Annals of Telecommunications
https://doi.org/10.1007/s12243-021-00898-0

Oddlab: fault-tolerant aware load-balancing framework for data center networks

Aymen Hasan Alawadi[1,2] · **Sándor Molnár**[1]

Received: 5 May 2021 / Accepted: 3 November 2021

Abstract
Data center networks (DCNs) act as critical infrastructures for emerging technologies. In general, a DCN involves a multi-rooted tree with various shortest paths of equal length from end to end. The DCN fabric must be maintained and monitored to guarantee high availability and better QoS. Traditional traffic engineering (TE) methods frequently reroute large flows based on the shortest and least-congested paths to maintain high service availability. This procedure results in a weak link utilization with frequent packet reordering. Moreover, DCN link failures are typical problems. State-of-the-art approaches address such challenges by modifying the network components (switches or hosts) to discover and avoid broken connections. This study proposes Oddlab (Odds labels), a novel deployable TE method to guarantee the QoS of multi-rooted data center (DC) traffic in symmetric and asymmetric modes. Oddlab creatively builds a heuristic model for efficient flow scheduling and faulty link detection by exclusively using the gathered statistics from the DCN data plane, such as residual bandwidth and the number of installed elephant flows. Besides, the proposed method is implemented in an SDN-based DCN without altering the network components. Our findings indicate that Oddlab can minimize the flow completion time, maximize bisection bandwidth, improve network utilization, and recognize faulty links with sufficient accuracy to improve DC productivity.

Keywords Load balancing · Software-defined networking (SDN) · Data centers · Fault-tolerant · Elephant flows

1 Introduction

Data center networks (DCNs) are employed in various fields, including web services, scientific computing, and MapReduce operations. These services typically demand high available bandwidth, fast response, and high availability. Hence, the fat-tree DC topology (see Section 2.1), for instance, is designed symmetrically to achieve a high bisection bandwidth because of multi-rooted paths between the end-hosts. Many traffic engineering (TE) methods have been employed to efficiently leverage multi-rooted paths. However, conventional TE methods, such as equal-cost multi-path (ECMP) [1], are based on local DCN switches to handle flow scheduling. Local optimization mainly undergoes local flow collisions that lead to severe congestion, weak network utilization, and significant flow latency [2]. Thus, implementing an effective DCN flow scheduling technique based on the current network state is needed to avoid potential flow collisions. Moreover, link failure is a crucial issue in DCN fabrics [3]. Consequently, maintaining a symmetric situation for the duration of network operations is difficult. In particular, DCN link failures include partial and complete failures [3]. The DCN topology becomes asymmetric when a failure occurs. Nevertheless, a scheme, such as ECMP, was deployed to hash every flow to a different path to handle traffic congestion in standard DC operations when considering a complete failure without monitoring network states.

✉ Aymen Hasan Alawadi
aymen.alawadi@edu.bme.hu

Sándor Molnár
molnar@tmit.bme.hu

[1] Department of Telecommunication and Media Informatics, Budapest University of Technology and Economics, Budapest, Hungary

[2] Department of Computer Science, Faculty of Education, University of Kufa, Najaf, Iraq

Published online: 24 December 2021

Several solutions and opportunities related to DCN TE solutions have emerged after introducing the Software-Defined Networking (SDN) as a new network administrative paradigm. The new paradigm has introduced a new orientation in managing DCN operations through a centralized controller. For reliable and effective resource handling, SDN decouples the control and data planes. The OpenFlow protocol [4] is a primary mechanism for separating traffic and communication between a centralized SDN controller and distributed switches. The SDN controller can collect statistical information from the entire network, including information regarding the current flows and port states in the data plane because it is centrally located in the control plane.

In the existing DCNs, SDN has been often applied in flow scheduling and traffic load balancing [5, 6]. However, several issues must be addressed before adopting SDN solutions in a DCN TE method. For instance, central SDN controllers can manage only a specified $packet\_in$ requests per second [7]. Moreover, TCAM (Ternary Content-Addressable Memory) space restrictions of OpenFlow switches affect the flow entries that the SDN controller can handle per second. For instance, HP 5130 EI switches can only handle 20 rules per second [8]. Regarding decision-making, frequent flow rerouting may degrade TCP performance through packet reordering.

Benson et al. [9] analyzed simple network management protocol logs for several DCNs to classify DCN flows, and found that most of the DC traffic (80%) comprised small TCP flows within a size of 10 kB for a duration of less than 11 s. In contrast, most DC bytes originate from only 10% of TCP long-lived flows (elephant flows).

DCN flow demand cannot be previously predicted to obtain an appropriate route without cooperating with other devices, such as the sFlow [10] or the end-hosts [11]. Other TE solutions tend to reroute the active elephant flows detected within the DCN, as discussed in Section 2.3). However, Roy et al. [12] discovered that the TCP elephant flows in Facebook DCN are not extremely large over long periods; therefore, the elephant flow rescheduling may not be an efficient solution.

In this study, we propose Oddlab, a heuristic and dynamic TE approach to balance the traffic load of a DCN based on a centralized SDN controller. The basis of our approach relies on the flow sampling (1:1 ratio) at the DCN edge switches between proactive paths defined by the ECMP and reactive paths obtained by the SDN controller. Nevertheless, it significantly differs from our previously published results in [13] and [14] in two main aspects. First, in the reactive method, the number of elephant flow entries [15] is utilized in addition to the path residual bandwidth to define the best path. Therefore, we achieved fewer installed flow entries by abandoning the frequent elephant flow rerouting procedure. Alternatively, we adopted the DCN edge sampling method in addition to forwarding the incoming flow over the less overloaded path, even when it was not the shortest one. Second, Oddlab detects and avoids the faulty links inside a DCN and reroutes the elephant flows from the affected paths to deliver high availability. Our proposed faulty link detection procedure method depends on the state of the DCN load temporally correlated with the highly congested links at the core switch layer, which needs to be the most reliable part of the DCN fabric.

Oddlab includes several stages. The controller begins to learn the topology and indicates the shortest paths of the edge host of the DCN that directly connects to the end-hosts. Subsequently, the controller periodically monitors the ports of the DCN switches to determine the residual bandwidth and number of installed flow entries that belong to the existing elephant flows. This stage includes estimating the DCN utilization state based on the number of edge switches that exceed a certain threshold and correlating it with the congested links at the core switches to detect the potential faulty links. Consequently, elephant flows on the affected link are rerouted to the least-congested paths. In general, Oddlab introduces a deployable, light, yet effective load-balancing framework that functions on the basis of SDN in symmetric and asymmetric DCN topologies without altering network components or TCP packets.

In summary, the key contributions of Oddlab are as follows:

1. We introduce a new SDN load-balancing framework to guarantee the QoS of the multi-rooted DCN traffic in symmetric and asymmetric DCN topologies, where flows are forwarded equally based on the controller and edge switches using ECMP. The proposed adaptive scheduling considers the detected healthy paths, available bandwidth, and active elephant flows to determine the best paths.
2. We present an adaptive flow scheduling based on the global view of DCN component information with a minimum flow scheduling overhead and without the need for frequent flow rerouting and altering of network components.
3. We formulate the faulty link detection problem in DCN as a temporal correlation between the number of loaded edge switches based on the $Odds$ ratio, and the throughput of the core switch links to gain sufficient accuracy in identifying the faulty links at the core switches.
4. Finally, we evaluate the performance of the proposed method by conducting extensive experiments on various traffic patterns with synthetic and real workloads. We found that the proposed method can identify faulty links and noticeably improves the bisection bandwidth and

moderate link utilization with an overall average flow completion time (FCT) reduction by up to 30, 25.7, 62, and 5% as compared to that of ECMP, Hedera, PureSDN, and Sieve, respectively.

The remainder of this paper is organized as follows. In Section 2, we present a preliminary background, including an overview of the existing studies. In Section 3, we introduce the design of the Oddlab framework. Section 4 presents the main algorithms of proactive sampling on DCN edges, proactive flow scheduling (ECMP), adaptive flow scheduling, faulty link detection procedure, and elephant flow rescheduling. In Sections 5, 6, and 7, we analyze, implement, and evaluate the Oddlab experimental results, respectively. Finally, Section 8 provides the concluding remarks.

2 Preliminary background

This section describes the DCN topology in addition to an overview of DCN traffic congestion and failures, followed by an overview of the existing studies.

2.1 DCN topology

In general, the fat-tree DCN topology [16] is widely used because it delivers high scalability through the tree structure of the switches. The fat-tree contains three layers of connected switches, including the core, aggregate, and edge switches. The DCN end-hosts are directly connected to the edge switch layer. The edge switches are combined with the aggregate switches in pods, where each fat-tree DC contains K pods. Each pod should be connected to $(K/2)^2$ end-hosts, and each switch has $(K/2)$ ports. The DC pods are aggregate switches connected to $(K/2)$ core switches on the upstream side and $(K/2)$ edge switches on the downstream side. The total number of end-hosts that the fat-tree DCN can support is $(K^3/4)$. For every pair of these hosts, the fat-tree provides $(K/2)^2$ equal-cost paths to achieve a high bisection bandwidth between each source and destination inside the DCN environment.

In the SDN paradigm, the central controller monitors the ports of each switch in the DCN data plane to effectively handle traffic flows and make decisions based on the gathered information. Hence, all DCN switches must be remotely connected to the SDN controller, as shown in Fig. 1. This study uses a $K-4$ fat-tree DCN topology with 16 end-hosts.

2.2 Adaptive flow scheduling in DCN

Adaptive flow scheduling primarily improves the DC productivity by avoiding congested paths. For instance, in Hedera [2], the initial flows are scheduled based on predefined flow entries that follow proactive paths using the ECMP hashing method and reroute the elephant flows only with adaptive rerouting. In our previous studies (Sieve [13] and [14]), we leveraged ECMP hashing together with the adaptive routing calculated by the SDN controller periodically based on the shortest path that guarantees the best available bandwidth. Furthermore, Sieve [13] attempted to reroute the fraction of elephants on an edge switch when the switch reaches a specific load threshold to a different path based on the residual bandwidth to achieve a better FCT. Still, such a mechanism may produce further congestion and flow contention, particularly for newly arrived flows.

As shown in Fig. 2, in T1, an elephant flow A originated from host H1 in pod 1 to host H11 in pod 3. Because the links in the core switches are highly congested layers in the DCN, local congestion in T2 may occur between aggregation switches (Agg. 2) and (switch 3) in the core layer. In addition, a severe bottleneck at the core layer may lead to a link failure [3]. In Hedera [2] and Sieve

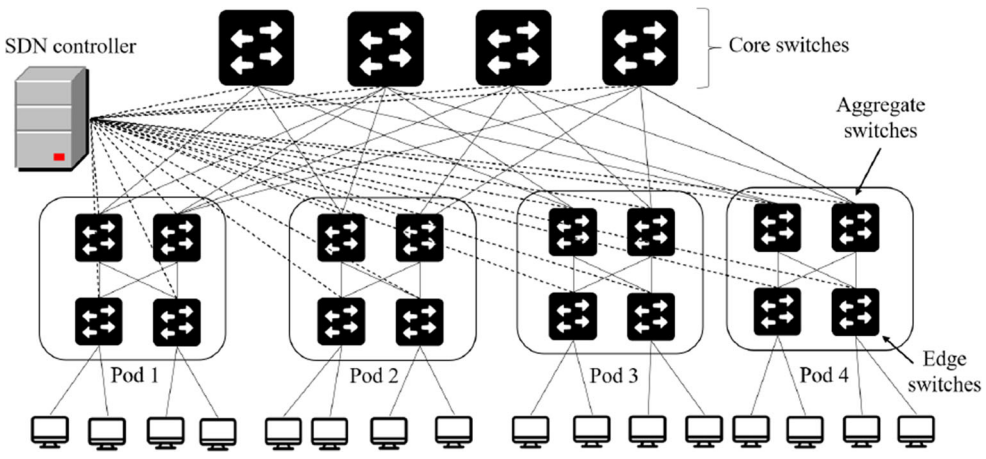

Fig. 1 $K-4$ fat-tree DC topology with a central SDN controller

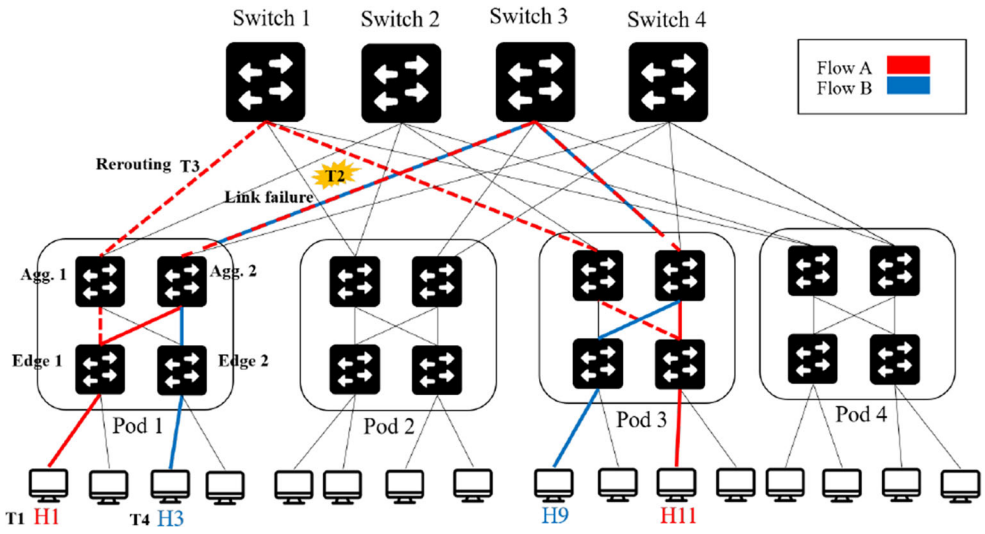

Fig. 2 Flow collisions and rerouting in fat-tree DC topology

[13], elephant flows in Edge1-Agg.2 links are rerouted to another path in T3. This decision may derive further congestion and delay the current active flow, particularly to the growing elephant flows. In T4, elephant flow B originated from H4 in pod 1 to H9 in pod 3. When all paths to the destination host H9 are congested, the path containing the link (Agg.2–switch3) may be considered. In this case, TCP congestion control reduces the number of packets passing through the congested link so that the edge switch may not bypass the rerouting threshold to reroute the congested elephant flow. Therefore, the current active elephant flows inside the DCN must be considered in the incoming flow scheduling decision. This example clearly explains that frequent flow rerouting may not achieve a high productivity, particularly for unsplittable flows [17]. Nevertheless, the rerouting decision must avoid possible link failures in adaptive flow scheduling.

2.3 Related studies

Several studies have proposed practical methods to solve the flow scheduling problem. Hopps [1] proposed ECMP, a flow hashing method based on distributing the flows among the available paths without considering the path status. The ECMP-supported switches are configured with proactive paths to route the arrived packets based on their header values. Methods such as Hedera [2] and Mahout [5] leave the baseline schedule for the mice to flow to the ECMP routing method. Hedera [2] assumed that every flow is a mouse flow until it reaches more than 10% of the total capacity of the edge link capacity to classify it as an elephant flow. Subsequently, Hedera reroutes the elephant flow to the best available path based on the global first fit and simulated algorithms. The hypothesis of handling the mice flows based on ECMP affects the FCT of such flows because the elephant flows would occupy the total link capacity either way.

Sharma et al. introduced DevoFlow in [18] based on the OpenFlow switch to schedule mouse flow using the wildcard rules to select the output port according to a probability distribution. The DevoFlow controller reacts to the elephant flows only after detecting them on the edge switches within a threshold (1–10 MB) to redirect them to the least-congested paths using the bin-packing algorithm. The performance of mice flow FCT has not been presented in the study to evaluate the effect of the wildcard rules on this criterion. BLEND was presented in [19], which relies on the DCN end-hosts and controller for flow scheduling. The end-hosts periodically monitor all the outgoing flows to verify elephant flows and select the paths for the remaining flows based on the lowest estimated round-trip time (RTT). Moreover, the controller maintains the global queuing calculation and elephant flows routing. However, the deployment in a productive DCN is complicated. Modifying the transport layer of the end-hosts is not a viable solution. In addition, many channels are required between the SDN controller and end-hosts to transfer the flow information. Some low-latency applications may bypass the host kernel to perform the transport process [20]. Levi et al. proposed an elephant flow detection and rerouting method in [21]. The method relies on exploiting the path diversity of the DCN topology so that the elephant flows are rescheduled on less congested shortest and non-shortest paths. The method is mainly based on Hedera's strategy to schedule the initial flows using the proactive paths (ECMP). Subsequently, elephant flows reaching 10% of the link capacity are rerouted on a congestion threshold. Then, the best path is selected based on the path congestion rank and

the estimated path delays. Although the proposed method enhanced elephant flow latency, it did not investigate the average overall FCT.

DCN undergoes various uncertainties in its operation, such as traffic dynamics, failures, and asymmetric topology [22]. Many approaches have been introduced to address these challenges. In [20], Alizadeh et al. presented CONGA, in which the concept of flowlet (packet-level granularity) was introduced to achieve optimal flow distribution in an asymmetric topology by collecting the congestion feedback from the switches. Flowlet is a small chunk of a flow sent dynamically across the switch ports depending on the congestion feedback. This process affects the FCT of mice flows and packet reordering in flowlet rerouting on congestion or failure detection. However, the switch hardware needs to be altered to provide a congestion feedback. Hermes is a congestion-aware load-balancing technique that was proposed in [22]. Hermes functions on packet routing and flow scheduling on congestion or failure. In congestion detection, the method depends on the explicit congestion notification (ECN) and RTT. Although Hermes is deployable because hardware modification is not needed, all end-hosts in the DCN need to participate in the sensing process. Therefore, the sensing technique is challenging to accomplish without an end-host aid. CAPS was presented in [23], which is an end-host-based local congestion-aware technique. The method includes three main modules: the packet encoder and decoder on each DC host, in addition to random packet spraying (RPS). In CAPS, traffic flows are divided into mice and elephant flows, where elephants are scheduled using ECMP and mice flows are scattered to all available paths based on RPS. This method requires changes in the software of end-hosts in addition to the availability of RPS switches. End-hosts have also been leveraged in SAPS [24] to handle flow scheduling in an asymmetric topology by providing virtual symmetric paths to each flow using an SDN controller and group tables. In SAPS, the elephant flows are identified based on the number of bytes sent inside each end-host. Hence, the shim layers of these hosts should be visible. The deployment of such a method is costly and may be restricted to cloud environments. DRILL was proposed in [25] as a per-packet scheduling technique that relies on random spread of packets to the shortest- and least-congested queues based on the power of the two-choice approach. The technique was evaluated under different loads on the leaf and spine switches in the Clos DCN topology and required hardware changes at the level of the DCN switches. Recently, FlowFurl was introduced in [26] as a flow-level routing approach that works in an asymmetric DC topology. The method reroutes the flows based on the ECN-bit state transferred between the source and destination over the intermediate switches to detect and discover the healthy paths. FlowFurl requires modifications to the end-hosts because it is in charge of flow rerouting, in addition to the DC intermediate switches.

Table 1 Comparison of DCN load-balancing methods

Method	DCN changes	Gran.	Failure handler
ECMP [1]	×	Flow	Full failure
Hedera [2]	×	Flow	PortLand protocol
Sieve [13]	×	Flow	×
CONGA [20]	Switches	Flowlet	✓
DRILL [25]	Switches	Packet	×
SAPS [24]	Hosts	Packet	✓
Hermes [22]	×	Packet/Flow	✓
FlowFurl [26]	Hosts & switches	Flow	✓
Oddlab	×	Flow	✓

Table 1 presents a comparison between the most relevant DCN load-balancing methods regarding DCN hardware or software components (switches/end-hosts), load-balancing granularity, and failure resilience.

However, this study investigates the possibility of introducing a deployable, light, yet effective load balancing technique that can be used in symmetric and asymmetric DCN topologies utilizing the SDN concept without altering the network components or TCP packets. We proved the performance of our proposed solution in a fat-tree symmetric topology in [13] and [14]. In this study, we demonstrate the performance of the proposed solution (Oddlab) for both symmetric and asymmetric topologies.

3 Framework design aspects

This section describes the main design aspects of Oddlab, starting with problem formulation, followed by a model description.

3.1 Problem formulation

The DCN is modeled as a directed graph $G = (V, E)$, where V is the set of nodes $V = \{v_0, v_1, ..., v_n\}$ and E is a set of directed edges $E = \{e_0, e_1, ..., e_n\}$. Network traffic is routed through flows between every source $s \in V$ and target $t \in V$ with path $P = (v_0, v_1, ..., vn)$, $\forall v \in V$. A directed graph defines the flow network as $G = (V, E, c)$, where each edge $(u, v) \in E$ has a capacity $c(u, v)$. The flow network can be classified into the single- and multi-commodity types. There are only single sources and targets (s, t) in a single commodity flow K, where

$s, t \in V$, and $s \neq t$. Moreover, multi-commodity flows contain K_i commodities originating from a set of (s_i, t_i), where $(s_i, t_i) \in V$. In general, the routing problem in flow assignment should satisfy four constraints: (i) the amount of all flows $f_i(u, v)$ routed on a link should not exceed its capacity $c(u, v)$; (ii) the number of flows entering a node v equals the flows that exit from the same node; (iii) A flow must leave its source node completely; and, (iv) a flow must enter its target node completely. The load-balancing dilemma is based on scheduling the flows with their demands d_i among all links' capacities $c(u, v)$ distributed evenly, as shown in the link utilization $U(u, v)$ (Eq. 1), where $(u, v) \in E$.

$$U(u, v) = \frac{\sum_{i=1}^{k} f_i(u, v).d_i}{c(u, v)}. \tag{1}$$

An obvious solution to this problem is to minimize the maximum utilization of the links U_{max}. In wired networks, the physical link capacity is fixed; however, inevitable failures affect it remarkably. Moreover, the flow demand d_i is not consistent, and its monitoring to deliver a more effective utilization is expensive. A cost of $t(u, v).f(u, v)$ exists when each flow is scheduled and monitored on (u, v). Therefore, we intend to minimize the flow scheduling cost f_cost (Eq. 2) by employing the ECMP flow hashing method and SDN controller based on the power of the two-choice concept [27], as explained in Section 4.1.

$$f\_cost = Min \sum_{(u,v)\in E} (t(u, v) \sum_{i=1}^{k} f_i(u, v)). \tag{2}$$

Owing to the cost of scheduling flows $f_i(u, v)$ among the edges (u, v) in DCN to a particular path $p = (v_1, v_2, ..., v_n)$ among n paths, where $\forall v \in V$, the process of choosing the path should be efficient so that rerouting is not frequently needed to handle the congestion. Therefore, Oddlab finds the least-congested path so that the maximum total cumulative throughput $total\_thr$ is achieved (Eq. 3).

$$total\_thr = Max \sum_{i=1}^{k} d_i. \tag{3}$$

The process of path p choices by the SDN controller for each pair of nodes (s_i, d_i) satisfies the following constraints: (i) the throughput l_e/C_e of every link e between the aggregate and the core switch layer should not be less than (potential link failure ratio) h_th (Eq. 4), where l_e is the current load, and C_e is the physical link capacity, (ii) minimum number of active elephant flows P_{e_l} already active on any link e (Eq. 5), and, (iii) maximum path p residual bandwidth from source s_i to target t_i (Eq. 6).

$$\frac{l_e}{C_e} \leqslant h\_th, \tag{4}$$

$$P_{e_l} = Min \sum_{e=1}^{n} El_e, \tag{5}$$

$$P_b = Max \sum_{e=1}^{n} (C_e - l_e). \tag{6}$$

The first constraint is to detect the potential link failure at the core links to avoid the path with this link in the adaptive flow scheduling model. This restriction is based on the DCN utilization status, determined by the edge switch load $Odds$ ratio. Second, instead of estimating each incoming flow's load (d_i), Oddlab filters the path with the least number of elephant flows. Therefore, elephant flow collisions are avoided to a feasible extent. The third constraint states that the chosen path p has the maximum residual bandwidth among all available paths p_n.

3.2 Model description

In Oddlab, two main issues are considered: load balancing and flow scheduling. It is essential to detect and avoid potential link failures so that flows are scheduled to healthy and symmetric paths. Initially, we assumed that all the DCN links functioned appropriately. Therefore, we leveraged hash-based routing (ECMP) and adaptive flow scheduling for flow scheduling depending on the number of installed flow entries belonging to active elephant flows beside the available bandwidth on the ports [15]. The primary purpose for the hashing part of the flows on ECMP to the pre-defined paths is to decrease the flow scheduling cost, thereby mitigating the controller overhead.

We considered three steps on the data plane to detect and avoid potential faulty links. In these steps, we established a temporal correlation between the loading state at the edge switches and possible link failure at the core switches. The main components of Oddlab and considered steps are presented in Fig. 3. The SDN controller represents the control plane, which contains the main functions of Oddlab and periodically collects network information. This information includes ports and flow states based on the OpenFlow protocol. Subsequently, the obtained data were utilized to determine the best paths between every pair of end-hosts in the data plane and to label the loaded edge switches. Thus, in Step 1, the labeling process (the red

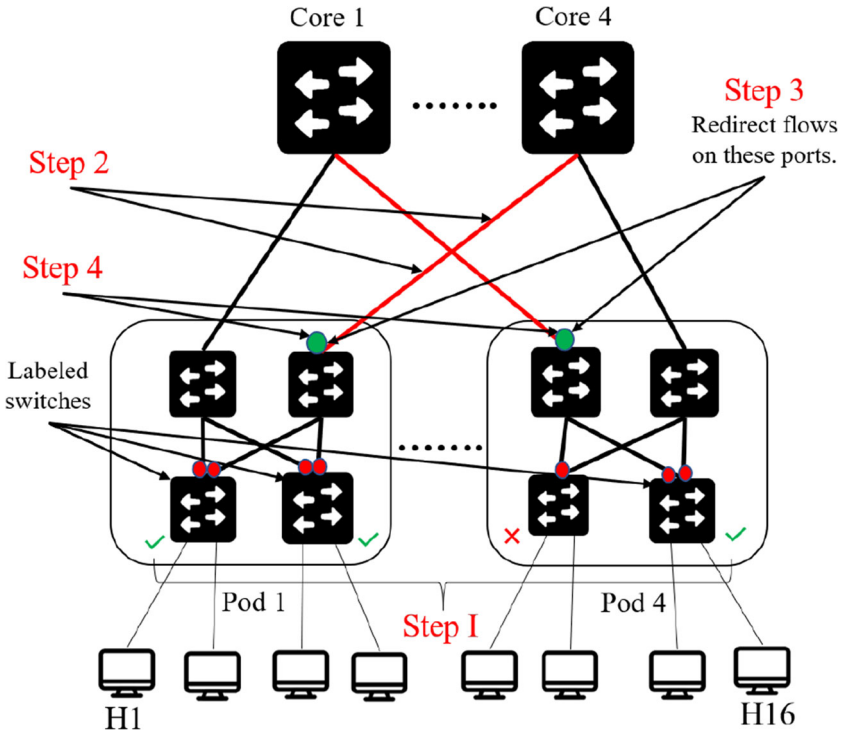

Fig. 3 OddLab approach on $K-4$ fat-tree DC

label in Fig. 3) starts after the SDN controller detects in any monitoring period that specific amounts of traffic pass on the upstream side of the edge switches (Ports 1 and 2). In this step, the ratio of the number of edge switches that meet the utilization condition to the number of non-loaded switches is calculated (the $Odds$ calculation) to determine the extent of DCN utilization. As depicted in Fig. 3, three edge switches fulfilled the utilization condition, except one edge switch in Pod 4, indicating that DCN is in the utilization state. Whenever the controller determines that most edge switches are sufficiently utilized, Step 2 begins by searching the faulty links, whose load level falls beyond 1% of the total link capacity. In Step 3, all the elephant flows found on the detected faulty links are rescheduled to other healthy paths. Subsequently, the adaptive flow scheduling model is updated; thus, the incoming flows avoid the affected paths. The entirely failed links typically continue to convey less throughput, while the DCN is in the loading state (Step 1). Consequently, the weighting function of the ECMP hashing at the aggregate switches is altered in Step 4 to avoid these links even in the proactive paths. Finally, an alert is also issued, including the information of the affected links, so that the DCN administrator can handle the failed links.

4 Oddlab framework within the SDN paradigm

The proposed model was designed using the control and data planes of the SDN paradigm. As shown in Fig. 4, we utilized Oddlab in three phases for flow scheduling and detecting failed links. The first phase resides in the DCN data plane, where the future flows are randomly hashed at the edge switches owing to the adopted flow sampling technique (1:1). The hashing technique is built based on the OpenFlow group tables supported by Open vSwitch in the data plane. Thus, a layer three packet forwarding occurs so that when the incoming packet can be handled by the controller when it does not match the pre-installed flow entries or the proactive group in the ECMP path. Moreover, the second phase is in the control plane and contains two main sub-models. The first sub-model is used for port stats polling and storing them into a directed graph, and the second one is used for $Odds$ calculation and labeling based on the edge switch consumption. In the third phase, the best path calculation and link health checking are estimated based on the network-directed graph's information to define the healthy paths and update the aggregate switch bucket weight in the data plane.

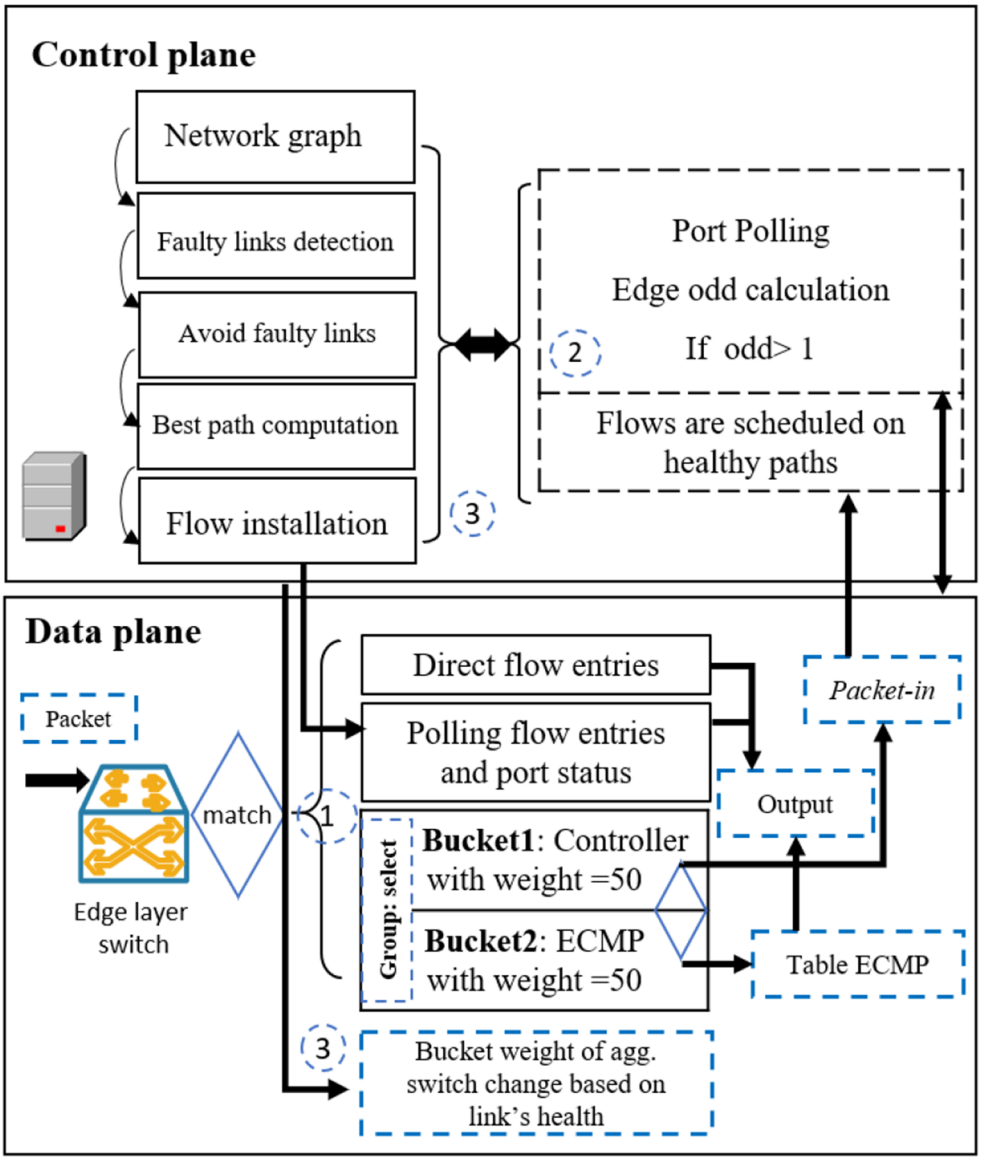

Fig. 4 OddLab SDN architecture

4.1 Flow sampling at the DCN edges

Leveraging the SDN controller to compute an appropriate path for each flow in the network is not feasible. In Oddlab, we utilized an edge flow sampling method, which was introduced in our previous study [13]. Thus, we sample the incoming flows based on the OpenFlow SELECT group type with two buckets; thus, the *packet_in* request of a flow is either directly scheduled to the ECMP pre-defined path or sent to the SDN controller. The selected group is defined with weighted buckets, and each bucket can perform specific actions on the switch port, that is, dropping or forwarding the packets. As depicted in Fig. 5, identical weights have been set for the proactive path (ECMP table) and control handling.

We defined multiple flow entries, including direct and polling flow entries at the edge switches. The direct flow entries contain the target end-host's IP address and a direct output port directly connecting the host to the edge switch. However, the polling flow entry contains information

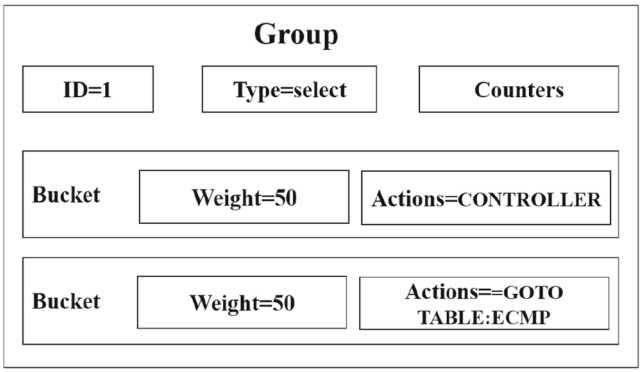

Fig. 5 The sampling group entry at the edge switches

on (*Ip_src, Ip_dst, transport_src_prt, transport_dst_prt, and action: the output of the edge switch port*). Subsequently, if the incoming packet does not match any existing flow entries at the edge switch, it is dynamically forwarded to the Oddlab controller or the ECMP table with a uniform probability.

4.2 ECMP proactive scheduling

ECMP proved to be a fast flow scheduling technique because it spread the flows across all available paths without considering the path status. Therefore, flow collisions frequently occur, causing packet losses and significant delays in the FCT. We reduce such collisions and congestion using an efficient adaptive flow scheduling of the SDN controller.

Additionally, we defined the SELECT group table with two buckets on the upstream side of the DC switches for ECMP path implementation. For instance, we established two buckets on the edge switches connected to the aggregate switches with actions (OUTPUTPORT:1 and OUTPUTPORT:2) and identical bucket weights to balance the incoming flows between the upstream ports evenly (Fig. 5). Consequently, the same procedure is applied to aggregate switch ports connected to the core switches on the upstream ports (Ports 1 and 2). In the fat-tree DCN topology, the DCN downstream side traffic cannot be balanced because the link is directly connected to the destination end-host through the core switch and aggregate switch until the edge switch. Therefore, we defined flow entries with fixed priority values for the directly connected sub-networks on the DCN downstream ports (Ports 3 and 4).

4.3 Oddlab SDN-based adaptive model

One of the objectives of Oddlab is to identify the potential faulty links based on the only DCN traffic. We observe and correlate two subsequent events inside a DCN between the edge switches (traffic source) and core switches (intermediate nodes) to achieve such a mission. The first event includes estimation of the DCN utilization state, calculated based on the traffic passing through both the upstream ports of the edge switches. The second event involves the presence of core switch links within an underutilized state (i.e., passing throughput less than the pre-defined threshold).

Algorithms 1, 2, 3, 4, and 5 illustrate the fundamental functionalities of the adaptive Oddlab model. This section presents the design perspectives and the main decisions of the framework.

Algorithm 1 Handling of port information for DC switches.

Data: $G=(V,E)$, *path*, *min_bw*, P_r, *dpid_list*, $aff\_edge\_list[dpid][prt\_no]$, *speed*, global ($af\_links[dpid][prt\_no]$), global *proac_ports* $all\_edges = 8$ edge switches in k-4 fat-tree
Result: *mini_bw_link*, *Odds*, *af_links[dpid][prt_no]*

```
healthy_links[dpid][prt_no] = [ ]
foreach P_r do
    Function OFPPortStatsRequest(dpid_list):
        Function save_free_bw(dpid, prt_no, speed):
            free_bw =capacity-speed
            G[j][i][j][i+1](bw) ← free_bw
            if dpid in edge_list and prt_no in [1,2] and free_bw ≤ edge_thr then
                aff_edge_list[dpid][prt_no] ← loaded
            for d in aff_edge_list[dpid][prt_no] do
                if prt_no [1,2] is loaded then
                    // Number of affected edge switches.
                    aff+=1
                    not_aff=all_edges - aff
                Odds= aff / not_aff
            return G[j][i][j][i+1](bw), Odds
        if dpid or dpid in edge_list and port_no in [1,2] then
            call save_health (dpid, prt_no, speed)
        Function save_health(dpid, prt_no, speed):
            // This function invoked periodically to estimate the path health.
            dpid_prt_health = speed / capacity
            fault_iter =0
            if dpid_prt_health ≤ health_thr and Odds > 1 then
                if prt_no in healthy_links[dpid] then
                    fault_iter = fault_iter+1
                    healthy_links[dpid][prt_no] ← health, fault_iter, Odds
                    G[j][i][j][i+1](src_prt) ← src_prt
                    G[i][j][i][j+1](dst_prt) ← dst_prt
                    G[j][i][j][i+1](health) ← health
                    G[j][i][j][i+1](fault_iter) ← fault_iter
            if prt_no in healthy_links[dpid] and dpid_prt_health > health_thr and prt_no in af_links[dpid] and prt_no in proac_ports[dpid] then
                // To delete false-positive faulty links and restore them to normal status.
                fault_iter =0
                af_links[dpid].remove(prt_no)
    return af_links[dpid][prt_no], Odds, G[j][i][j][i+1](bw)(health)(fault_iter)(src_prt)(dst_prt)
    Function min_bw_links(G(V, E), path, min_bw):
        // To estimate the path bottleneck.
        for i in len(path)-1 do
            bw = G[j][i][j][i+1](bw)
            mini_bw_link = min(bw, min_bw)
        return mini_bw_link
```

Algorithm 2 Oddlab adaptive flow scheduling.

Data: *G=(V,E)* , *src_IP*, *dst_IP*, *src_prt*, *dst_prt*, *min_bw = capacity*, *max_bw = 0*, *max_fnum = 0*, *path_health = 0*, *k*, *shortest_p*= { }, global(*Odds*, *af_links*, *proac_ports*)
Result: *best_path*= [ ], *failure_alert*

```
while Odds ⩽ 1 do
    // Initially, calculate the shortest path that has the least elephant active flows and high available bandwidth.
    for i in (0, k) do
        if shortest_p [i]= shortest_paths (G, src_IP, dst_IP) then
            for j in shortest_paths do
                max_fnum = maxfnum_of_path (G, j, max_fnum)
                fnum_of_paths (src_IP, dst_IP) ← max_fnum
                min_fnum_path = min(fnum_of_paths (src_IP, dst_IP,max_fnum))
            max_bw_of_paths = 0
            for f in fnum_of_paths do
                min_bw = bottleneck_of_path (G, j, min_bw)
                if min_bw > max_bw then
                    max_bw = min_bw
                best_path = f
return best_path
while Odds > 1 do
    // In utilized DC, the best path will be determined based on the shortest and healthy path with the least elephant active flows and high available bandwidth.
    for i in (0, k) do
        if shortest_p [i]= shortest_paths (G, src_IP, dst_IP) then
            for h in shortest_paths do
                path_health = path_health_check (G, j, link_health)
                if path_health ∀e ∈ path ==0 then
                    max_fnum =0
                    path_health (src_IP, dst_IP) ← max_fnum
                else
                    max_fnum = ∞
                    path_health (src_IP, dst_IP) ← max_fnum
            path_health (src_IP, dst_IP) ← max_fnum
            for j in path_health do
                max_fnum = maxfnum_of_path (G, j, max_fnum)
                fnum_of_paths (src_IP, dst_IP) ← max_fnum
                min_fnum_path = min(fnum_of_paths (src_IP, dst_IP,max_fnum))
            for f in fnum_of_paths do
                min_bw = bottleneck_of_path (G, j, min_bw)
                if min_bw > max_bw then
                    max_bw = min_bw
                best_path = f
return best_path, failure_alert(af_links & proac_ports)
```

Algorithm 3 Path flow number estimation.

Data: *G=(V,E)* , *path*, *max_fnum*, global(af_links)
Result: *max_flownum*

```
for i in len(path)-1 do
    src_prt = G[j][i][j][i+1](src_prt)
    dst_prt = G[j][i][j][i+1](dst_prt)
    fault_src_itr = G[j][i][j][i+1](fault_src)
    fault_dst_itr = G[i][j][i][j+1](fault_dst)
    fnum = G[j][i][j][i+1](fnum)
    // This condition will avoid detected links when fault iteration reaches the fault_iter threshold.
    if (i in af_links and src_prt in af_links[j] and dst_prt in af_links[i] and fault_src ⩾ fault_iter and fault_dst ⩾ fault_iter) then
        max_flownum = ∞
    else
        max_flownum = max(fnum, max_fnum)
return max_flownum
```

Algorithm 4 Path health estimation.

Data: *G=(V,E)* , *path*, *link_health*, P_r, *health_thr*, global*(af_links)*, *fault_iter*
Result: *path_health*

```
foreach P_r do
    health_list = [ ]
    non_health_list = [ ]
    proac_ports= [ ]
    for i in len(path)-1 do
        src_prt = G[j][i][j][i+1](src_prt)
        dst_prt = G[i][j][i][j+1](dst_prt)
        fault_src_itr = G[j][i][j][i+1](fault_iter)
        fault_dst_itr = G[i][j][i][j+1](fault_iter)
        _health_src_prt = G[j][i][j][i+1](health)
        _health_dst_prt = G[i][j][i][j+1](health)
        if (_health_src_prt >health_thr and fault_src_itr ⩽ fault_iter and _health_dst_prt > health_thr and fault_dst_itr ⩽ fault_iter) then
            health_list ← _health_src_prt)
        else
            if (_health_src_prt < health_thr and fault_src_itr > fault_iter and _health_dst_prt < health_thr and fault_dst_itr > fault_iter ) then
                non_health_list ← (_health_src_prt)
                if path[i] not in af_links then
                    af_links[path[i]] ← [src_prt]
        if path[i+1] not in af_links then
            af_links[path[i+1]] ← [dst_prt]
        else
            if dst_prt not in af_links[path[i+1]] then
                af_links[path[i+1]] ← [dst_prt]
        // The following statement for proactive links changing on the aggr-core switches in case of long-lived failures.
        if path[i] in agg_swt and path[i] not proac_ports and src_prt in [1,2] and fault_src_iter ⩾ 5 and fault_dt_iter ⩾ 5 then
            ovs-ofctl mod-group (path[i], src_prt)
            proac_ports ← (path[i],src_prt)
    if len(non_health_list)>0 then
        path_health= ∞
    else
        path_health= 0
return path_health
```

Algorithm 5 Handling of flow information for DC switches.

```
Data: G=(V,E), dpid_list, path, flow_list,
      link_health, P_r, health_thr, global(af_links)
Result: red_links[dpid][prt_no]
1  foreach P_r do
     Function EventOFPFlowStatsReply(dpid_list):
2      for f in in flow_list do
3        flow_net = flow_byte / flow_duration
4        if flow_net ≥ 50 Kbps then
5          fnum[dpid][prt_no]
             ←fnum[dpid][prt_no]+1
6          G[j][i][j][i+1](fnum) ← fnum
7          if len (af_links) is not null and Odds > 1
             then
               // Rerouting the congested
                  elephant flows to alternative
                  paths.
8              for d in af_links do
9                for p in af_links[d] do
10                 if af_links[d][p] ∩
                     red_links[d][p] is true then
11                   continue
12                 else
                     dpid, prt_no = d, p
13               for fd in flows installed on
                   (dpid, prt_no) do
14                 if fd_load ⩾ 50 KB then
15                   E_count++
16               for e in E_cont do
17                 if e[dpid][prt_no] not in
                     red_links then
                     red_links ←
                       e[dpid][prt_no]
18                   Based on src_ip of each
                       flow e, get e_flow_info
19                   get_best_path
                       (G, e, e_flow_info)
                       // The congested
                       flows rerouted
                       based on the
                       execution of
                       algorithm 2.
20 return red_links[dpid][prt_no], G[j][i][j][i+1](fnum)
```

4.3.1 DCN utilization state estimation

We consider the DCN utilization state estimation as valid evidence that the detected congested links in the core layer are potential link failures with high probability $P(failed\_link|loaded\_DC)$. As depicted in Algorithm 1, Oddlab periodically invokes all port statistics from the switches using the OFPPORTStats messages based on the pre-defined polling rate ($P_r = 2$ s). We leverage the collected port information in several functions to estimate the DCN loading state, such as finding a port free bandwidth function ($save\_free\_bw$), finding a path bottleneck (min_bw_links), and detecting the faulty links ($save\_health$). First, the $save\_free\_bw$ function is defined to store the throughput consumption information ($free\_bw$) of the DCN switch ports ($dpid$, prt_no) in a directed graph $G(V, E)$. Thus, the primary flow scheduling procedure relies on the reserved graph free bandwidth values (min_bw_links) when defining the path bottleneck. Gradually, this function guarantees that the selected path has the best available bandwidth from the available paths between each end-host.

However, we utilize the obtained information of port ($dpid$, prt_no) consumption ($free\_bw$) to determine the DCN utilization. To this end, we only filter the available bandwidth values of the edge switch ports on the upstream side (Ports 1 and 2). Here, the edge switch is labeled as loaded L_e if the residual bandwidth value of each port is at least 95% of the link capacity for both ports. Subsequently, the $Odds$ value is calculated based on the number of labeled switches to the number of unlabeled switches NL_e (Eq. 7).

$$Odds = \frac{\sum L_e}{\sum NL_e}. \tag{7}$$

Table 2 presents the variables used in Oddlab algorithms.

4.3.2 Oddlab adaptive flow scheduling

The adaptive method enhances the scheduling performance by avoiding hashing collisions and choosing the best paths with minimum flow entries for active elephant flows and better residual bandwidth. Consequently, we define the flow entries with a 5-tuple <IP protocol, src port, src IP, dst port, and dst IP> based on the OpenFlow protocol. The Oddlab controller reacts in two ways to the $packet\_in$ requests based on the estimated $Odds$ value of the edge switches. Therefore, when the DC is not fully utilized (i.e., $Odds \leqslant 1$), the $packet\_in$ reaction regarding the $best\_path$ will be based on finding the path with less installed elephant flows and a high available bandwidth, as shown in Algorithm 2. In this case, the $maxfnum\_of\_path$ explained in Algorithm 3 and $bottleneck\_of\_path$ in Algorithm 1 is invoked to indicate the best available path between the end-hosts ($best\_path$). Additionally, flow entry statistics obtained from the EventOFPFlowStatsReply function will be collected and stored in the graph $G = (V, E)$ to estimate the number of active elephant flows. Subsequently, only flow entries with a data transfer speed of $\geqslant$ 50 kbps [15] are counted as active elephant flow entries ($fnum$), as illustrated in Algorithm 5 (line 4). Therefore, in this step, the path containing the minimum active flows is considered as the $best\_path$. Next, the free bandwidth values ($free\_bw$) of all paths are gathered and stored in the graph $G = (V, E)$

Table 2 Oddlab's variables

Variable	Description
min_bw	The bottleneck bandwidth of the link.
max_bw	Maximum available bandwidth of the path.
k	Fat-tree DC order.
$shortest\_p$	List of the shortest paths between src_ip and dst_ip.
$best\_path$	The best available healthy and lightest path between src_ip and dst_ip.
$edge\_thr$	Available bandwidth on the edge switch ports (1, 2).
$dpid\_list$	List of switches' IDs.
p_r	Polling rate.
$flow\_list$	List of the affected active elephant flows on the aggregate switch port.
aff_edge_list	List of the loaded edge switches.
af_links	List of the faulty links between aggregate and core switches.
$proac\_ports$	List of aggregate switches with faulty ports whose ECMP bucket weight has been modified.
$health\_thr$	Threshold of the faulty link.
non_health_list	List of the affected switches and ports' IDs.
max_fnum	The number of active elephant flows on the port.
$fault\_iter$	Threshold of the fault iteration of the port (src_prt or dst_prt).

based on the OFPPortStatsRequest function, as explained in Algorithm 1. Finally, the adaptive flow scheduling method chooses the path with the least active elephant flows, and the lightest is loaded as the $best\_path$. This method guarantees that each flow, regardless of the flow demand, obtains the best possible path ($best\_path$); however, not necessarily the shortest path. Consequently, most flows do not agglomerate into a single short path. Hence, the flow load-balancing equation in Oddlab is as follows:

$$U(u, v) = \frac{\sum_{i=1}^{k} f_i(u, v)}{mf(u, v).(mb(u, v))}, \tag{8}$$

where mf represents the $maxfnum\_of\_path$ between the u and v nodes, and mb is the $bottleneck\_of\_path$ value between the same nodes.

Furthermore, Oddlab leverages the number of effective elephant flows inside a DCN to predict the future state of the path as compared to that presented in our previous study [13]. Therefore, Oddlab eliminates the need to frequently redirect elephant flows to achieve better FCT values.

4.4 Detection of the faulty links based on spatial-temporal correlation

First, the SDN controller and OpenFlow protocol lack native access to the network end-hosts. Therefore, the congestion information obtained from the application layer that can be used to determine the flow latency and packet losses may be overlooked. In Oddlab, we only estimated the effect of link failure based on the network throughput. In this module, we estimated the amount of traffic mounted on the failed link during the failure.

In DCN, link failure can be classified into hardware and software failures based on the causes of failure and duration. Hardware failures are typical, which are caused by physical faults in the switch's ports. Nevertheless, software failures are caused by software bugs or IOS hotfix issues [3]. In addition, hardware failures are known to be long-lived failures that may require hardware replacement. Simultaneously, the software type is considered short-lived and can be automatically resolved comparable to a root guard in the spanning tree protocol [3]. Thus, we adopted the definitions of both faulty links in Oddlab.

The DCN traffic properties include similarities, periodicity, and correlation [28]. Hence, we leverage the spatial-temporal correlation between the two events in the input data space on the edges and crossing the DCN over the aggregate and core switches at different time intervals to detect faulty links. The spatial-temporal correlation used in detecting the faulty links demonstrates how the two conditions of the DCN loading state and underutilized links fit together in the DCN operation life span. As shown in Fig. 6, the controller monitors and estimates the edge load and links' health based on the pre-defined thresholds. Whenever both events are correlated in consecutive monitoring intervals (i.e., from $t3$ to $t5$ for short-lived failures), the controller indicates the affected link as "$unhealthy\ link$", and redirects all elephant flows found on the upstream port of the link. As depicted in interval $t6$ in Fig. 6, the correlation is broken; however, the link failure threshold is still

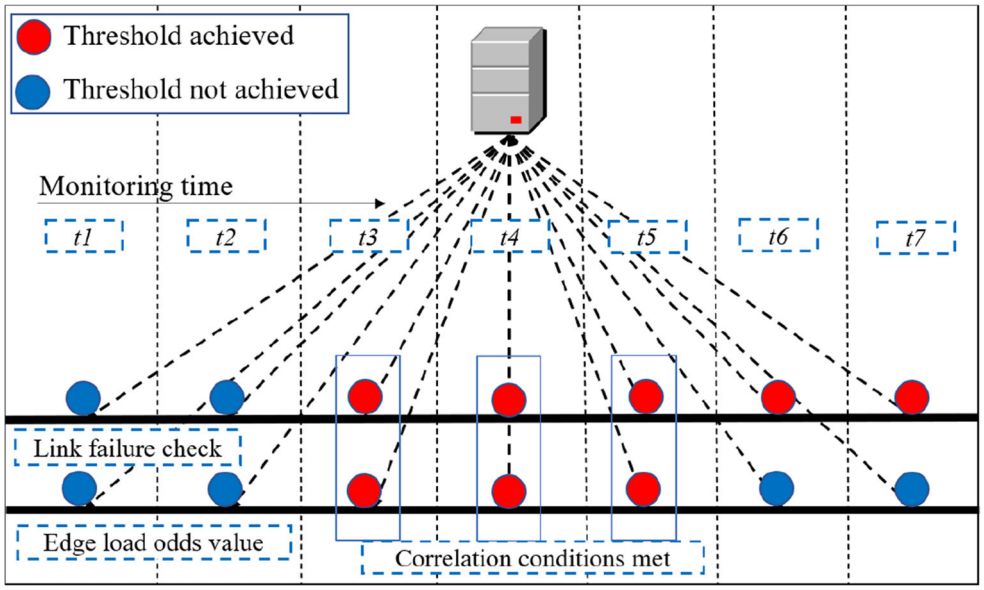

Fig. 6 Process of spatial-temporal correlation to detect potential link failure

fulfilled. In this case, the path that contains the detected link is avoided in the adaptive flow scheduling. When the link is recovered (i.e., receiving regular traffic) from the proactive paths using ECMP, it will be excluded from the detected-links list.

Faulty link detection procedure When the number of edge switches that fulfill the minimum loading threshold exceeds the number of unloaded switches (i.e., $Odds > 1$), the Oddlab controller invokes the second initiated reaction to obtain the best path, as depicted in ($path\_health\_check$) in Algorithm 2. The links' health is periodically estimated based on the port statistics initiated in OFPPortStatsRequest together with the $save\_free\_bw$ function, as shown in Algorithm 1. First, the links of core switches are considered healthy unless their throughput falls below the failure threshold (i.e., $health\_thr \leqslant 1\%$ of the link capacity). The number of consecutive faulty link frequencies is calculated over the network monitoring duration ($fault\_iter$) to maintain the correlation conditions ($health\_thr$ and $Odds$), as described in Algorithm 1 (line 17). When the correlation is fulfilled, the faulty link information is saved on the directed graph $G(V, E)$, including ($src\_prt, dst\_prt, health, fault\_iter$). Moreover, the correlation parameter of the detected faulty links ($fault\_iter$) is nulled when the link throughput increases above the threshold (i.e., $health\_thr > 1\%$ of the link capacity), as illustrated in Algorithm 1 (line 25). The information of the genuine faulty links, including $dpid$ and src_prt, is stored in a list (af_links) to be applied as elephant flow rescheduling information, as shown in Algorithm 4 (line 22).

4.4.1 Elephant flows rescheduling

The Oddlab controller uses the af_links list information to reschedule all congested elephant flows on these links. As illustrated in Algorithm 5 (line 7), the flow information of all ports is periodically monitored to participate in the flow scheduling process. Thus, when the value of $Odds > 1$, all elephant flows mounted on the upstream side of the aggregate switch ports ($dpid$ & prt_no) with a cumulative flow size of $\geqslant 50$ kB [9, 13] will be rescheduled to other healthy paths. As explained in Algorithm 2, these flows will be redirected from the end-to-end host based on the flow source IP address (src_ip) to new healthy paths ($best\_path$). The reason for choosing end-to-end redirection is to obtain the shortest healthy path instead of overwhelming more links if we decide to reroute from the aggregate switch. Moreover, the elephant flow scheduling was decided to provide the mice flow with more resources to pass through the affected links. In addition, rescheduling mice flows is not reasonable because they are small and may be significantly affected by the completion time. Finally, the rescheduled flow information is stored in the red_links list with the switch and port number information ($dpid$ & prt_no); thus, the port will not be checked in the following flow monitoring round.

4.4.2 Adaptive flow scheduling after detecting the faulty links

The identified faulty links in the (af_links) list were avoided by the Oddlab adaptive flow scheduling method. As shown in Algorithm 2 (line 15), the Oddlab second reaction

relies on the estimated $Odds$ value ($Odds > 1$). The algorithm primarily guarantees that the determined paths are free of any faulty links in the core switches. Therefore, the health of the path is checked before computing the active flow numbers of the path. For simplicity, we define a high value for $max\_flownum$ whenever a faulty link is detected in any path to estimate the path flow number; thus, the adaptive flow will not choose that path later in the adaptive flow scheduling, as described in Algorithm 3 (line 8).

As an outcome of the faulty link detection, the flow load-balancing equation is as follows:

$$U(u,v) = \frac{\sum_{i=1}^{k} f_i(u,v)}{hp(u,v).(mf(u,v))(mb(u,v)))}, \tag{9}$$

where hp represents the checked healthy paths between the u and v nodes.

Moreover, even when the value of $Odds$ depreciates below 1, adaptive flow scheduling avoids the defected paths based on Eq. 9. In Algorithm 3 (line 7), we define a statement to check the status of the detected links (af_list) and avoid the links with a large number of $max\_flownum$ parameter in the best path decision $best\_path$.

For long-lived failures, we defined a higher value for $fault\_iter \geqslant 5$ (i.e., the monitoring time is equivalent to $5 \times P_r$). Accordingly, the proactive bucket weight of links that achieves this threshold will be modified; hence, the affected port will not obtain the last equal number of flows. Therefore, the weight value of the affected upstream port is omitted using the OpenFlow **ovs-ofctl mod-group** message ($dpid$, src_prt), as described in Algorithm 4 (line 23). Eventually, an alert is triggered for the detected faulty link, with the link information and whether the link has been excluded from the service (i.e., hardware failure), $failure\_alert = (af\_links \ \& \ proac\_ports))$.

5 Oddlab analysis

In this section, the main characteristics of Oddlab are analyzed and evaluated regarding the adopted DCN edge flow sampling procedure and complexity of the adaptive flow scheduling.

5.1 Edge sampling performs under a finite system

We assume that the flows arrive at the DCN edge switches following a Poisson stream with a rate of λ_n. At the edges, we adapted two identical bucket-forwarding decisions ($d = 2$). Therefore, the incoming flows are independently hashed either to the ECMP or SDN controller uniformly. In particular, the flows are served based on the first-in-first-out manner with an exponential distribution for the service time of flows. This phenomenon is associated with the supermarket model discussed in [27]. Therefore, we used the supermarket model to provide a precise and systematic analysis of the Oddlab load-balancing model behavior when the number of flows tends to be significant ($n \to \infty$). To obtain the expected waiting time of the flow in addition to the maximum queue length, Kurtz's theorem for large numbers and Chernoff-like bounds [27] are applied as follows:

Theorem 1 *The anticipated time that the flow can wait at the DC edge to obtain the forwarding service, whether with ECMP or Controller ($d = 2$) over the period $[0, T]$ is limited to*

$$\sum_{i=0}^{\infty} \lambda^{\frac{d^i - d}{d-1}} + O(1), \tag{10}$$

where $O(1)$ depends on the T and λ states, and represents the error bound between the system state with fixed n, and when n goes to infinity.

Theorem 2 *The maximum initial queue length on the DC edge switches with ($d = 2$) over the period $[0, T]$ is equal to*

$$\frac{\log \log n}{\log d} + O(1). \tag{11}$$

In contrast to the methods that rely solely on one decision at the edges $d = 1$, the expected waiting time is $1/1 - \lambda$. Hence, our proposed method presents an exponential improvement in the waiting time and queue length because the end of queues is reduced double-exponentially rather than single-exponentially when $d = 1$ [27]. In addition, the central controller will not be overwhelmed by a large number of $packet\_in$ requests. Hence, the controller can precisely handle the flows, and thus multiple applications can be performed in the SDN application plane without an excessive burden.

The central SDN controller is a central weakness of failure in DCN management. Consequently, several security solutions tend to use distributed controllers [29]. However, the proposed measure can be employed to defend the controller against DDoS attacks.

5.2 Complexity evaluation for the adaptive flow scheduling

As explained in Section 4.3.2, the best paths are periodically determined based on the DC edge loading status ($Odds$ value). The best paths are frequently optimized based on the acquired network parameters (i.e., residual bandwidth and the number of active elephant flows)

from the ports of the DC switches. The scheduling algorithm does not become complicated even after identifying the utilized switches and $Odds$ value; additional iterations will only be appended to the previously gathered parameters to learn the best paths. The added complexity is in the number of redirected elephant flows $|F|$ from the affected paths only when faulty links are identified.

We estimated the time and space complexity of Oddlab by considering the worst-case scenarios regarding DCN density (number of monitoring switch ports p), in addition to the average number of elephant flows $|F|$ inside the DCN traffic. The primary adaptive traffic scheduling algorithm depends on the collected port states to determine the best paths (EventOFPFlowStatsReply and OFPPortStatsRequest). In addition, the time complexity is $O(k^2)$ because we choose not to reroute the elephant flows in this step. Nevertheless, the time complexity includes the number of elephant flows so that the complexity will be $(O(k^2 + |F|))$ in the case of elephant flow rescheduling from the detected faulty links. As for the space complexity, the controller memory should maintain the number of active elephant flows, the residual bandwidth of each port, and the number of redirected elephant flows of the affected links on the upstream ports of the aggregate switches. For instance, as shown in Fig. 1, out of 80 ports in the $K-4$ fat-tree DCN switches, only 16 upstream ports connect the aggregate and core switches. Hence, the space complexity is $O(k^3 + |F|\frac{k}{5}))$.

Moreover, we proved that the sampling process with two buckets on edge switches reduced the controller overhead by half with the means of eliminating the $packet\_in$ requests to the controller at a time interval of $(t \rightarrow \infty)$ [13]. Additionally, we adopted a commercial DCN introduced in [9] with 10,000 end-hosts and 1,000 flows per second for each host to prove the simplicity and effectiveness of the proposed method when increasing the number of DCN end-hosts. However, within 2 ms as the median arrival time for each flow, we anticipate that the Oddlab controller can handle $\frac{10{,}000\times 1{,}000}{2} = 5,000,000$ (i.e., 2,500,000 $packet\_in$ requests only) [13]. This is considerably less than the number of flows a single controller can manage per second (more than 12 million requests [7]). Consequently, Oddlab implementation is highly achievable in hardware such as NetFPGA OpenFlow switches [2] owing to uncomplicated arithmetic operations and less overhead.

6 Oddlab model implementation

The proposed model was implemented based on our prior study [13]. Oddlab operates as a Python application within the Ryu [30] controller utilizing mininet [31] as a real-time network emulator to establish a multi-rooted $K-4$ fat-tree DCN (for example, Fig. 1) in addition to OpenFlow protocol 1.3.1 as a communication protocol. The testbed topology includes 16 hosts interconnected to twenty 4-port OpenFlow switches and four pods with four core switches. Additionally, we set the link capacity through the DCN to 10 Mbps to maintain the connection in the real-time environment of the mininet. The controller is connected to each DCN switch to collect the required network statistics and maintain the network flows, as illustrated in Fig. 1. Our DCN deployment was implemented using a commodity PC with an Intel Core i5-8400 2.80 GHz CPU, 16 GB RAM running Ubuntu 16.04.

6.1 Experimental environment and evaluation metrics

We conducted extensive experiments on different traffic patterns with synthetic and realistic workloads of productive DCNs. The traffic scenarios are conducted both symmetrically, in which all links are healthy, and asymmetrically with a certain number of unhealthy links in the DCN to evaluate the performance of Oddlab in addressing these challenges. In this scenario, we use the same communication pattern applied to evaluate the performance of Hedera [2], introduced in [32]. The generated communication pattern consists of random and staggered probability patterns, according to the following details:

1. Random: every end-host transmits traffic to another end-host in the DCN with a uniform probability.
2. Staggered probability $(Edge\_p, Pod\_p)$: every end-host transmits traffic to another host in the same edge with a probability of $(Edge\_p)$, to the same pod with a probability of (Pod_p) and to other pods in the DCN with a probability of $(1 - Edge\_p - Pod\_p)$.

The simulation results involve the following fundamental QoS metrics:

- *Average bisection bandwidth*: One of the most important features of a multi-rooted topology is to grant full bisection bandwidth among the connected end-hosts. However, without an efficient flow scheduler, the amount of the DCN bisection bandwidth considerably decreases as it appears in ECMP owing to flow collisions [2]. The average accumulative throughput received at the downstream side of the edge switches is compared to the full bisection bandwidth of the fat-tree DCN topology.
- *FCT*: It denotes the efficiency of the flow scheduler algorithm to deliver different flows rapidly. For Oddlab, we present the overall average flow completion time (AFCT) of the transferred flows for each end-host.

- *Link utilization*: It refers to the average DCN link consumption comparing to the actual capacity. Here, we determine how the proposed scheduling method will achieve a moderate link utilization so that links will not become prone to congestion.
- *Faulty link detection*: As mentioned in Section 4.4, failed links can occur for several reasons and directly affect the multi-rooted DCN. In general, failed links degrade the overall DCN throughput and notably increase the FCT due to the lack of symmetry of the multi-rooted DCN topology. Therefore, we present the results when Oddlab detects these links and redirects the affected elephant flows.

We compare the obtained results with those of ECMP as a standard existing industrial flow forwarding method, in addition to Hedera [2] as a dynamic elephant flow scheduling method, and PureSDN [15] as a performance reference for a completely adaptive flow scheduling solution that solely depends on the SDN controller. Besides, we compare the obtained results from the bisection bandwidth experiment with those of the ideal network situation using nonblocking, where the switch is directly connected to all end-hosts. In the FCT experiment, we included the results presented in our previous study (Sieve [13]) in the comparisons because it has been proposed to reduce the FCT in a symmetric DCN topology.

7 Experimental results

This section discusses and analyzes the results obtained for the adopted load-balancing and flow scheduling strategies in symmetric and asymmetric DCNs, including faulty link detection. Moreover, we made the data on the Oddlab GitHub repository[1] available publicly.

7.1 Performance under symmetric DCN topologies

In this scenario, we conducted two benchmark tests. The first test includes the average bisection bandwidth and link utilization tests. The second test was used to examine the FCT. We performed tests on 16 hosts of the DCN topology, where flows were generated using TCP Iperf [33] according to the traffic patterns described earlier. We repeated the average bisection bandwidth test for 10 independent runs to obtain more reliable and realistic results. Each run lasted 60 s, during which the average bisection bandwidth was accumulated using bandwidth-NG (BWN-NG) [34] on the edges downstream, as implemented in the case of Hedera [2] and [15].

[1] https://github.com/aymeniq/Oddlab

Figure 7 presents the average bisection throughput obtained under the randomized and staggered patterns with stressed TCP flows. The experimental results confirm that the static hashing method (ECMP) achieved the lowest throughput owing to collisions produced by scheduling multiple elephant flows along the same path. Oddlab outperformed Hedera and ECMP in random and stag0.1_0.2. Moreover, most of the traffic (70% in stag0.1_0.2) will be among different pods in these patterns. Thus, most flows benefit from the bisection bandwidth granted by the multi-rooted DCN topology. The average throughput in the rest patterns is slightly higher because of the resource contention, and the collision rate decreases when the traffic is within the same edge switch or in the same pod.

The PureSDN method outperforms the random and stag0.1_0.2 patterns at a high controller overhead expense. It depends on the central controller to find an optimal path for each incoming flow. In addition, Oddlab scheduling depends on the central iterative optimization with the help of ECMP, which enables the proposed method to operate with less burden on the controller and delivers a notable improvement in the average bisection bandwidth.

Figure 8 illustrates the CDF (Cumulative Distribution Function) of Oddlab link utilization compared to that of the other scheduling algorithms under different traffic patterns. The majority of the links were underutilized in the case of ECMP. For instance, the random and stag0.1_0.2 patterns for ECMP showed that 50% of the links (x-axis) were occupied with less than 20% of the total capacity (y-axis) owing to entirely random flow scheduling. Compared with the other comparison methods, Oddlab performs at the medium level under various traffic patterns. Analyzing the stag0.1_0.2 pattern reveals that 70% of links were occupied by 75% of capacity in Hedera, 82% in Oddlab, and 90% in PureSDN.

Consequently, most of the links in PureSDN reached an overutilized situation, which makes it prone to more congestion, and thus increases the FCT.

The FCT experiment was conducted to prove the effectiveness of Oddlab on the flow scheduling by reducing the flow delay. To this end, we applied real workloads to obtain the inter-flow dynamics and traffic burstiness, as used in a production DCN. Hence, two different workloads were utilized: web search [35] and cache jobs [12]. In the web search dataset, 33% of the data had a size of less than 1 kB, 93% had a size in the range of 1–10 kB, and that of the rest was between 10 and 300 kB. Regarding the cache workload, the data size distribution was as follows: 18% of the data had a size of less than 0.1 kB, 38% had a size in the range of 0.1–10 kB, less than 90% had a size of 1 MB, and the rest had a size of less than 10 MB. In addition, each end-host in the DCN started to transmit samples of the workloads (16 flows for each dataset) to another end-host based on the same traffic patterns to implement the test in

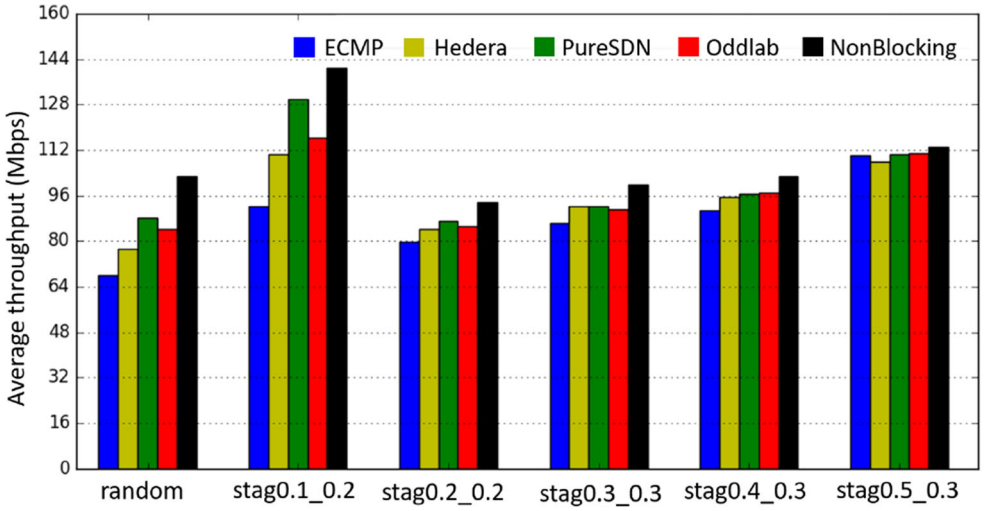

Fig. 7 Average bisection throughput for the comparison TE methods under different traffic patterns

the mininet environment. The flows are initiated based on the Poisson process with a certain mean value to simulate realistic traffic between the DCN end-hosts. Therefore, 512 flows were transmitted within 126 s as the traffic simulation time for each traffic pattern.

Figure 9 presents the average overall FCT for the scheduled flows using each method under different traffic patterns. Oddlab significantly reduces flow delays with an increasing number of flows passing through the bisection part of the DCN (refer to random pattern) when compared with the other comparison methods. As shown in Fig. 10, Oddlab reduces the overall average FCT by up to 30, 25.7, 62, and 5% compared to that of ECMP, Hedera, PureSDN, and Sieve, respectively. Nevertheless, PureSDN relies on the wildcard flow entry rules for TCP flows, whereas flows that belong to the same source and destination IP addresses are scheduled on the same path. Hence, the selected path in the case of PureSDN is highly saturated.

Moreover, Hedera and ECMP perform in significantly different ways. Because Hedera scheduling for mice flows is based on ECMP, it only reroutes elephant flows to the best path once a flow reaches 10% of the link capacity. Consequently, flow collision is inevitable, which necessarily delays the arrival of the flows. As for Sieve, the frequent elephant flow rerouting reduces the delay for the mice flows, as proved in [13]. This procedure affects the arrival time of the elephant flows. In addition, the newly added flow entry rules into the DCN switches manage the rerouted elephant flows. The adaptive scheduling of Oddlab attempts to avoid congested paths without the necessity

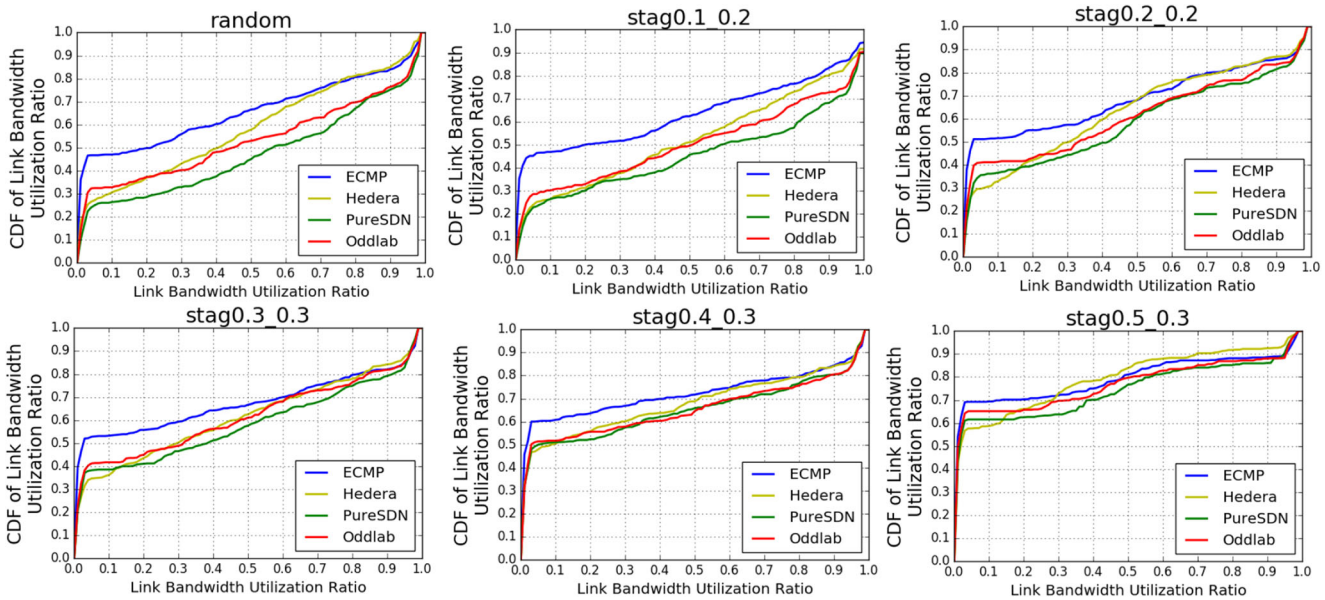

Fig. 8 CDF of link bandwidth utilization for the four algorithms under different traffic patterns

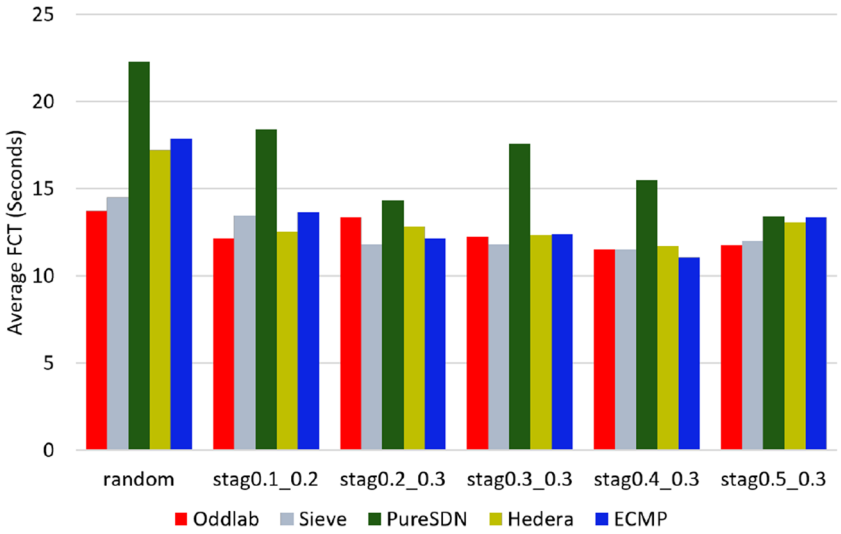

Fig. 9 Average overall FCT for the five methods under different traffic patterns

for rerouting the elephant flows. Figure 9 shows that ECMP and Hedera outperform in reducing the average FCT in the traffic patterns stag0.2_0.3 and stag0.4_0.3 where most of the traffic is inside the same pod. In this case, Oddlab may choose farther paths based on the shorter path conditions, which may affect the arrival time of the flows.

In summary, the symmetric DCN topology results reveal that the proposed flow scheduling of Oddlab delivers better average throughput, moderate link bandwidth utilization, and reduced average overall FCT as compared to the other the existing methods.

7.2 Performance under asymmetric DCN topologies

DCN topologies are imperatively asymmetric because of multiple factors, such as partial failures. In this experiment, we modified some link bandwidths to achieve an asymmetric fat-tree topology. To this end, we set the capacity of some links between the aggregate and core switches to shift to 5 Mbps, while other links remain at 10 Mbps [19]. This experiment used the same workloads as the web searches and cache jobs. Moreover, the FCT values for the flows were significantly affected by the bandwidth reduction. Therefore, we assessed the performance based on the achieved FCT by the comparison TE methods. In this experiment, we applied a random traffic pattern to allow sufficient flows to pass over the core switches. Hence, the asymmetric topology effect was recognized at the end-host application layer.

Figure 11 shows the achieved reduction of the overall average FCT values. The results demonstrate that the Oddlab scheduler can finish the overall flows within 19.4 s as an average FCT, which is better by up to 12, 8, and 10% compared with that of ECMP, Hedera, and PureSDN, respectively. This experiment proves that the decrease in the link bandwidth was not considerably high, and the adaptive scheduling of Oddlab achieved a better FCT reduction

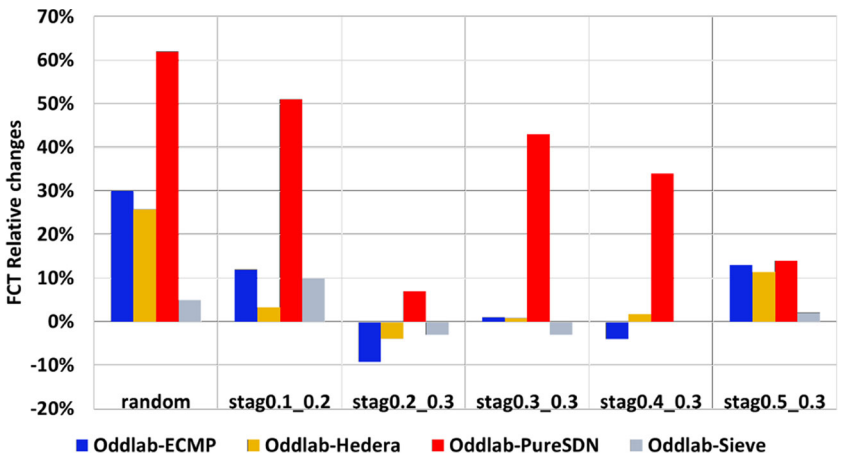

Fig. 10 Average overall FCT relative changes for Oddlab compared with ECMP, Hedera, PureSDN, and Sieve

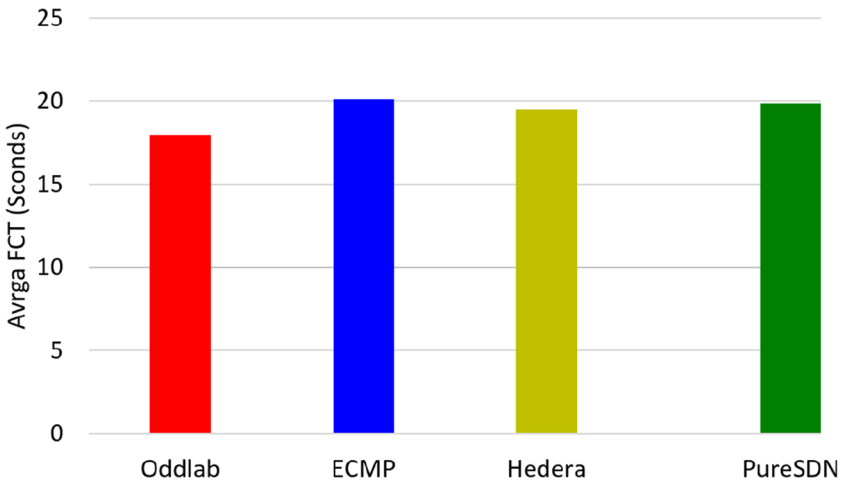

Fig. 11 Average overall FCT for Oddlab compared with ECMP, Hedera, and PureSDN under asymmetric DCN topology

by avoiding congested paths without a complicated flow rerouting.

7.3 Failure detection

The failure detection experiment was employed to investigate multiple failures between the aggregate and core switch links with a severe bandwidth reduction. Therefore, we set the capacity of specific core links to 0.01 Mbps between switches 1 and 1 and between switches 4 and 4, as shown in Fig. 2. The bandwidth failure threshold in Oddlab was assumed to simplify the faulty link simulation, and the algorithm could be adjusted manually to any required bandwidth severity. Additionally, the same experiment was repeated on real workloads in an asymmetric DCN topology with random traffic patterns. We set the link fault threshold iteration ($fault\_iter$) to be larger than one.

Consequently, the faulty links were expected to maintain an identical link faulty threshold of $health\_thr = 0.01$ Mbps for at least two monitoring periods in a row when $Odds > 1$. After detecting the faulty links, all the active elephant flows on the upstream side of the aggregate switch were rescheduled to other healthy paths. Thus, all incoming flows could still obtain symmetric paths of the DCN topology by avoiding faulty links. The remaining flows at the faulty link were mice flows (i.e., flow size < 50 KB). When the link productivity increases the throughput to the slow start of a TCP flow, it will be excluded from the list of affected links (i.e., false-positive cases). However, when the link remains at the same productivity rate (long-lived failure) with $Odds > 1$ for further monitoring periods, then the upstream hashing weight will be altered accordingly. The evaluation of this experiment was based on the obtained average FCT values for redirected elephant flows after redirection. This experiment was conducted in five runs because different numbers and sizes of elephant flows were randomly scheduled on the faulty links based on the edge switch flow sampling and adaptive flow scheduling decisions. During the experiment, we found that only 97 elephant flows were found on the faulty links and recovered based on the adaptive flow scheduling. Figure 12 shows the average FCT for the rescheduled elephant flows found

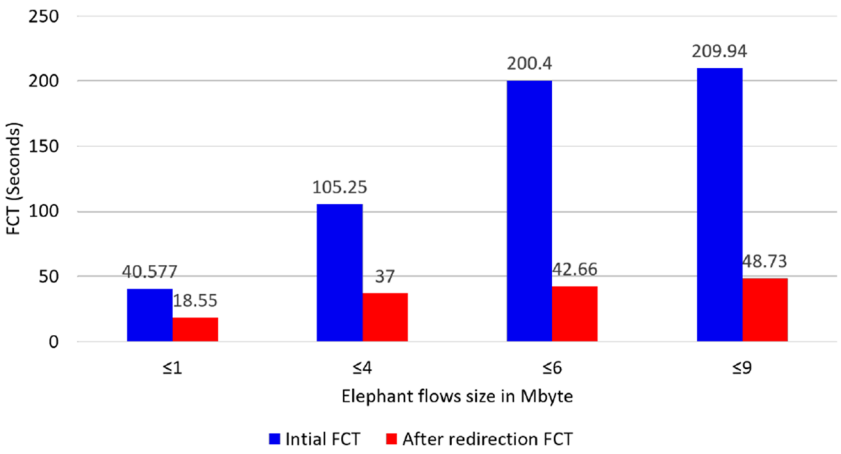

Fig. 12 Average FCT for the redirected elephant flows after failure detection

on the detected faulty links. The initial FCT refers to the primary determinate FCT value on the end-host application layer for each flow over the faulty links. We found unusual initial outlier FCT values (i.e., more significant than 1,000 s) for three elephant flows with a size larger than 6 MB. Therefore, we omitted them to obtain more precise average results. As depicted in Fig. 12, Oddlab considerably reduces the average FCT for the redirected elephant flows over the initial determined FCT owing to detecting the faulty links and adaptive flow scheduling. After detecting the faulty links, the Oddlab adaptive scheduling model spreads the flows across the available paths without significant congestion in the paths, depending on the proposed adaptive scheduling, as discussed in Section 4.4.2. Hence, the new average FCT for redirected elephant flows is acceptable considering their sizes.

7.4 Discussion and comparison of faulty link detection strategies

In specific TE methods, such as Hedera [2], Mahout [5], and Sieve [13], the elephant flows are rescheduled only when a certain threshold is reached or based on the availability of sufficient bandwidth on other paths. For instance, Hedera adopted Portland specialized mechanisms [36] for link failure detection and flow rerouting to manage the faulty link issue.

DCN end-hosts can indicate early elephant flows using the initially determined FCT at the cost of altering all end-hosts to monitor each flow and participating in routing decisions, such as SAPS [24], Hermes [22], and FlowFurl [26]. Furthermore, end-host sensing does not provide sufficient information about the affected DCN links; therefore, network administrators cannot efficiently explore the problem. Consequently, the experimental results reveal Oddlab feasibility and advantages using the SDN controller and OpenFlow protocol.

Regarding the detection of faulty links inside a DCN topology, Gill et al. [3] correlated link failure logs with the earlier observed traffic link in a 5-min time window. Although this type of correlation requires a link-state memory for each DCN link defined in each monitoring time, the link should not fail because of a preceding traffic decline. For instance, the current state of the link could be in a typical traffic situation because of a routine service reduction from the end-hosts. Therefore, multiple false-positive detections may occur. Moreover, the correlation process of Oddlab occurred with current and upcoming events in a shorter time (2 s of the information polling rate). Hence, when the correlation breaks in an upcoming event, the link is identified as a false-positive detection and returns to normal.

In our tests, we may achieve some false-positive results, particularly when the simulation is near the end and no more traffic exists; thus, Oddlab cannot delete the false-positive detections. The faulty link detection sensitivity (the true positive rate) was 100% accurate in detecting faulty links. The true negative rate of Oddlab was 92.85% specific in identifying the correct faulty links (i.e., one false-positive link out of 14 normal links).

Although we adopted ECMP paths to overcome the controller overhead, we encountered the dilemma of not identifying the precise number of flows that the edge switch handles. Therefore, we counted the port consumption in edge labeling. However, faulty link detection by Oddlab is based on the following observation: “As far as most edge switches are labeled based on the odds concept, the faulty paths can be detected quickly”. Consequently, the expected time for identifying faulty links relies on the network traffic demands and monitoring time to achieve a high detection precision.

8 Conclusion and future studies

This study presented Oddlab, a novel hybrid and deployable load-balancing approach that guarantees the QoS of multi-rooted DCN traffic in symmetric and asymmetric topologies, including faulty link detection. We employed proactive and adaptive scheduling for flow scheduling that considers the healthy paths, available bandwidth, and active elephant flows to determine the best paths and avoid significant flow rescheduling. The procedure of identifying faulty links relies on correlating two events within the DCN: loaded edge switches and underutilized core links. Therefore, Oddlab employs the statistics of the DCN switches on a single and central SDN controller to detect faulty links and achieves promising results in both the symmetric and asymmetric topologies. However, extensive experiments were conducted on a wide range of traffic patterns with synthetic and realistic workloads to prove the feasibility of Oddlab without altering any network component, including hosts or switches. The results indicated that Oddlab significantly improves the bisection bandwidth and link utilization with an overall average FCT reduction of up to 30, 25.7, 62, and 5% compared to that of ECMP, Hedera, PureSDN, and Sieve, respectively. We demonstrated that Oddlab functions with considerably low complexity and low computational overhead on the SDN controller. Therefore, Oddlab has the potential to be applied to commercial DCNs at a low cost.

Nevertheless, further investigation is needed to examine the effectiveness of the proposed method on different DCN topologies, including multi-stage topologies such as

the Clos network. In addition, the proposed faulty link detection strategy functions only on a three-stage topology and beyond, including the $k-4$ fat-tree topology. Hence, a possible approach to overcome this issue is to correlate other events, such as queue lengths of the paths.

Acknowledgements Aymen Hasan Alawadi would like to thank the University of Kufa – Iraq, the Tempus Public Foundation (TPF) – Stipendium Hungaricum program, and the Department of Telecommunication and Media Informatics in Budapest University of Technology and Economics- Hungary for supporting his Ph.D. scholarship.

Funding Open access funding provided by Budapest University of Technology and Economics. 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. Hopps C (2000) Analysis of an equal-cost multipath algorithm (No. RFC 2992). https://doi.org/10.17487/RFC2992
2. Al-Fares M, Radhakrishnan S, Raghavan B, Huang N, Vahdat A (2010) Hedera: Dynamic Flow scheduling for datacenter networks. In: Nsdi (vol 10, No 2010. https://doi.org/10.5555/1855711.1855730
3. Gill P, Jain N, Nagappan N (2011) Understanding network failures in data centers: Measurement, analysis, and implications. In: Proceedings of the ACM SIGCOMM 2011 Conference, pp 350–361. https://doi.org/10.1145/2018436.2018477
4. McKeown N, Anderson T, Balakrishnan H, Parulkar G, Peterson L, Rexford J, Shenker S, Turner J (2008) OpenFlow: Enabling innovation in campus networks. ACM SIGCOMM Computer Communication Review 38(2):69–74. https://doi.org/10.1145/1355734.1355746
5. Curtis AR, Kim W, Yalagandula P (2011) Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection. In: Infocom, vol 11, pp 1629–1637, https://doi.org/10.1109/INFCOM.2011.5934956
6. Wang CA, Hu B, Chen S, Li D, Liu B (2017) A switch migration-based decision-making scheme for balancing load in SDN. IEEE Access 5:4537–4544. https://doi.org/10.1109/ACCESS.2017.2684188
7. Erickson D (2013) The Beacon OpenFlow controller. In: Proceedings of the second ACM SIGCOMM workshop on hot topics in software defined networking, pp 13–18. https://doi.org/10.1145/2491185.2491189
8. Zhao G, Xu H, Fan J, Huang L, Qiao C (2020) Achieving fine-grained flow management through hybrid rule placement in SDNs. IEEE Transactions on Parallel and Distributed Systems 32(3):728–742. https://doi.org/10.1109/TPDS.2020.3030630
9. Benson T, Akella A, Maltz DA (2010) Network traffic characteristics of data centers in the wild. In: Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement, pp 267–280. https://doi.org/10.1145/1879141.1879175
10. Tang F., Zhang H., Yang LT, Chen L (2019) Elephant flow detection and differentiated scheduling with efficient sampling and classification. IEEE Transactions on Cloud Computing. https://doi.org/10.1109/TCC.2019.2901669
11. Irteza SM, Bashir HM, Anwar T, Qazi IA, Dogar FR (2018) Efficient load balancing over asymmetric datacenter topologies. Comput Commun 127:1–12. https://doi.org/10.1016/j.comcom.2018.05.010
12. Roy A, Zeng H, Bagga J, Porter G, Snoeren AC (2015) Inside the social network (datacenter) network. In: Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication. pp 123–137. https://doi.org/10.1145/2785956
13. Zaher M, Alawadi AH, Molnár S (2021) Sieve: Flow scheduling framework in SDN-based data center networks. Comput Commun 171:99–111. https://doi.org/10.1016/j.comcom.2021.02.013
14. Zaher M, Alawadi AH, Molnár S (2020) Class-based flow-scheduling framework in SDN-based data center networks. In: 2020 International Conference on Computing, Electronics & Communications Engineering (iCCECE), IEEE, pp 51–56. https://doi.org/10.1109/iCCECE49321.2020.9231052
15. Machi H (2017) Research on data center network traffic scheduling strategy based on SDN, Chongqing university of posts and telecommunications, Chongqing, China. https://wap.cnki.net/touch/web/Dissertation/Article/10617-1018972647.nh.html
16. Al-Fares M, Loukissas A, Vahdat A (2008) A scalable, commodity data center network architecture, vol 38
17. Onogi F, Kasuga H, Shinomiya N (2020) On approximate approaches the unsplittable flow edge load factor balancing problem. In: 2020 35th International Technical Conference on Circuits/Systems, Computers and Communications (ITC-CSCC), IEEE, pp 73–77. https://doi.org/10.5555/795663.796365
18. Curtis AR, Mogul JC, Tourrilhes J, Yalagandula P, Sharma P, Banerjee S (2011) DevoFlow: Scaling flow management for high-performance networks. In: ACM SIGCOMM Computer Communication Review, Vol 41, No 4, ACM, pp 254–265. https://doi.org/10.1145/2018436.2018466
19. Liu L, Jiang Y, Shen G, Li Q, Lin D, Li L, Wang Y (2019) SDN-based hybrid strategy for load balancing in data center networks. In: 2019 IEEE Symposium on Computers and Communications (ISCC), IEEE, pp 1–6. https://doi.org/10.1109/ISCC47284.2019.8969673
20. Alizadeh M, Edsall T, Dharmapurikar S, Vaidyanathan R, Chu K, Fingerhut A, Varghese G (2014) CONGA: Distributed congestion-aware load balancing for datacenters. In: Proceedings of the 2014 ACM Conference on SIGCOMM, pp 503–514. https://doi.org/10.1145/2740070.2626316
21. Levi C, Segal M (2021) Avoiding bottlenecks in networks by short paths. In Telecommun Syst 76(4):491–503. 10.1007/s11235-020-00720-7
22. Zhang H, Zhang J, Bai W, Chen K, Chowdhury M (2017) Resilient datacenter load balancing in the wild. In: Proceedings of the Conference of the ACM Special Interest Group on Data Communication, pp 253–266. https://doi.org/10.1145/3098822.3098841
23. Hu J, Huang J, Lv W, Zhou Y, Wang J, He T (2019) CAPS: Coding-based adaptive packet spraying to reduce flow completion time in data center. IEEE/ACM Trans Netw 27(6):2338–2353. https://doi.org/10.1109/TNET.2019.2945863

24. Irteza SM, Bashir HM, Anwar T, Qazi IA, Dogar FR (2018) Efficient load balancing over asymmetric datacenter topologies. Comput Commun 127:1–12. https://doi.org/10.1016/j.comcom.2018.05.010
25. Ghorbani S, Yang Z, Godfrey P, Ganjali Y, Firoozshahian A (2017) DRILL: Micro load balancing for low-latency data center networks. In: Proceedings of the Conference of the ACM Special Interest Group on Data Communication, ACM, pp 225–238
26. Sharma K, Yadav RN (2020) An adaptive, fault tolerant, flow-level routing scheme for data center networks. Computer Networks 175:107235. https://doi.org/10.1016/j.comnet.2020.107235
27. Mitzenmacher M (2001) The power of two choices in randomized load balancing. IEEE Transactions on Parallel and Distributed Systems 12(10):1094–1104. https://doi.org/10.1109/71.963420
28. Liu Y, Zhang J, Li W, Wu Q, Li P (2021) Load balancing-oriented predictive routing algorithm for data center networks. Future Internet 13(2):54. https://doi.org/10.3390/fi13020054
29. Al Awadi AHR (2017) Dual-layer SDN model for deploying and securing network forensic in distributed data center. Current Journal of Applied Science and Technology, pp 1–11. https://doi.org/10.9734/CJAST/2017/34752
30. Ryu: Ryu SDN Framework. https://ryu-sdn.org (Accessed 22 Mar. 2021)
31. Mininet: A realistic virtual network. http://mininet.org (Accessed 22 Mar. 2021)
32. Al-Fares M, Loukissas A, Vahdat A (2008) A scalable, commodity data center network architecture. ACM SIGCOMM Comput Commun Rev 38(4):63–74. https://doi.org/10.1145/1402946.1402967
33. iPerf - The ultimate speed test tool for TCP, UDP, and SCTP. https://iperf.fr (Accessed 4 Apr. 2021)
34. BWM-ng - Bandwidth Monitor NG (Next Generation). https://www.gnutoolbox.com/bwmng (Accessed 4 Apr. 2021)
35. Alizadeh M, Greenberg A, Maltz DA, Padhye J, Patel P, Prabhakar B, Sengupta S, Sridharan M (2010) Data center tcp (dctcp). In: Proceedings of the ACM SIGCOMM 2010 Conference, pp 63–74. https://doi.org/10.1145/1851275.1851192
36. Niranjan Mysore R, Pamboris A, Farrington N, Huang N, Miri P, Radhakrishnan S, Subramanya V, Vahdat A (2009) Portland: a scalable fault-tolerant layer 2 data center network fabric. In: Proceedings of the ACM SIGCOMM 2009 conference on Data communication, pp 39–50. https://doi.org/10.1145/1594977.1592575