieve saves this information on a directed graph. Furthermore, port information used by it to reschedule elephant flows by comparing the occupied bandwidth of edge switch ports, connected to the aggregate layer, with the predefined threshold. Threshold refers to the bandwidth occupation on an edge switch port in this paper unless otherwise indicated. Specifically, when the available bandwidth on any edge switch port, connected to the aggregate layer, passes the predefined threshold $U\_BW < Th$, this module will invoke *Elephant Flow Detection* module by invoking $reschedule(i, j, U\_BW)$ function, as shown in line 6 of Algorithm 2.

4.5. Elephant flow detection

Sieve employs polling flow entries on edge switches to find the sent byte counts. In particular, each polling flow entry is defined uniquely, and it consists of four-tuple fields which are source IP, destination IP, source transport port, and destination transport port. In particular, upon threshold hits, Sieves fetches all elephant flows forwarding out a specific edge switch port, i.e., j, and whose accumulated sent bytes is more than 50 KB, i.e., $F\_load > 50\ KB$, as well, as shown in lines 9–12 of Algorithm 2. In this context, we consider a flow whose size more than 50 KB as an elephant flow. We adopt this classification principle according to the results in [1,5].

4.6. Elephant flow reschedule

This module then receives the total number of elephant flows that are to be rescheduled, i.e., num_redir_EFlows, as shown in line 13 of Algorithm 2. In addition, the information of each elephant flow, the edge switch and the port, are sent from the previous module, as presented in lines 14–16. Furthermore, this module tries to find a new path that the original edge switch port is not a part of it as shown in lines 18–22. Then, It invokes function *get_best_path_by_bw* to compare the bottleneck available bandwidth of the possible new paths with the available bandwidth on the edge switch port. Then, if an alternative path has sufficiently higher available bandwidth than that on the original port as in line 30, it sends the information of the elephant flow and the new path to *Flow Installation* module; otherwise, a log message will be displayed that no path met the specified conditions.

In case only one elephant flow exists on an edge switch port upon threshold hits, Sieve tries to reschedule it to another path with higher available bandwidth. On the other hand, if there are many elephant flows, the framework tries to reschedule as many as of them. It proportions the available bandwidth on the original edge switch port to the number of existed elephant flows as in line 13.

4.7. Path computation

This module receives requests from *Flow Schedule* and *Elephant Flow Reschedule* modules. In particular, the former module needs a path for a new sampled flow upon receiving *Packet-in* message from the data plane, and the later invokes this module upon threshold hits to reschedule as possible as of elephant flows on a specific edge switch port, i.e. function *bottleneck_of_path* of Algorithm 1 and function *get_best_path_by_bw* of Algorithm 2, respectively. This module tries to find the shortest paths between *src_ip* and *dst_ip* then selects the one whose bottleneck link has the maximum available bandwidth. Therefore, this module fetches the network graph, *G*, containing the available bandwidth. Finally, it sends the chosen path to the *Flow Installation* module.

4.8. Flow installation

As depicted in Table 3, edge switches have proactive flow entries for directly connected hosts and buckets group entry. Specific priority values have been assigned in descending order, as shown in Table 3. However, this module installs new polling flow entries with incremental priority based on the last priority value provided that it cannot be more than that of the directly connected hosts flow entries. Polling flow entries are installed into all switches along the path.

4.9. Network graph

The gathered network information is saved on a directed graph (G) to represent the network topology and situation. Hence, operations like finding the shortest path between any two hosts $<S, D>$ can be achieved by Dijkstra algorithm based on edges weights (G, S, D, W). The algorithm starts searching the graph G for the shortest and most-available-bandwidth path from the source node (S) to the destination node (D) by computing the edges' weights (W) which are the free available bandwidth. We present our proposed second layer functions in Algorithm 1, and the third layer functions in Algorithm 2. The variables used by Sieve framework are presented and described in Table 4.

Table 4
Variables used by Sieve.

Variable	Description
min_bw	Available bandwidth on the bottleneck link
max_bw	Maximum available bandwidth along a path
k	The scale of Fat-tree topology
shortest_p	List of the four shortest paths between src_ip and dst_ip
best_p	The best path between src_ip and dst_ip based on hop count and available bandwidth
F_bw	Available bandwidth on an edge switch port
Th	Threshold of the bandwidth occupation
U_bw	Utilized bandwidth on an edge switch port
dpid_list	List of switch IDs
PR	Polling rate
EF_list	List of active elephant flows on an edge switch port
Paths	List of the shortest paths excluding a specific edge switch port
port_list	List of port IDs

Algorithm 1 New Flows Scheduling

```
Input: G=(V,E) , src_IP, dst_IP, src_port, dst_port,
min_bw = capacity, max_bw = 0, k,
shortest_p= { }
Output: best_p= [ ]
1.for i in (0, k):
2.    if shortest_p [i]= shortest_paths (G, src_IP, dst_IP)
3.for j in shortest_p:
4.    min_bw = bottleneck_of_path (G, j, min_bw)
5.    if min_bw > max_bw:
6.        max_bw = min_bw:
7.        best_b = j
8.    return best_p
9.Function bottleneck_of_path (G, j, min_bw):
10.   min_bw_link = min_bw
11.   for i in (0, len(j)):
12.       bw = G[j[i]][j[i+1]]
13.       min_bw_link = min (bw, min_bw_link)
14.   return min_bw_link
```

5. Framework design aspects

Algorithms 1 & 2 describe the functionalities of the framework. This section presents the design aspects and decisions of the framework.

5.1. Problem formulation

Network is modeled as a directed graph $G = (V, E)$, where V: set of the nodes, E: set of the directed edges, p: a path where $p = (v_0, v_1, \ldots, v_n)$, $\forall v_i \in V$, $\forall i \in [0, n]$, $n \in \mathbb{Z}$. However, the connection throughput is limited to the available bandwidth on the bottleneck link of a path as shown in Eq. (1). In particular, the load of any link e is $\frac{l_e}{C_e}$, where l_e is the currently used fraction of its capacity C_e as shown in Eq. (3) where s_i^e is the ith flow size. Therefore, the condition in Eq. (2) should be maintained to avoid congestion on path p, so that the utilization on any link along path p should be smaller than the bottleneck capacity link. Let us assume that flows arrive at a switch according to Poisson process with rate λ and size s, and the predefined threshold of flow size is T. Hence, the number of elephant flows until time t is as in Eq. (4) where F is CDF of flow sizes. In particular, elephant flows on path p_1 and path p_2 are $N_1(t)$ and $N_2(t)$, respectively. Portion of $N_1(t)$ should be redirected to p_2 if $N_1(t) > 0$ and $C_{p_1} > Th$ and $C_{p_2} < Th$, where Th is the threshold of triggering elephant flow rescheduling. Consequently, the maximum number of elephant flows should be redirected to path p_2 is computed so that the condition in Eq. (1) is maintained.

$$C_{p_i} = min \quad C_e, \quad \forall e \in E \tag{1}$$

$$\frac{l_e}{C_e} < C_p \tag{2}$$

$$l_e = \sum_{i=1}^{N(t)} s_i^e \tag{3}$$

$$N(t) = \lambda(1 - F(T))t \tag{4}$$

5.2. Flow detection

There are many principles to classify flows in data center networks. In the case of consumption based classification [12], the flow throughput must be tracked periodically. In this context, the phase of flow detection starts by polling flow statistics from switches in terms of source IP, destination IP, source port, destination port and Byte count. Hence, any flow consumes bandwidth more than the predefined percentage of the link capacity is identified as an elephant flow. However, this methodology discards the large link capacity sizes in recent DCN. As a result, considering the percentage consumption as a classification criterion does not guarantee a rapid reaction for mice flows since they have a very short life span.

The other methodology employs flow size as a classification criterion [5]. In particular, whenever the cumulative flow size hits the predefined threshold T, it will be considered as an elephant flow. We adopt this methodology to distinguish between elephant and mice flows. Specifically, we follow the fact discovered in study [5], where more than 80% of the flows in DCN had less than 10 KB.

Algorithm 2 Detect and Reschedule Elephant Flows

```
Input: G= (V, E), F_BW, Th, U_BW, min_bw= Capacity,
dpid_list, max_bw= 0, k, shortest_p= { }, PR, EF_list= { },
Paths= { } , Portid_list
Output: best_p= [ ]
1.Repeat each PR
2.  Function Polling_stat (dpid_list):
3.    OFPPortStatsRequest (dpid_list)
4.    If $U\_BW_{ij}$ < Th : ; i ∈ pdid_list, j ∈ Portid_list
5.      OFPFlowStatsRequest(i)
6.    Reschedule (i,j, $U\_BW_{ij}$)
7.Function Reschedule (i, j, $U\_BW_{ij}$):
8.    E_count = 0
9.    For f in F_list:
10.       If F_load > 50 KB & Output_port == j:
11.           EF_list.append(f.info)
12.           E_count++
13.   num_redir_EFlows= E_count × $\frac{U\_BW(i,j)}{Capacity(i,j)}$
14.   If num_redir_EFlows > 0 :
15.       For f in (0, num_redir_Eflows) :
16.               get_best_Path (G, i, f.info, $U\_BW_{ij}$,
                  num_redir_EFlows)
17.Function get_best_Path (G, i, F.info,$U\_BW_{ij}$,
                          num_redir_Eflows)
18.   For P in Shortest_P:
19.       If (link (P[i] P [i+1]) != j):
20.           Paths.append (P)
21.       else:
22.           Continue
23.   best_p = get_best_Path_by_Bw(i, G, Paths, $U\_BW_{ij}$)
24.   return best_p
25.Function get_best_Path_by_Bw (i, G, Paths, $U\_BW_{ij}$)
26.   min_bw= Capacity
27.   max_bw= $U\_BW_{ij}$
28.   For P in Paths :
29.       min_bw = bottleneck_of_path (G, P, min_bw)
30.       If (min_bw > max_bw and
          min_bw - $U\_BW_{ij}$ > 1Mbps) :
31.               max_bw = min_bw
32.               best_p = P
33.   If (best_p):
34.       return best_p
35.   else:
36.       print "No path met the conditions"
```

5.3. Flow sampling

Sieve employs edge layer switches to sample a portion of flows by sending the first packet, i.e., *packet-in*. Sieve installs unique polling flow entries into the switches along the chosen path to forward packets belonging to the sampled flows. Subsequently, upon threshold hits, i.e., the occupied bandwidth on an edge switch port gets more than 25%, our framework will detect elephant flows forwarding out of the port based on the cumulative bytes count of the corresponding installed polling flow entries.

However, this solution discovers just a portion of the total DCN flows so that a fraction of the flows are not sampled. Assuming that flows arrive at an edge switch according to a Poisson process with a rate of λ. Hence, a portion of flows will be scheduled based on ECMP, and the rest will be scheduled based on Sieve after sampling their first packet. Let *x(t)* is a random variable represents the number of arrived flows at the time interval [0,t] and *s(t)* is a random variable represents the cumulative flow size in the network during [0,t].

Theorem 1. *The cumulative flows size sampled by the controller is approaching the half of the total size of flows in the network as t increases to infinity, i.e.,* $\lim_{t\to\infty} n(t)s(t) \to \frac{x(t)s(t)}{2}$ *n(t) is the number of sampled flows until time t.*

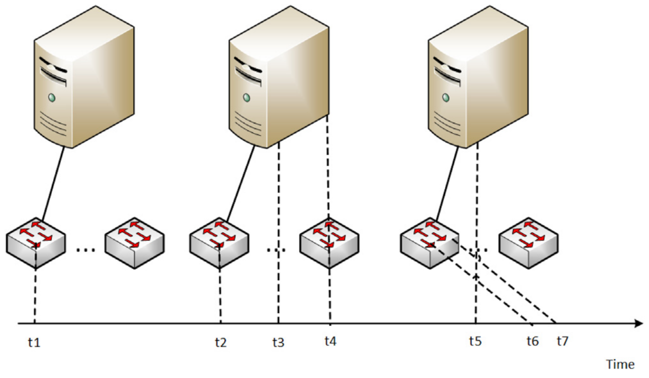


Fig. 5. Estimate the error resulted from the delay between the time instants of exchanging control messages and reaction instant by the controller.

Proof. Mitigating the load on the controller is our aim so that it will not be overwhelmed by samples. Let us assume a probabilistic scheme by applying the strong law of large numbers. In this context, each flow has a probability p_c of being sampled by the controller or p_e of being scheduled by ECMP. To denote the event of sampling the first packet by the controller, we use $I_j^c = 1$ with probability p_c; in contrast, $I_j^e = 1$ with probability p_e indicates the event of ECMP based scheduling. Therefore, the expected number of the sampled flows is $E[I_j^c] = p_c$, and similarly the expected number of the flows scheduled based on ECMP is $E[I_j^e] = p_e$. Note that the variances of these values are equal to equation Eq. (5) and Eq. (6) respectively:

$$V[I_j^c] = p_c(1 - p_c) \tag{5}$$

$$V[I_j^e] = p_e(1 - p_e) \tag{6}$$

The cumulative size of the sampled flows resulted by the proposed sampling process is, regardless of whether these flows are mice or elephants, represented in Eq. (7) and Eq. (8) of both cases, where S denotes the flow size:

$$B_c = \sum_{j=1}^{N(t)} I_j^c S_j \tag{7}$$

$$B_e = \sum_{j=1}^{N(t)} I_j^e S_j \tag{8}$$

By applying the strong law of large numbers, the total cumulative size of these flows will be $B_t = \frac{B_c + B_e}{2}$. Therefore, Sieve can manipulate about half of the DCN transferred data between end-hosts connected to different edge switches since we defined proactive flow entries for directly connected hosts; Hence, flow conflicts and congestion due to the sole dependence on ECMP can be mitigated.

5.4. Mitigate obsolete information

As shown in Fig. 5, the polling messages sent from the switches sequentially to the controller. Hence, there is a delay between the instant of sending the statistics at t_2, as a reply to the request at t_1, and the instant of taking a decision by the controller at t_4. The controller receives the statistical information message at t_3 then it sends the decision at t_5. Furthermore, the decision could lead to installing new flows in a switch at t_7 after receiving it by the switch at t_6. This delay is due to the number of switches, controller activities and network conditions. Consequently, some obsolete decisions could be taken, so the error estimation of the taken decisions can be computed as in Eq. (9). Assuming τ is the delay between t_2 and t_4, P_{rate} is the polling rate,

C is the capacity of the link connecting a switch with the controller, N is the switches number, M_{len} is the length of the reply message.

$$\tau = \frac{N \quad M_{len} \quad P_{rate}}{C} \tag{9}$$

$$C_l = N \quad P_{rate} \tag{10}$$

$$P_{rate} = \begin{cases} 10^{\frac{B_{edge} - \frac{\sum_{i=1}^{4k} V_i}{4k}}{B_{edge}}} & if \quad \frac{\sum_{i=1}^{4k} V_i}{4k} \leq B_{edge} \\ T_{base} = 2sec & otherwise \end{cases} \tag{11}$$

Based on Eqs. (9) and (10), the delay is related to the controller load C_l, where they are directly proportional to each other. For example, given the size of statistics message M_{len} = 112 Bytes,[1] the link capacity connects the controller to the data plane is 1Gbps and there are 100 switches with P_{rate} = 2 s. Under these conditions, the delay can be maximum 179.2 μs. Moreover, the polling period influences network stability; therefore, the polling period should be dynamic. Eq. (11) is used to compute the dynamic values of the polling rate. Since it is not necessary to probe the data plane when the average utilization of the edge switch ports is far from the threshold value of the bandwidth occupation, i.e., B_{edge}, where V_i is the utilization of port i and k is the Fat-Tree scale, i.e. $k = 4$. T_{base} is 2 s, and it is the default polling interval. Thus, the value of P_{rate} will not grow too much, and the controller can still probe the data plane when ports utilization under the predefined threshold. In particular, P_{rate} can extend from 1 s under high traffic until 10 s under light traffic to maintain the accuracy and to avoid the overhead, as shown in Eq. (11). Besides, the polling rate's default value can be used in the case of the average port utilization is more than the occupation threshold at any instant. As a result, based on Eqs. (9) and (11) and as the numerical example, the delay in our case, i.e., $N = 20$, will be 35.84 μs which ensures delivering of up-to-date statistical information. Furthermore, based on the real traffic measurements in [1], P_{rate} value range is efficient since 25% of Web service, 85% of Cache and 25% of Hadoop flows are last for more than 1 s Therefore, P_{rate} value range is feasible to take rescheduling decisions for elephant flows where Sieve probes statistics at a rate whose value is within the elephant flows' life span.

5.5. Controller overhead

Let us assume that the sampling probability is p; hence, the controller receives a *packet-in* packet with a probability p. Likewise, n is the number of the sampled packets out of the total arrived packets, x.

Theorem 2. *The total number of packet-in, n, sampled to the controller is $\ll \frac{x}{2}$ given that x is the total number of the packets arrived to an edge switch*

Proof. Let us assume f is the number of flow entries on an edge switch, and c is the count of packets forwarded according to a flow entry. Consequently, $n(t)$ could be asymptotically computed as in Eq. (13).

$$x = n + cf \Rightarrow n = x - cf \tag{12}$$

$$\begin{aligned} n(t) &= \int_0^t x(t)dt - \int_0^t cf(t)dt \\ &= t^2 \left(\frac{x}{2} - \frac{cf}{2}\right) \end{aligned} \tag{13}$$

Accordingly, the maximum value of $n(t) \ll 50\%$ of the total number, since over the time $c\&f$ will get larger, and the load on the controller will be less consequently. For example, let us assume that no more new flows arrived at an edge switch after some time, so $cf \approx x$ which yields no more packets will be sent to the controller.

In the following, we analyze the expected overhead of processing new flows with Sieve. We set up an numerical study to inspect the number of the new flows, Sieve has to process in case of real DCN parameters. Sieve samples a portion of the new flows by employing two buckets. Assuming that number of the sampled flows is half of the total number of flows. We consider a Fat-tree DCN with real parameters as shown in Table 5.

[1] OpenFlow Switch Specification Version 1.3.1 Section A.3.5.6.

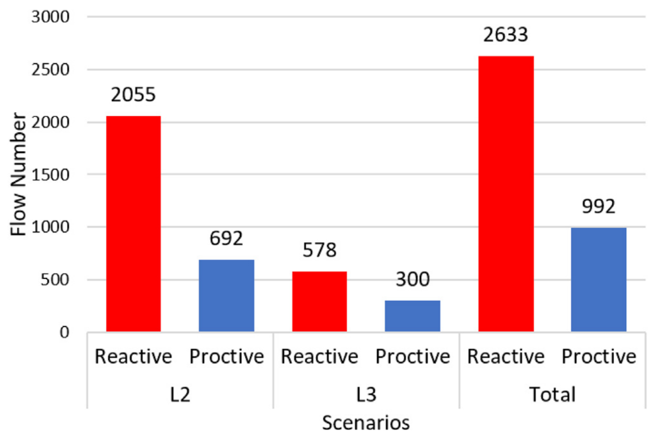


Fig. 6. Flow entries numbers generated in case of proactive and reactive schemes.

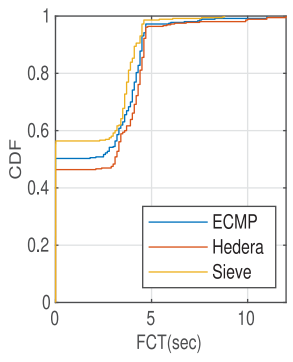
Table 5
Parameters and values of the controller overhead evaluation.

Parameter	Description	Value
H	Number of end-host	10000 [5]
R	Number of edge switches	578 [27]
S	Number of total switches	1445 [27]
F	Median inter-arrival time	2 ms [1]
B	Link bandwidth	10 Gbps [1]
P	Packet size	1500 Byte
P_{rate}	Default value of the polling rate	2 sec

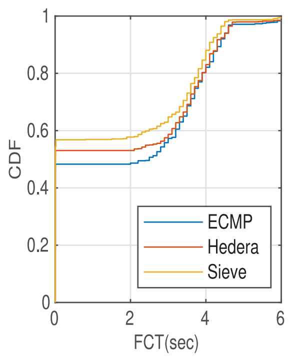
Sieve needs to handle half of $H \times 10^3/F$ flow set up per second, which is 2.5 million requests per second. Using specific hardware, a single controller can handle up to 12 million requests per seconds as in [28]. In this numerical study, we adopt a size of the commercial data centers as presented in [5]. On the other hand, Eq. (10) detects the rate at which Sieve needs to process port statistics messages from switches. Consequently, Sieve will handle $S \times P/P_{rate}$, i.e., 723 packets per second. Assuming that the controller can handle these packets at the same rate of handling flow setup, as in [28], so it unlikely under the mentioned parameters that Sieve's performance will reduce severely.

5.6. Impact of threshold values

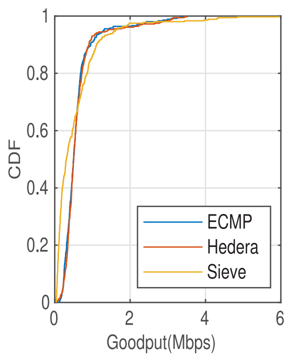
Our framework probes the occupied bandwidth on edge switch ports connected to the aggregate layer to figure out if it is below the predefined threshold. However, due to the overhead and rescheduling failure probability associated with different threshold values, we evaluate the occupied bandwidth threshold values. In particular, we investigate the effects of different threshold values on the number of $OFPFlowStats$ message replies to measure the yielded overhead in the controller, since upon threshold hits, Sieve probes flow statistics by sending $OFPFlowStats$ message to detect elephant flows on a specific switch port. Besides, we measure the associated failure probability of rescheduling elephant flows to other paths in case of each threshold value by computing the number of successfully rescheduled elephant flows out of the total number of rescheduling requests. In particular, the frequency of rescheduling requests is inversely proportional to the threshold value. Table 6 presents the number of messages and the number of failures associated with the different values of the bandwidth occupation percentages under different traffic patterns, which are described in Section 6. We aim to find the optimal value so that the



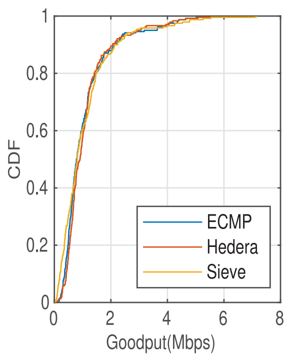
(a) FCT for mice flows under 1:1 pattern



(b) FCT for mice flows under 3:1 pattern



(c) Goodput for Elephant flows under 1:1 pattern



(d) Goodput for Elephant flows under 3:1 pattern

Fig. 7. Performance of Sieve framework under CT scenario.

Table 6
Effect of different values of occupied bandwidth threshold on the probability of finding alternative paths to reschedule elephant flows in case of CT and UT traffic patterns.

Criterion	$OFPFlowStats$_number		Failure_Probability%	
Threshold	CT	UT	CT	UT
75%	0	25	0	8
50%	3	168	0	42.6
25%	296	1265	28.7	70.6
10%	1226	1686	5.9	60

probability of finding alternative paths for rescheduling elephant flows is reasonable. Besides, the threshold value should preserve the controller from an intensive message load as well. Therefore, we decided to adopt 25% as a value for the predefined threshold of bandwidth occupation where it yields 28.7%, 70.6% as failure probabilities in CT and UT scenarios, respectively. In contrast, 10% threshold yields fewer failure probabilities, 5.9% and 60% for CT and UT, respectively, but with a much higher number of messages, which increases the load on the controller. On the other hand, such a small value as 10% of the bandwidth occupation threshold could result in network instability due to the elephant flows rescheduling fluctuations. However, 50% and 75% percentages are not desirable since we aim to improve the FCT of mice flows, and such values can slow down the framework reaction.

5.7. Number of flow table entries

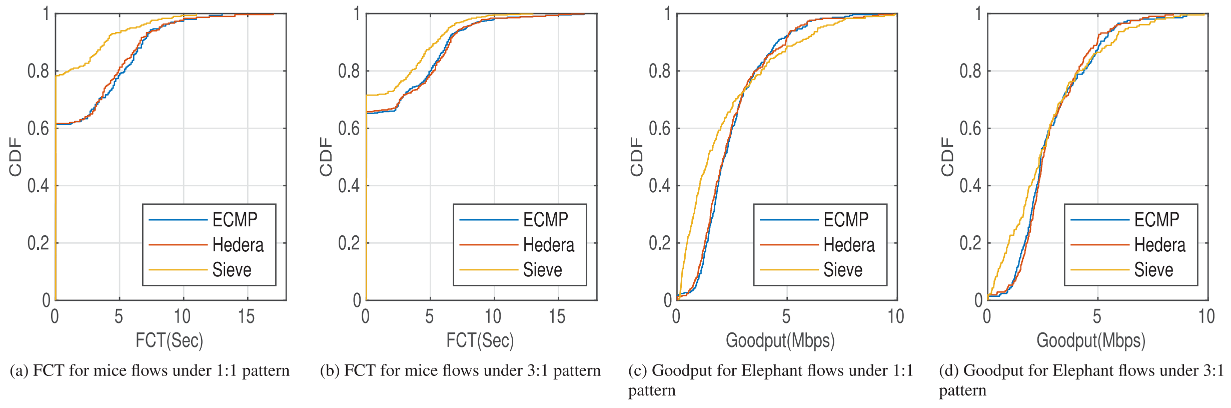
We measure the number of flow entries generated by our framework and compare it to a fully reactive scheme. In particular, fully reactive scheme sends a *packet-in* packet upon receiving the first packet of a new flow to the controller. Subsequently, the controller tries to find a path, and then it installs a new flow entry into switches along the path. In this context, we aim to figure out the difference in flow numbers between fully reactive scheme and our scheme presented in Table 3 which is called proactive. Besides, we investigate if the controller can cap with the received requests and if the number of the flow entries can be absorbed based on the flow table size. Accordingly, we count the flow entries number generated by the second and the third layers in both cases under the uniform traffic pattern UT, which is described in Section 6. The results shown in Fig. 6 indicates that the proactive scheme, i.e., our scheme, has fewer flow entries up to 50% than that in case of the fully reactive one. In addition, since a controller can deal with about 10 millions of flows per second [28,29] and the flow table can contain up to 5k flows [14], we conclude that our framework yields reasonable load and flow entry number. Furthermore, the numbers in Fig. 6 are the cumulative numbers of the generated flow entries during the whole experiment whose length is 300 s. Therefore, this implies that the number of the simultaneous flow entries at one instant is so lower. Moreover, the presented numbers in Fig. 6 in case of L2, coincides with Theorem 2 since the sampling process is held by L2 and the presented number in case of the proactive scheme is less than that in the case of the reactive scheme by more than half.

5.8. Framework implementation

The framework modules in the control plane are implemented as Python modules and integrated with Ryu SDN controller [30]. We leverage OpenFlow 1.3.1, and the testbed environment has been implemented by Mininet 2.2.2d where we evaluate our framework in 4-ary FatTree data center network, as shown in Fig. 1 [27]. We employed Intel Core i5-8400 3.20 GHz, 16 GB RAM, Ubuntu 16.04.

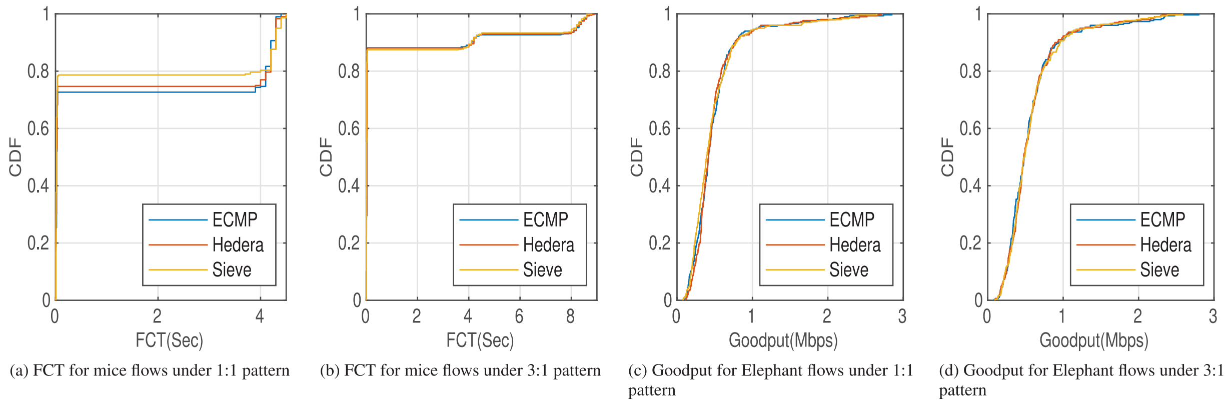
6. Experimental results

We compare our framework's performance to Hedera and ECMP [12] since Hedera is the mainstream scheduling and detection framework for DCN, and ECMP acts as a commonly used scheduler in academic and business sectors. Since Mininet runs in real-time and for the sake of precision, we did not use high values for link capacity. Therefore, each core switch connects to four aggregation switches with 100 Mbps bandwidth and 250 μs one-way propagation delay links, 20 Mbps and 1 ms one-way propagation delay for links connect aggregation and edge layers, and 10 Mbps and 2 ms one-way propagation delay for links connect edge switches and end-hosts where each edge switch connected with two end-hosts. Hence, the oversubscription ratio is 1:2 at the edge layer. We evaluate the framework performance by conducted three different 300 s scenarios containing a mix of mice and elephant flows for each scenario. In the first scenario, *concentrated traffic (CT)*, elephant and mice flows follow many-to-one patterns, in which twelve end-hosts send data to three end-hosts on different pods than the sources. The second one follows the uniform model, *Uniform Traffic (UT)*, where connections span all layers and all end-hosts have been employed to generate the traffic, and each source has a different destination. Finally, *Multi Destinations*, we generate traffic from two end-hosts connected to the same edge switch to ten different end-hosts on the other pods, five destinations for each source. We employ iperf for generating elephant flows and Apache server [31] for generating mice flows by repeatedly requesting a webpage of size 10 KB at the tenth second of the simulation lifespan and elephant flows last randomly between [20,60] seconds. In the case of this group of scenarios, our framework will be examined in a situation where mice flows are synchronized to generate burstiness and elephant flows exist to evaluate the framework under high load. Specifically, every 10 s, the previous pattern repeats to generate the burstiness during the experiment life span. We conduct the experiments for two different traffic classes. First, we employ high elephant flows share of 1:1, i.e., mice:elephant ratio, to investigate the impact of the framework under a high volume of elephant traffic. Second, we simulate a mice:elephant ratio of 3:1 as



(a) FCT for mice flows under 1:1 pattern (b) FCT for mice flows under 3:1 pattern (c) Goodput for Elephant flows under 1:1 pattern (d) Goodput for Elephant flows under 3:1 pattern

Fig. 8. Performance of Sieve framework under UT scenario.



(a) FCT for mice flows under 1:1 pattern (b) FCT for mice flows under 3:1 pattern (c) Goodput for Elephant flows under 1:1 pattern (d) Goodput for Elephant flows under 3:1 pattern

Fig. 9. Performance of Sieve framework under MD scenario.

Table 7
Number of mice to elephant flows in CT, MD and UT scenarios in case of equilibrium ratio which is 1:1 and when mice flows are three times more than elephant flows, i.e. 3:1.

Scenario	1:1	3:1
CT	360:360	871:300
MD	300:300	871:300
UT	300:300	871:300

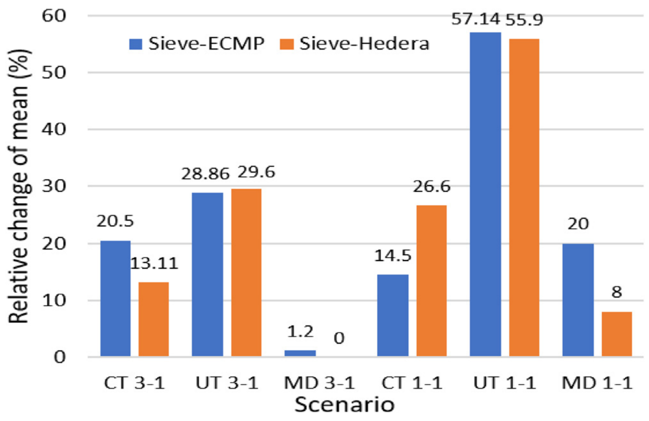


Fig. 10. Relative changes of average mice flows FCT of Sieve in comparison to Hedera and ECMP in the first scenario group.

the ratio reported in [5]. Moreover, Table 7 presents the details of the flow numbers generated in each scenario. We follow the traffic pattern in [17] to compare Sieve performance to Hedera and ECMP in terms of FCT of mice flows and goodput of elephant flows. We repeat each experiment 10 rounds for each different scenario. For mice flows, we present Cumulative Distribution Function (CDF) of FCT. For elephant flows, we essentially present CDF of the goodput. These results are shown in Fig. 7 for the CT scenario, Fig. 8 for the UT scenario and Fig. 9 for the MD scenario. Furthermore, we evaluate our framework under a second scenario group. Specifically, we compare our framework performance under the traffic pattern employed in [12] in terms of average goodput of elephant flows and the average aggregate throughput of all flows in the network. Finally, we conduct a third scenario group in which we employ real workloads to investigate Sieve performance in web services, cache [1] and data mining [6] applications scenarios.

6.1. FCT of Mice flows

We compare FCT of each algorithm, as shown in Figs. 7–9. Since mice flows are delay-sensitive flows, FCT is the most important metric to measure the algorithms performance. As shown in the Fig. 7a-7b, Sieve outperforms Hedera and ECMP in case of many-to-one traffic pattern. Consequently, our framework can mitigate the delay of mice flows by rescheduling elephant flows upon bandwidth occupation hits the threshold and efficiently utilizes the other network links. On the other hand, ECMP provides no consideration of this problem, and Hedera does not invoke the first global fit algorithm based on link situation but based on elephant flows consumption. Similarly, FCT of mice flows provided by Sieve is less compared to Hedera and ECMP in case of UT scenario, as shown in Fig. 8a-8b. As the number of the sources and destinations are the same, the opportunity of finding other paths for elephant flows upon threshold hits is high. Finally, Fig. 9a-9b show the results of MD scenario. The performance of all methods is

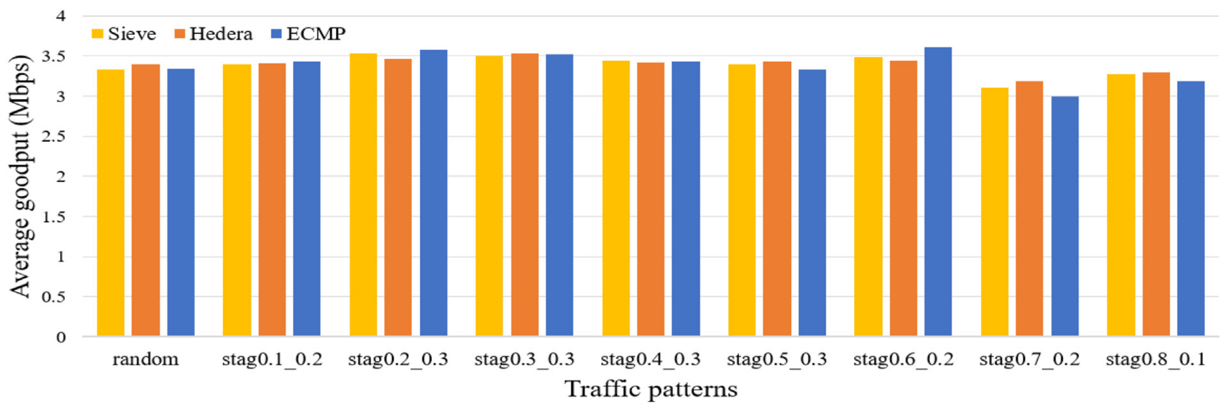


Fig. 11. Average goodput of the elephant flows from H1 to H16 in case of all traffic patterns of the second scenario group.

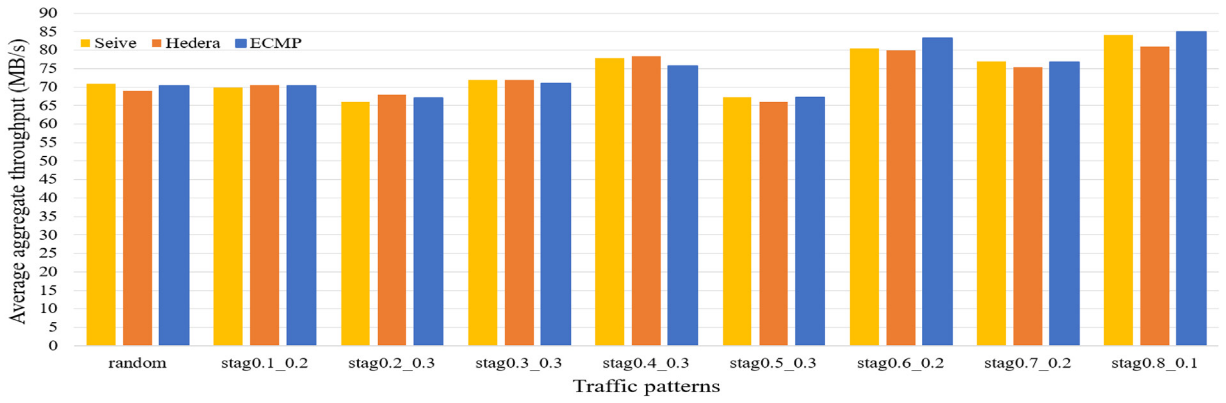


Fig. 12. Average aggregate throughput of all elephant flows in the network in case of all traffic patterns of the second scenario group.

Table 8
Confidence interval of goodput of all algorithms in different scenarios and flow ratios. CDF of algorithms' goodput is shown in Figs. 7d–Fig. 9d Figs. 7c–Fig. 9c.

Scenario	Hedera	ECMP	Sieve
CT 3–1	1.11 ± 0.11	1.09 ± 0.12	1.08 ± 0.12
UT 3–1	2.9 ± 0.2	2.92 ± 0.21	2.73 ± 0.26
MD 3–1	0.57 ± 0.04	0.57 ± 0.04	0.57 ± 0.04
CT 1–1	0.61 ± 0.05	0.59 ± 0.04	0.54 ± 0.07
UT 1–1	2.5 ± 0.18	2.53 ± 0.17	2.2 ± 0.23
MD 1–1	0.49 ± 0.04	0.49 ± 0.04	0.48 ± 0.04

the same because there are only two sources connected to the same edge switch. Therefore, the opportunity of finding other alternative paths for elephant flows is rare. Fig. 10 presents the relative changes of average FCT of the mice flows under all scenarios. As shown, Sieve outperforms Hedera and ECMP in all scenarios, but the greatest positive improvement is under UT 1–1 since the load in the network links is balanced. On the other hand, the lowest positive change value is in MD scenario where all links toward the sources are saturated, especially in the case of MD 3–1. Besides, Sieve provides less FCT for mice flows under many-to-one traffic pattern, which is the common pattern in DCN.

Table 8 presents the confidence intervals of the average goodput of the elephant flows under the various scenarios, where the confidence interval is 95%. Based on the shown numbers, Sieve has similar average goodput to Hedera and ECMP for all scenarios. As a result, Sieve can provide almost equivalent average goodput to Hedera and ECMP for elephant flows.

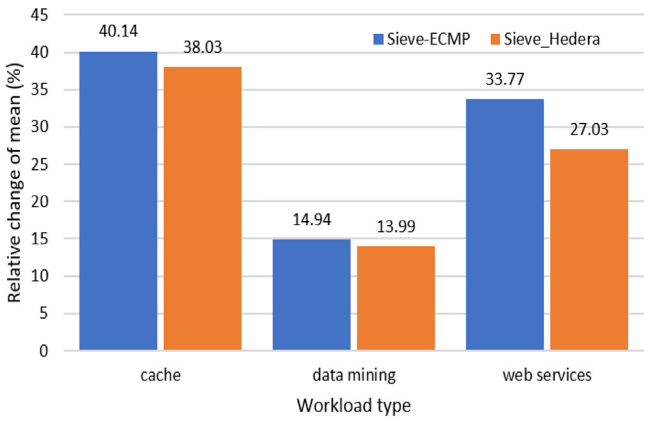


Fig. 13. Relative changes of average mice flows FCT of Sieve in comparison to Hedera and ECMP resulted from employing realistic traffic loads in the third scenario group depicted according to traffic type.

6.2. Throughput of elephant flows

In this section, we compare the goodput of elephant flows of the first scenario group, as shown in Figs. 7–9. Sieve provides elephant flows with a goodput close to that of Hedera and ECMP under almost all cases. Furthermore, we compare the goodput of elephant flows under the second scenario group. The generated traffic patterns consist of a random pattern and staggered probability pattern. In particular, the traffic patterns are detailed as follows.

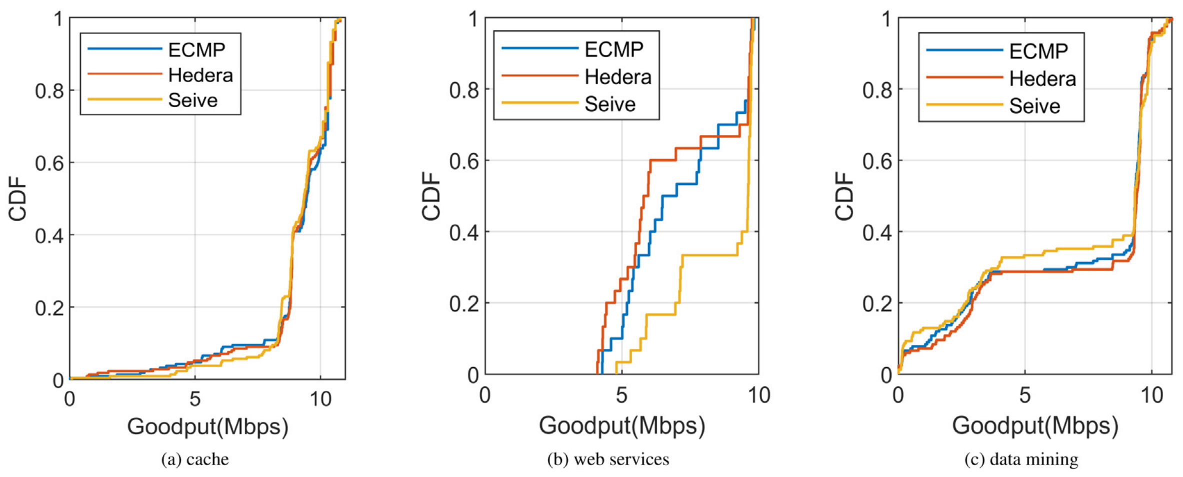


Fig. 14. CDF of the goodput of elephant flows resulted from employing realistic traffic loads in the third scenario group of all algorithms according to traffic type.

1. Random: each end-host sends traffic to any other end-host in the network with equal probability.
2. Staggered probability ($Edge\_p, Pod\_p$): each end-host sends traffic to another one connected to the same edge switch with probability ($Edge\_p$), to the same pod with probability (Pod_p) and to other pods in the network with probability ($1 - Edge\_p - Pod\_p$).

The generated flows differ in sizes and numbers. Each end-host generates one elephant flow lasts for 55 s as well as eight mice flows which have different and random sizes ≤ 50 KB. Furthermore, flows arrive in Poisson distribution with 10 ms inter-arrival time. Moreover, we repeat the experiment 10 rounds for each algorithm in case of each pattern. In addition, we generate three parallel elephant flows along with nine mice flows from H1 to H16 to compute the average goodput, where elephant flows lasts for 55 s.

Similar to the results we have in the first scenario group, Sieve provides pretty close goodput in comparison to ECMP and Hedera for the flows initiated from H1 to H16, as shown in Fig. 11. In this context, the works in [32–34] prove that Hedera and ECMP have so close throughput as well. On the other hand, Fig. 12 depicts the average aggregated throughput of all elephant flows in the network. Basically, the throughput achieved by Sieve confirms the approximate equivalence.

6.3. Real workload

In this scenario group, we employ real workloads from production datacenter networks. The flow distribution of web services is less than 10 KB for 90% of the flows, and 90% of cache flows are less than 500 KB [1]. On the other hand, 90% of data mining flow size distribution is less 100 KB [6]. Therefore, we generate flows based on the mentioned flow size distributions in the topology shown in Fig. 1 by employing uniform traffic pattern (UT) so that the load on all network segments are equal and the generated flow number of the three workloads as well. Specifically, we generate 1920 flows with inter arrival time of 10 ms as in [1], so 120 flows from each host. For this scenario group, we compare FCT of mice flows and goodput of elephant flows under Sieve with ECMP and Hedera results.

Fig. 13 depicts the relative change of mice flow FCT according to the workload types. As shown, Sieve outperforms Hedera and ECMP consistent with the results presented in Fig. 10. Similar to the performance in the first and second scenario groups, the goodput of elephant flows under Sieve framework is equivalent to that in case of ECMP and Hedera, as shown in Fig. 14. However, elephant flows in case of web services are higher than the other algorithms due to the fact the majority of the web flows are scheduled as mice flows, so the rescheduling of the other workload type flows yields more bandwidth for web flows.

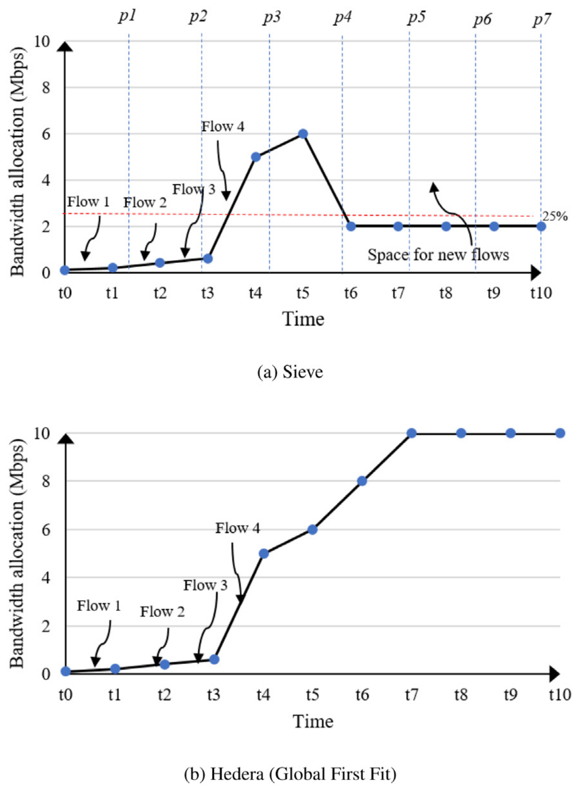


Fig. 15. Illustration of the difference between flow rescheduling on link X in case of Sieve employing link bandwidth occupation as a threshold, Fig. 15a, and employing flow size as threshold adopted by Hedera depicted in Fig. 15b.

7. Discussion

So far, we presented our flow scheduling solution. In this section, we justify our results by analyzing the behavior of Sieve in comparison to Hedera and ECMP. For this sake, let us assume two links X and Y receive several mixed flows whose arrival is Poisson process with λ parameter. For illustration, the link X will be demonstrated with four flows, e.g., three mice flows and one elephant flow. The first three flows arrived and served in the first three time intervals, i.e., $t1, t2, t3$, as shown in Fig. 15. Then, the elephant flow, i.e., Flow 4, begins with slow start phase within the third time interval, $t3$, as depicted in Fig. 15a-15b.

In case of Sieve, we define polling rates $p1, p2, \ldots$ to monitor the load on the edge switches. Once the occupation rate on an edge switch port of link X reaches 25% of the total capacity, the controller starts to detect elephant flows on it. Therefore, during the polling rates $p1$ and $p2$, the controller only saves the total load values on X into the network graph. However, during $p3$ the occupation reaches the threshold, so *elephant flow detection* module discovers the elephant flow, and it tries to reschedule it to another link with better bandwidth, i.e., Y, during $p4$ as shown in Fig. 15a. Consequently, some bandwidth will be available for the current and upcoming mice or elephant flows. Worth to mention, elephant flows rescheduling is not occurring frequently since the alternative links may not absorb them.

On the other hand, Hedera monitors all flows at edge switches to detect the elephant flows. However, the elephant flows detection is based on exceeding 10% of link capacity. Therefore, in case of 1 Gbps link, an elephant flow must occupy more than 100 Mbps. Hence, in the co-existing of many elephant flows whose consumptions are below 10% of the link capacity for each, the link will be saturated before elephant flows are appropriately detected and rescheduled. Thus, mice flows will suffer from latency, and their FCT will increase as a result. As depicted in Fig. 15b, every mice flow scheduled by ECMP after $t7$ will be exposed to significant delay or even be lost since the elephant flows have already overtaken the link capacity. Furthermore, our framework invokes the elephant flow rescheduling only when a link occupation approaches 25% of its capacity. Besides, we sample a portion of the flows. As a result, the monitoring and detection burden is reasonable. In contrast, Hedera monitors consumption's of each flow to invoke the rescheduling procedure.

8. Conclusion

The emerging cloud-based technologies leverage DCN. Thus, it is essential to guarantee a suitable QoS to meet the requirements of delay-sensitive applications. In this paper, we tackle the problem of improving FCT of mice flows in DCN, as well as maintain the throughput of elephant flows. We present Sieve which schedules a portion of flows and reschedules the detected elephant flows. In addition, our framework is a distributed solution that balances the scheduling burden between the data plane and the control plane. Furthermore, we propose a sampling mechanism by which Sieve can sample a portion of the network flows to mitigate the sampling overhead and ECMP-related packet collisions. Besides, we investigate its impact on flow table size, the controller overhead and the optimal values of the polling rate and the threshold. We compare FCT of mice flows in case of Sieve to that in case of Hedera and ECMP. We show that Sieve improves FCT up to 58% without impairing the throughput of elephant flows by conducting three scenario groups. Moreover, Sieve does not require any change neither in the network hardware nor in end-hosts.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

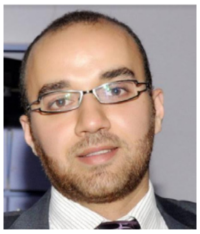
Acknowledgment

Project no. 135074 has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary under the FK_20 funding scheme.

References

[1] A. Roy, H. Zeng, J. Bagga, G. Porter, A.C. Snoeren, Inside the social network's (datacenter) network, in: Proceedings of the ACM Conference on Special Interest Group on Data Communication, 2015, pp. 123–137.
[2] A. Sivanathan, H.H. Gharakheili, F. Loi, A. Radford, C. Wijenayake, A. Vishwanath, V. Sivaraman, Classifying IoT devices in smart environments using network traffic characteristics, IEEE Trans. Mob. Comput. 18 (8) (2018) 1745–1759.
[3] J.A. Rashid, Sorted-GFF: An efficient large flows placing mechanism in software defined network datacenter, Karbala Int. J. Mod. Sci. 4 (3) (2018) 313–331.
[4] M. Alizadeh, A. Greenberg, D. Maltz, J. Padhye, P. Patel, B. Prabhakar, M. Sridharan, DCTCP: Efficient packet transport for the commoditized data center, 2010.
[5] 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.
[6] A. Greenberg, J.R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D.A. Maltz, P. Patel, S. Sengupta, VL2: a scalable and flexible data center network, ACM SIGCOMM Comput. Commun. Rev. 39 (4) (2009) 51–62.
[7] M. Noormohammadpour, C.S. Raghavendra, Datacenter traffic control: Understanding techniques and tradeoffs, IEEE Commun. Surv. Tutor. 20 (2) (2017) 1492–1525.
[8] J. Brutlag, Speed matters for Google web search, 2009, Google.
[9] N. Dukkipati, N. McKeown, Why flow-completion time is the right metric for congestion control, ACM SIGCOMM Comput. Commun. Rev. 36 (1) (2006) 59–62.
[10] A.R. Curtis, W. Kim, P. Yalagandula, Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection, Infocom 11 (2011) 1629–1637.
[11] C.A. Wang, B. Hu, S. Chen, D. Li, B. Liu, A switch migration-based decision-making scheme for balancing load in SDN, IEEE Access 5 (2017) 4537–4544.
[12] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, A. Vahdat, Hedera: dynamic flow scheduling for data center networks, 10, (8) 2010, pp. 89–92.
[13] F. Tang, H. Zhang, L.T. Yang, L. Chen, Elephant flow detection and differentiated scheduling with efficient sampling and classification, IEEE Trans. Cloud Comput. (2019).
[14] A.R. Curtis, J.C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, S. Banerjee, Devoflow: Scaling flow management for high-performance networks, ACM SIGCOMM Comput. Commun. Rev. 41 (4) (2011) 254–265.
[15] R. Trestian, G.M. Muntean, K. Katrinis, MiceTrap: Scalable traffic engineering of datacenter mice flows using OpenFlow, in: 2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013), 2013, pp. 904–907.
[16] A. Yazidi, H. Abdi, B. Feng, Data center traffic scheduling with hot-cold link detection capabilities, in: Proceedings of the 2018 Conference on Research in Adaptive and Convergent Systems, 2018, pp. 268–275.
[17] Y.C. Wang, S.Y. You, An efficient route management framework for load balance and overhead reduction in SDN-based data center networks, IEEE Trans. Netw. Serv. Manag. 15 (4) (2018) 1422–1434.
[18] S. Kandula, D. Katabi, S. Sinha, A. Berger, Dynamic load balancing without packet reordering, ACM SIGCOMM Comput. Commun. Rev. 37 (2) (2007) 51–62.
[19] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. Lam, F. Matus, R. Pan, N. Yadav, G. Varghese, CONGA: Distributed congestion-aware load balancing for datacenters, in: Proceedings of the 2014 ACM Conference on SIGCOMM, 2014, pp. 503–514.
[20] S. Wang, J. Zhang, T. Huang, T. Pan, J. Liu, Y. Liu, Flow distribution-aware load balancing for the datacenter, Comput. Commun. 106 (2017) 136–146.
[21] L. Liu, Y. Jiang, G. Shen, Q. Li, D. Lin, L. Li, Y. Wang, An SDN-based hybrid strategy for load balancing in data center networks, in: 2019 IEEE Symposium on Computers and Communications, 2019, pp. 1–6.
[22] M. Zaher, S. Molnar, Enhancing of micro flow transfer in SDN-based data center networks, in: Proceeding of ICC 2019-2019 IEEE International Conference on Communications, 2019, pp. 1–6.
[23] Y. Afek, A. Bremler-Barr, S.L. Feibish, L. Schiff, Detecting heavy flows in the SDN match and action model, Comput. Netw. 136 (2018) 1–12.
[24] J. Zheng, Q. Ma, C. Tian, B. Li, H. Dai, H. Xu, Q. Ni, Hermes: Utility-aware network update in software-defined wans, in: Proceedings of 2018 IEEE 26th International Conference on Network Protocols (ICNP), 2018, pp. 231–240.
[25] J. Hu, J. Huang, W. Lv, Y. Zhou, J. Wang, T. He, CAPS: Coding-based adaptive packet spraying to reduce flow completion time in data center, IEEE/ACM Trans. Netw. 27 (6) (2019) 2338–2353.
[26] P. Wang, G. Trimponias, H. Xu, Y. Geng, Luopan: Sampling-based load balancing in data center networks, IEEE Trans. Parallel Distrib. Syst. 30 (1) (2018) 133–145.
[27] M. Al-Fares, A. Loukissas, A. Vahdat, A scalable commodity data center network architecture, ACM SIGCOMM Comput. Commun. Rev. 38 (4) (2008) 63–74.
[28] D. Erickson, The beacon OpenFlow controller, in: Proceeding of 2nd ACM SIGCOMM Workshop Hot Topics Softw. Defined Netw., 2013, pp. 13–18.
[29] Y. Zhao, L. Iannone, M. Riguidel, On the performance of SDN controllers: A reality check, in: Proceeding IEEE Conf. Netw. Funct. Virtualization Softw. Defined Netw., 2015, pp. 79–85.

[30] Ryu: Ryu SDN Framework. http://osrg.github.io/ryu/ (Accessed 12 Mar. 2019).
[31] Apache.org: Apache HTTP server benchmarking tool. http://httpd.apache.org/docs/current/install.html.
[32] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, N. McKeown, Reproducible network experiments using container-based emulation, in: Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies, 2012, pp. 253–264.
[33] A. Alawadi, M. Zaher, S. Molnár, Methods for predicting behavior of elephant flows in data center networks, Infocommunications J. XI (3) (2019) 34–41.
[34] H. Zhang, F. Tang, L. Barolli, Efficient flow detection and scheduling for SDN-based big data centers, J. Ambient Intell. Humaniz. Comput. 10 (5) (2019) 1915–1926.



Maiass Zaher received his M.Sc. in Computer Science from Faculty of Information Technology Engineering, Damascus University, Damascus, Syria, in 2016. Since 2016, he is a Ph.D. student in the Department of Telecommunication and Media Informatics, Budapest University of Technology and Economics, Budapest, Hungary.

Aymen Alawadi received his M.Sc. in Computer Science from Universiti Sains Malaysia in Penang - Malaysia, in 2012. Since 2017, he is Ph.D. student in the Department of Telecommunication and Media Informatics, Budapest University of Technology and Economics, Budapest, Hungary.

Sándor Molnár received his M.Sc., Ph.D. and Habilitation in Electrical Engineering and Computer Science from the Budapest University of Technology and Economics (BME), Budapest, Hungary, in 1991, 1996 and 2013, respectively. In 1995 he joined the Department of Telecommunications and Media Informatics, BME. He is now an Associate Professor and the principal investigator of the tele traffic research program of the High-Speed Networks Laboratory.