An Efficient Route Management Framework for Load Balance and Overhead Reduction in SDN-based Data Center Networks

You-Chiun Wang and Siang-Yu You

Abstract—A data center network (DCN) is composed of many servers interconnected by well-organized switches, which involves massive amount of data transmissions to provide cloud services. It is critical to manage packet routes in DCNs to avoid congestion on some links. The software-defined networking (SDN) technique supports auto-configuration of switches by a central controller, and gives an easy way to manage traffic flows. In the paper, we exploit SDN to improve performance of fat-tree DCNs, and propose a *low-cost, load-balanced route management (L2RM) framework*. L2RM keeps monitoring network traffics and computes a load-deviation parameter to check if some links of switches are burden with heavy loads. Then, an *adaptive route modification (ARM) mechanism* is triggered if necessary, which considers the size limitation of flow tables and uses group tables in OpenFlow to distribute flows among different links to balance their loads. Besides, L2RM uses a *dynamic information polling (DIP) mechanism* to query switches about their statuses, so as to reduce the message overhead of the controller. Through Mininet simulations, we show that our L2RM framework can better increase link utilization, alleviate table overflow, and reduce message cost, as comparing with other SDN-based approaches for fat-tree DCNs.

Index Terms—data center network (DCN), fat tree, load balance, OpenFlow, software-defined networking (SDN).

1 Introduction

The boom in the service of cloud computing and big data results in a corresponding boom in the deployment of *data center networks (DCNs)* [1]. In a DCN, numerous servers are connected by many switches to form a huge intranet or local area network. To support scalability, switches need to be organized, so different network topologies such as fat tree, virtual layer 2 (VL2), and jellyfish are proposed [2].

To handle the growing demand of cloud computing, it is critical to efficiently route traffic flows among servers. Traditional switches possess both control and data planes, so they decide where to forward packets on their own. However, as network scale substantially increases, popular routing methods like the shortest-path routing may not work well. In particular, some links of switches could become "hotspots", which will be inevitably congested by a large amount of data, causing significant degradation in DCN performance [3].

The technique of *software-defined networking (SDN)* provides a new paradigm to manage the network by centralizing the control plane in a *controller* and distributing the data plane among switches [4]. In this way, the controller is capable of managing traffic flows by setting transmission rules in switches to command their behavior in terms of packet processing. In addition, administrators can easily manage the whole network with the help of the controller, rather than doing configuration on every switch. Due to its flexibility, there have been various SDN applications developed, from routing multicast flows [5] to providing anonymous communications [6], isolating rogue access points [7], and eliminating redundant packets [8].

In this paper, we aim to improve performance of fat-tree DCNs by dynamically rerouting traffic flows to avoid link congestion. Based on SDN, we propose a *low-cost, load-balanced route management (L2RM) framework* to help the controller get network status by using three tables to store information of paths, loads, and switches. According to the traffic loads of links, L2RM computes a *load-deviation parameter* to check whether there is potential congestion. If so, the *adaptive route modification (ARM) mechanism* is invoked to reroute packets via alternative paths and share the loads of congested links. To save the controller's overhead, a *dynamic information polling (DIP) mechanism* is also developed to adjust the time period for the controller to query switches about their information.

Our L2RM framework has three features. First, the ARM mechanism is invoked only if necessary by using a dynamic load-deviation parameter. Thus, we not only save the cost to run the ARM mechanism but also keep the network stable (as the routes of traffic flows will not change frequently). Second, the ARM mechanism helps switches remove old entries from their flow tables to avoid buffer overflow. Moreover, it amends routes of traffic flows by varying bucket weights in group tables defined by OpenFlow [9]. The design is lightweight and incurs a low cost. Third, our DIP mechanism applies an exponential backoff idea to adjust the query period. In this way, we can reduce the controller's overhead while keeping correct information of switches. These features distinguish the L2RM framework from existing solutions and give our contributions. Simulation results verify that L2RM outperforms other SDN-based methods developed for fat-tree DCNs, including round robin, LABERIO, and DLPO, in terms of link utilization, table overflow, and message overhead.

This paper is outlined as follows: Section 2 gives background knowledge, and Section 3 surveys related work. We present the L2RM framework in Section 4, followed by its performance evaluation in Section 5. Then, a conclusion will be drawn in Section 6.

The authors are with the Department of Computer Science and Engineering, National Sun Yat-sen University, Kaohsiung, 80424, Taiwan. Email: ycwang@cse.nsysu.edu.tw; m043040004@student.nsysu.edu.tw.

TABLE 1: Example of a flow table (we omit some fields).

match fields	priority	instructions
in_port=1	priority=1	actions=output:3
eth_type=0x0800, ipv4_src="10.0.0.1"	priority=1	actions=flood
eth_type=0x0800, ipv4_dst="10.0.0.2"	priority=1	actions=group:87
	priority=0	actions=controller:6633

TABLE 2: Example of a group table (we omit some fields).

group identifier	group type	action buckets
group_id=87	select	bucket=weight:30, actions=output:3 bucket=weight:70, actions=output:4

2 PRELIMINARY

2.1 SDN Implementation: OpenFlow

OpenFlow is a popular protocol to implement SDN, which is regulated by ONF (open networking foundation) [10]. It defines the communication interface between the data and control planes in SDN. Specifically, switches (i.e., data plane) consult their *flow tables* and *group tables* to decide how to process packets. The controller (i.e., control plane) establishes a TLS (transport layer security) connection with each switch to install rules in its tables. In this way, the controller is capable of commanding the transmission behavior of the switch.

A flow table is composed of one or more *flow entries*, where each flow entry has the following fields:

- Match fields: The switch uses it to check whether a packet satisfies the conditions specified by the flow entry.
- Priority: When a packet meets the conditions of multiple flow entries, the switch can select the flow entry with the highest priority to process that packet.
- Counters: It stores the number of packets (and their bytes) that the switch has processed by using the flow entry.
- Instructions: Once a packet satisfies the conditions given in the match fields, the switch will conduct the operations indicated in this field to process that packet.
- Timeouts: It has *idle-timeout* and *hard-timeout* sub-fields. The flow entry will be removed only when it is not used during idle-timeout. Hard-timeout gives the lifetime of the flow entry, so it must be discarded after timeout.
- Cookie: It is an opaque value set by the controller to filter flow statistics, modification, or deletion. Thus, the switch will not use the cookie field to process packets.

Table 1 gives an example. The 1st flow entry means that a packet coming from port 1 should be forwarded to port 3. The 2nd flow entry tells the switch to broadcast a packet to all its ports if the packet has source IP address of 10.0.0.1. The 3rd flow entry asks the switch to handle a packet by the group table with ID 87 if the packet's destination IP address is 10.0.0.2. The 4th flow entry has no match fields, which means that if none of the above flow entries can handle the packet, it should be sent to the controller whose port is 6633. We remark that the last flow entry will trigger an event of *table miss*, which makes the switch send a *Packet_In message* to the controller that contains the packet's information. In this case, the controller will generate a new flow entry (e.g., by algorithms or administrators), and install the flow entry in the switch by replying a *Packet_Out message* to it.

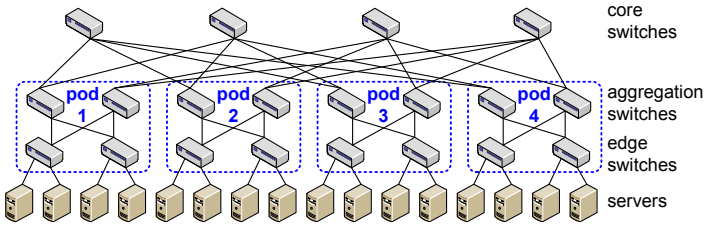


Fig. 1: A 4-ary fat-tree DCN.

A group table can be viewed as an extension of the flow table. It consists of *group entries*, each with four fields:

- Group identifier: It is a 32-bit unsigned integer used to uniquely identify each group.
- Group type: If the value is "all", the switch conducts each action in the group. When the value is "select", the switch picks some actions to conduct based on their weights.
- Counters: It increases whenever a packet is handled by the group entry.
- Action buckets: Each bucket contains a set of actions (and parameters) that tell the switch how to handle the packet.

Table 2 shows an example. Recall that the 3rd flow entry in Table 1 asks the switch referring to Table 2 (with group_id=87) to handle the packet. Since the value of group type is "select", the switch picks one bucket of actions based on its weight. In the example, the switch will forward the packet to its ports 3 and 4 with probabilities of 0.3 and 0.7, respectively.

2.2 DCN Topology: Fat Tree

Fat tree is a classic and common DCN topology [2], which organizes switches into a tree-like structure. In general, a k-ary fat tree can connect at most $k^3/4$ servers by using $5k^2/4$ switches, where k is the number of ports in a switch. In a fat-tree topology, switches are divided into three layers, called *core*, *aggregation*, and *edge*. Edge switches directly connect with servers. They are also grouped with aggregation switches into k pods, where each pod has $k/2$ edge switches and $k/2$ aggregation switches. Thus, each edge switch uses $k/2$ ports to connect with servers, and the residual ports are connected with aggregation switches. Besides, the number of core switches is $k^2/4$, where a core switch uses one port to connect with each pod. Fig. 1 gives an example, where $k = 4$. In a fat tree, we can use the same type of switches (usually cheap commodity products) to save the hardware cost. Moreover, since there are multiple paths with an equal length between any two servers, the fat-tree topology can support full bisection bandwidth in theory [11]. However, this topology requires more switches to maintain the tree structure, so it is economical and efficient to adopt SDN to manage these switches.

3 RELATED WORK

In the literature, some studies are dedicated to the issue of load balance in DCNs. We classify them into two categories. One aims to balance working loads of servers, while the other seeks to balance traffic loads in the network.

3.1 Load Balance on Servers

As a DCN usually consists of identical servers, it is natural to distribute jobs among them to avoid burdening some servers with heavy loads. Load-balancing methods can be either *static* or *dynamic* [12]. Static methods assign jobs to different servers according to predefined rules, so they are easy to use but may not give quick response to the change of network condition. Dynamic methods consider the working load of each server and can better support load balance. Lee et al. [13] survey three common load-balancing methods, including *round robin (RR)*, *least connection (LC)*, and *shortest expected delay (SED)*. RR is static, which asks servers to take jobs by turns. Both LC and SED are dynamic, which select the server that has the fewest client connections and spends the shortest working time to do the job, respectively.

Some studies adopt SDN to achieve load balance on servers. For example, [14] analyzes packets sent from clients by the controller. Both RR and LC methods are then used to share working loads among servers. However, [14] uses SDN merely to collect the information of packets. The work [15] assigns jobs to servers by referring to IP addresses of clients. When a server becomes overloaded, the controller modifies its flow entries to dispatch traffic flows to other servers. On the other hand, [16] employs multipath TCP along with SDN to save transmission time of traffic flows by allowing simultaneous use of multiple paths in a fat-tree DCN. Thus, some network devices will have longer idle time and can be put to sleep to reduce energy consumption. Obviously, these studies have different objectives with our paper.

3.2 Load Balance on Network

The work [17] considers scheduling a mix of traffic flows with and without deadlines in a DCN. To prioritize deadline flows while avoiding starving non-deadline flows, a minimal-impact congestion control protocol is used to transmit deadline flows by spending the least amount of bandwidth. Then, non-deadline flows are scheduled with the objective of decreasing their flow competition time. Bai et al. [18] apply the technique of *explicit congestion notification (ENC)* to DCNs to improve throughput and reduce latency. However, previous ENC methods assume that each switch port has only one queue. Thus, they propose an ENC solution for multi-service DCNs, where there are multiple service queues associated with each port to isolate different traffic classes. The study [19] aims to support micro load balance for DCNs by letting each switch make its decision without a central controller. To do so, a set of candidate next-hops for each destination should be installed in the forwarding table of every switch. Then, a switch locally decides where to route each packet and seeks to balance loads of its ports. In [20], a small-scale DCN is considered, where each top-of-rack (ToR) switch connects with the rest components in the DCN (e.g., aggregation switches or servers). To balance traffic loads, it reconfigures the network topology to enable ToR switches with heavy loads to use under-utilized uplinks from their neighbors. However, the above studies do not exploit SDN to improve DCN performance.

A number of research efforts adopt SDN to manage traffic flows in DCNs. Assuming that switches form a mesh topology, [21] assigns a cost for each link for traffic routing. When a link has lower utilization, VoIP flows are given with a lower cost on that link. Then, the Dijkstra's algorithm [22] is used to find the shortest route for each flow. Thus, VoIP flows can have precedence over others to get network bandwidth. The work [23] uses depth-first search to find multiple routes from the source to the destination in a DCN with the Abilene topology [24]. Then, the route which offers the maximum bandwidth is selected to send data. In addition, when the total utilization of bandwidth is above 80%, new traffic requests will be denied to avoid network congestion.

How to manage routes in fat-tree DCNs is also discussed. The work [25] aims at *elephant flows*, each carrying a great deal of data. The controller keeps monitoring link utilization and splits these flows by sending their packets via different paths. However, [25] uses SDN to adjust packet routes based on the existence of elephant flows, instead of network status. Adami et al. [26] classify traffic flows into bronze, silver, and gold. For a bronze flow, the controller recalculates a new path only when route failure occurs. For a silver flow, the controller finds its working and recovery paths, where the former is used to send data and the latter is used only when the working path is broken. Then, a gold flow can send its data via both working and recovery paths. However, [26] addresses how to recover routes from failure, not to adjust routes to balance their loads.

In [27], a *load-balanced routing with OpenFlow (LABERIO)* method is proposed to manage traffic flows. When the network utilization exceeds a threshold, LABERIO iteratively selects the flow using the most bandwidth from the busiest link, and reroutes that flow to another path. However, it does not make good use of group tables to distribute flows among different paths. Besides, LABERIO may be frequently invoked due to *impulse flows* (i.e., the flows which last in a short time but have volumes of data), which wastes the controller's resource to do the load-balancing job. To cope with the problem, [28] develops a *dynamic load-balanced path optimization (DLPO)* method and uses the technique of simple moving average [29] to reduce the triggering frequency of the DLPO method. In DLPO, the top 10% of busiest links are selected and their flows are transferred to other links with lower utilization for load balance. Nevertheless, [28] does not consider saving the controller's overhead caused by SDN messages. Since both [27] and [28] address the load-balancing issue in fat-tree DCNs by using SDN, we will compare our L2RM framework with LABERIO and DLPO in Section 5. Simulation results will show that L2RM outperforms them in terms of load balance, table usage, and message overhead.

4 The Proposed L2RM Framework

Fig. 2 presents the system architecture of our L2RM framework, which is installed in the controller to manage a fat-tree DCN. It maintains *path*, *load*, and *entry* tables to monitor the statuses of routes, traffic loads, and flow entries in relation to each switch, respectively. Since some switches may be broken (e.g., hardware failure or disconnection), L2RM uses a *switch health check (SHC) mechanism* to check whether switches are alive and remove invalid entries of failed switches from these tables. Then, the controller periodically estimates the load-deviation parameter L_D by referring to the load table. In case that the number of times that L_D exceeds a threshold δ, the ARM mechanism is invoked to arrange routes for traffic flows. In addition, the controller can use the DIP mechanism to poll switches to update the three tables. Below, we detail our design in each component.

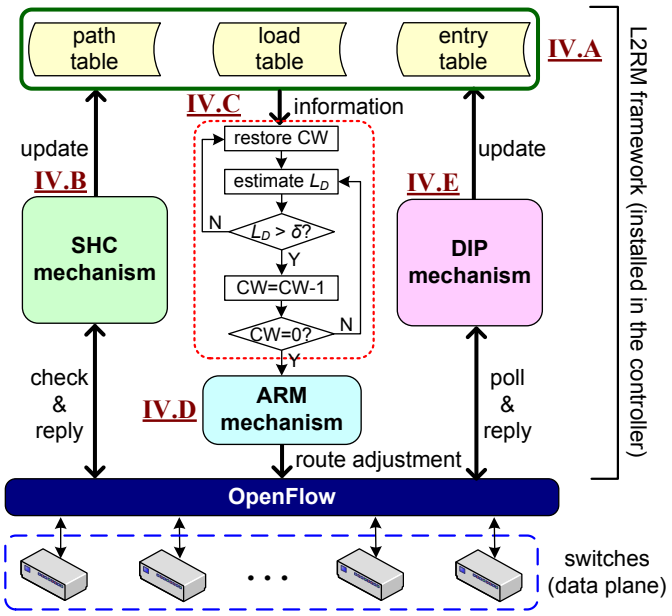


Fig. 2: Architecture of our L2RM framework, where underlined texts indicate the sections that discuss the detailed designs of the corresponding components.

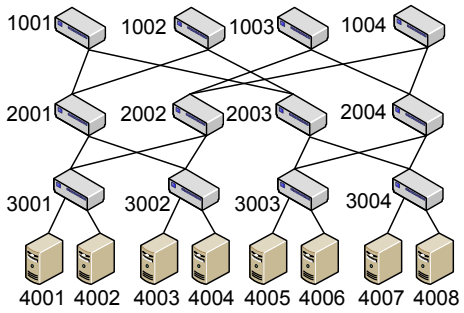


Fig. 3: Example of a DCN topology.

TABLE 3: Example of the path table (we omit some records).

(source, destination)	routes
(4001, 4003)	{3001, [2001, 2002], 3002}
(4002, 4005)	{3001, 2001, [1001, 1002], 2003, 3003}, {3001, 2002, [1003, 1004], 2004, 3003}
(4003, 4007)	{3002, 2001, [1001, 1002], 2003, 3004}, {3002, 2002, [1003, 1004], 2004, 3004}

4.1 Table Maintenance

As mentioned earlier, the controller keeps three tables to get network status efficiently. Specifically, the path table gives the shortest routes between each pair of source and destination servers. To condense this table, we use a notation '[]' to represent alternative switches in the same layer. Table 3 gives an example, where we consider the network topology in Fig. 3. The 1st record in Table 3 indicates that there are two shortest paths between servers 4001 and 4003. One is through switches 3001→2001→3002 and the other is through switches 3001→2002→3002. These two paths include switches 3001 and 3002, but they choose different switches in the aggregation layer. Therefore, we use a term {3001, [2001, 2002], 3002} to represent both paths, which saves table size.

The controller can consult the path table to select alternative paths to balance traffic loads of links by the ARM mechanism, which will be discussed in Section 4.4. Besides, it can use the SHC mechanism in Section 4.2 to remove the invalid records (due to switch failure) from the path table. Theorem 1 analyzes the size of the path table.

TABLE 4: Example of the load table (we omit some records).

switch ID	traffic amount
2001	{1:3283, 2:218748, 3:1853, 4:3158}
2003	{1:3480, 2:1351, 3:288536, 4:2195}
3001	{1:597811, 2:1421, 3:878787, 4:585}

Theorem 1. *Let n and k be the number of servers and the number of ports in a switch, respectively. When switches are arranged in a k-ary fat tree, the path table will contain no more than $(-1-\frac{k}{2}+\frac{5k^2}{4}+\frac{k^3}{8}+\frac{k^4}{32})\cdot n$ switches in its records.*

Proof: There are three cases to be discussed.

Case I: Both source and destination servers are connected by the same edge switch. In this case, the path contains only the edge switch. According to Section 2.2, an edge switch connects at most $\frac{k}{2}$ servers. Thus, each server can choose $(\frac{k}{2}-1)$ destinations. Since we have n servers, there will be at most $n(\frac{k}{2}-1)$ switches stored in the records for case I.

Case II: Both source and destination servers are connected only by the same aggregation switch. Observing from the example of (4001,4003) pair in Table 3, each record of paths contains two edge switches (which directly link to the source and destination servers) and $\frac{k}{2}$ alternative aggregation servers. Thus, a record must have $(\frac{k}{2}+2)$ switches. In case II, the source server can choose the destination servers connected by $(\frac{k}{2}-1)$ aggregation switches in the same pod, where each aggregation switch further connects with $\frac{k}{2}$ servers. In other words, we have totally $\frac{k}{2}(\frac{k}{2}-1)$ choices of destination servers for each source server. Consequently, we need to store at most $n\cdot\frac{k}{2}(\frac{k}{2}-1)(\frac{k}{2}+2)$ switches in the records for case II.

Case III: Both source and destination servers are connected only by the same core switch. A k-ary fat tree has no more than $\frac{k^2}{4}$ core switches, each connecting to an aggregation switch in one pod. Thus, a record of paths will contain two edge switches, two aggregation switches, and $\frac{k^2}{8}$ alterative core switches. Observing from the example of (4002, 4005) pair in Table 3, each source server can select $\frac{k}{2}$ aggregation servers to connect with the core switch. Also, each destination server can find out $\frac{k}{2}$ aggregation servers to connect with the same core switch. Therefore, we need to keep at most $(\frac{k}{2}\cdot\frac{k}{2})$ records for each pair of source and destination servers. Since there are n servers in the DCN, we will have no more than $n\cdot\frac{k^2}{4}(\frac{k^2}{8}+4)$ switches in the records for case III.

By combining the results of these cases, we will store at most $n(\frac{k}{2}-1)+n\cdot\frac{k}{2}(\frac{k}{2}-1)(\frac{k}{2}+2)+n\cdot\frac{k^2}{4}(\frac{k^2}{8}+4)=(-1-\frac{k}{2}+\frac{5k^2}{4}+\frac{k^3}{8}+\frac{k^4}{32})\cdot n$ switches in all records of the path table, which proves this theorem. □

Then, the load table stores the amount of traffic load of each switch (in terms of its ports) collected in the previous T_{poll} duration, where T_{poll} is an adjustable parameter decided by the DIP mechanism in Section 4.5. Each record of the load table has the format of (switch ID, {port 1: traffic amount 1, port 2: traffic amount 2, ⋯, port k: traffic amount k}), where the amount of traffic is measured in bytes. The controller queries each switch about its information by DIP and updates the load table accordingly. Table 4 gives an example. The controller also uses the load table to determine whether to invoke the ARM mechanism, as discussed in Section 4.3.

The entry table keeps track of the utilization of the flow table owned by each switch. Its record has the format of (switch ID, the number of used flow entries, the size of the flow table). For example, a record (1001, 18, 20) means that switch 1001 has stored 18 flow entries in its flow table, whose maximum size is 20 flow entries. The controller can refer to the entry table to monitor the usage of flow tables (to prevent them from overflowing) and update its information by the DIP mechanism. Note that every flow table must contain one default flow entry, which is shown in the last entry of Table 1. The default entry allows the switch to notify the controller of the new packet that the switch has no rules to handle it, so the controller can decide how to process that packet by installing new entries in the switch's flow table. Theorem 2 analyzes the sizes of both load and entry tables.

Theorem 2. *Given a k-ary fat-tree DCN, the maximum sizes of load and entry tables will be $5k^3/4$ and $5k^2/4$, respectively.*

Proof: As discussed in Section 2.2, a k-ary fat tree has at most $5k^2/4$ switches, where k is also the number of ports per switch. In the load table, since we need to record the amount of traffic passed through each port of every switch, the maximum size of the load table will thus be $5k^3/4$. On the other hand, each switch has one flow table, so the maximum size of the entry table will be $5k^2/4$. □

We remark that the analysis of both Theorems 1 and 2 hold only for fat-tree DCNs.

4.2 The SHC Mechanism

The SHC mechanism follows the OpenFlow protocol for the controller to check the health of each switch. In particular, a switch has four states with respect to its connection with the controller. In the beginning, the switch starts from the *HANDSHAKE_DISPATCHER* state to get in touch with the controller. Then, it changes to the *CONFIG_DISPATCHER* state to negotiate parameters with the controller. After that, the switch enters the *MAIN_DISPATCHER* state. In this state, the controller can get the messages of *switch feature* to capture the network topology (and update the path table accordingly), and also transmit *set-config* messages to command switches. However, in case of switch failure (e.g., shut down or network disconnected), the controller will mark the switch's state as *DEAD_DISPATCHER* and trigger an event of *EventOfPState-Change* to indicate some failure in the network equipment.

Therefore, once the controller receives the EventOfPState-Change event in regard to a switch s_i, it searches all records in the path table and removes s_i from existing paths. When a path becomes partitioned due to the removal of s_i, the path is discarded accordingly. Besides, the records that contain s_i in both load and entry tables will be also marked as *invalid*.

4.3 Moment to Invoke The ARM Mechanism

To update the load table, the controller uses the function of *OFPPortStatsRequest* to query each switch about the amount of traffic passing through it. Then, the switch will reply the number of received packets, the amount of received data (in bytes), the number of transmitted packets, and the amount of transmitted data (in bytes) by each of its ports. However, since every port p_i of a switch will connect to at most one port p_j of another switch, it means that the amount of data transmitted by port p_i must be equal to the amount of data received by port p_j. Therefore, we store only the information of transmitted data in the load table.

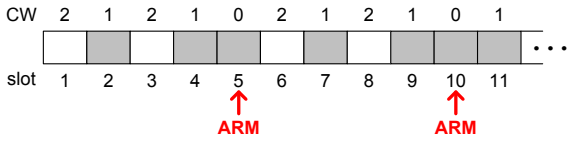


Fig. 4: Example of using the CW counter to decide when to invoke the ARM mechanism, where gray slots mean that the condition of $L_D > \delta$ holds.

Based on the amount of data transmitted by each port, we define the traffic load of a link $l_{i,j}$ (which connects ports p_i and p_j) at time t as follows:

$$\zeta_{i,j}(t) = \frac{A_{i,j}(t)}{C_{i,j}(t)}, \tag{1}$$

where $A_{i,j}(t)$ is the amount of data transmitted through $l_{i,j}$ and $C_{i,j}(t)$ is the maximum capacity of $l_{i,j}$ at time t. The units of both $A_{i,j}(t)$ and $C_{i,j}(t)$ in Eq. 1 are bits, so $\zeta_{i,j}(t)$ will be a rational number without any unit. In particular, we have $0 \leq \zeta_{i,j}(t) \leq 1$, because a link never carries data more than its capacity. Let $\mathcal{L}$ denote the set of all links in the DCN. We can estimate the load-deviation parameter by

$$L_D = \frac{1}{|\mathcal{L}|} \sum_{l_{i,j} \in \mathcal{L}} (1 + |\zeta_{i,j}(t) - \zeta_{\text{avg}}(t)|)^2, \tag{2}$$

where $\zeta_{\text{avg}}(t)$ is the average traffic load of all links at time t. Here, Eq. 2 borrows the notion of the variance equation. Since $0 \leq |\zeta_{i,j}(t) - \zeta_{\text{avg}}(t)| < 1$, we thus add an offset of 1 to it. Obviously, when traffic loads of links become more unbalanced, L_D increases accordingly. Theorem 3 analyzes the range of L_D's value.

Given the load-deviation parameter L_D, we can check if it exceeds a threshold δ (i.e., some links are burdened with heavy loads) and invoke the ARM mechanism accordingly. However, there may be impulse flows in the DCN that generate volumes of data in a short time but soon disappear. In this case, there is no need to run the ARM mechanism, as switches will not be congested by such flows for a long time. Thus, we check the condition of $L_D > \delta$ for CW *consecutive* times to determine whether to invoke the ARM mechanism. As shown in Fig. 2, we reduce a counter CW by one whenever the condition is true, or restore CW to its default value. When $CW = 0$, the ARM mechanism will be executed. After the execution of ARM, we restart the above procedure to check L_D in the next slot. In this case, the "restore CW" block in the flowchart of Fig. 2 will reset CW to its default value. We give an example in Fig. 4, where the default value of CW is 2. In the example, the ARM mechanism will be invoked on slots 5 and 10.

Theorem 3. *The inequality $1 \leq L_D \leq 2.25$ must hold.*

Proof: Observing from Eq. 2, the minimum value of L_D occurs when every $\zeta_{i,j}(t)$ is equal to $\zeta_{\text{avg}}(t)$ (i.e., all links have the same traffic load). In this case, we can derive that

$$L_D = \frac{1}{|\mathcal{L}|} \sum_{l_{i,j} \in \mathcal{L}} (1 + 0)^2 = 1.$$

The maximum value of L_D occurs when one half of all links in $\mathcal{L}$ each has the maximum load, while the other half of links each has the minimum load. Based on the definition in Eq. 1,

TABLE 5: Default routes installed in switch s_i's group table.

group identifier	group type	action buckets
group_id=10	select	bucket=weight:100, actions=output:1 bucket=weight:0, actions=output:3

the load $l_{i,j}$ is between 0 and 1. In this case, we can calculate that $\zeta_{\text{avg}}(t) = 0.5$, and derive that

$$L_D = \frac{1}{|\mathcal{L}|} \sum_{l_{i,j} \in \mathcal{L}} (1 + 0.5)^2 = 2.25,$$

which proves the theorem. □

From the analysis in Theorem 3, we can quantify L_D's range into percentage to reflect the degree of how traffic loads of links are *unbalanced*, where the case of $L_D = 1$ corresponds to 0% (i.e., the least unbalancing case) and the case of $L_D = 2.25$ corresponds to 100% (i.e., the most unbalancing case). In this way, one suggested value of δ is to take 25% of L_D's range (i.e., $\delta = 1.3125$). In our simulations, we thus set δ to 1.3. Except for the threshold δ, the default value of CW will also determine the triggering frequency of the ARM mechanism. Therefore, we will evaluate the effects of both δ and CW in Section 5.4.

4.4 The ARM Mechanism

When some links are about to be congested, the controller carries out the ARM mechanism to reroute their traffic flows for load balance. Specifically, the ARM mechanism has three stages. In the *initial stage*, each new flow is assigned routes based on the path table. Then, the controller adjusts the amount of flows' data sent through different routes in the *rerouting stage*. Finally, the controller installs necessary flow entries to make switches reroute these flows in the *reaction stage*.

4.4.1 Initial Stage

The stage is used only when a switch s_i catches one packet of a *new* flow f_j. In this case, s_i notifies the controller of f_j's source and destination IP addresses via a Pack_In message. The controller then refers to the path table to select two routes for f_j to send its packets. One is the *primary* route and the other is a *backup* route[1]. As their names would suggest, f_i's packets will be sent through the primary route as a default, and the backup route is used for the load-balancing purpose. Afterwards, the controller installs flow entries in the switches along these two routes by replying Packet_Out messages to them. Let us consider an example. Suppose that the source and destination IP addresses of f_j are 10.0.0.1 and 10.0.0.16, respectively. Then, the controller installs a flow entry in s_i's flow table as follows:

match fields: ip, nw_src=10.0.0.1, nw_dst=10.0.0.16
instructions: actions=group:10

The flow entry asks s_i to consult the group table with ID 10 to send f_j's packets, as given in Table 5. In the example, the primary and backup routes of f_j will run through s_i's 1st and 3rd ports, respectively. Theorem 4 analyzes the number of Packet_Out messages that the controller uses to install flow entries in switches by the initial stage.

1. In Section 5.5, we will evaluate the effect of the number of backup routes. Simulation result show that it is enough to use one backup route to keep flow throughput while reducing the overflow probability.

To deal with the *stale route problem*, where some links may be broken but s_i still forwards packets to these links, we set an idle-timeout T_{idle} for the flow entry. Thus, s_i will remove the flow entry (and also the group table) after T_{idle} expires. Using the idle-timeout can also helps s_i periodically weed out old flow entries and prevent its flow table from overflowing. Remark 1 discusses how to decide the value of T_{idle}.

Theorem 4. *Given m new flows, the controller requires no more than $8m$ Packet_Out messages to install necessary flow entries in switches to deal with these flows in a fat-tree DCN.*

Proof: Observing from the fat-tree topology in Fig. 1, the longest route for a flow will go through one core switch, two aggregation switches, and two edge switches. However, the primary and backup routes of a flow must share the same edge switches (i.e., one connects to the source server and the other connects to the destination server). Thus, the controller has to install flow entries in at most 8 switches (for both primary and backup routes of a flow) in the worst case. Since it is enough to use one Packet_Out message to install flow entries in a switch, the controller will transmit at most $8m$ Packet_Out messages to switches for the new flows, which proves this theorem. □

Remark 1 (Set the value of T_{idle})**.** *To prevent each flow entry from occupying the space of a flow table for a long time, the work [30] sets the value of idle-timeout between $T_{\text{idle}}^{\min}$ and $T_{\text{idle}}^{\max}$, whose suggested values are 10 and 50, respectively. In fact, the idle-timeout T_{idle} should vary depending on the size of available space of a flow table. Specifically, if less space remains in the flow table, we should assign a shorter idle-timeout for the flow entry, and vice versa. Thus, we improve the method in [30] by setting T_{idle} as follows:*

$$T_{\text{idle}} = T_{\text{idle}}^{\max} - \frac{N_{\text{used}}}{N_{\text{total}}} \times (T_{\text{idle}}^{\max} - T_{\text{idle}}^{\min}), \tag{3}$$

where N_{used} and N_{total} denote the number of used flow entries and the maximum size of the flow table (also measured in flow entries), respectively. In Eq. 3, T_{idle} will be limited between $T_{\text{idle}}^{\min}$ and $T_{\text{idle}}^{\max}$. Besides, it decreases as the flow table contains more flow entries. We will also measure the effect of different $T_{\text{idle}}^{\min}$ and $T_{\text{idle}}^{\max}$ values in Section 5.6. □

4.4.2 Rerouting Stage

When a link becomes congested, the controller asks its switch to also forward some packets to the backup route to share the link's load. In particular, the rerouting stage contains the following steps:

- **S1.** Select the link $l_{i,j}$ with the maximum $\zeta_{i,j}(t)$ value from $\mathcal{L}$ (i.e., the busiest link). If there is a tie, we arbitrarily pick a link. Let us denote by s_a the host switch of $l_{i,j}$.
- **S2.** Among the flows transmitted through $l_{i,j}$, we select the flow f_k with the largest traffic demand. If there is a tie, we arbitrarily pick a flow. Assume that the primary and backup routes of f_k go through ports p_i and p_b of switch s_a, respectively.
- **S3.** If $\zeta_{b,x}(t) \geq \zeta_{i,j}(t)$ by referring to Eq. 1, which means that link $l_{b,x}$ in the backup route is also busy, we find another backup route from the path table, say, with link $l_{b',y}$ through s_a's port $p_{b'}$ such that $\zeta_{b',y}(t) < \zeta_{i,j}(t)$ and $\zeta_{b',y}(t)$ is the minimum. Then, we replace the backup route via p_b with the new route via $p_{b'}$ in s_a's group entry. Otherwise, we skip this step.

S4. Let A_i and A_b be the amount of data sent through ports p_i and p_b, respectively[2]. Then, we adjust the bucket weights of both p_i and p_b by

$$w_i = \frac{A_b}{A_i + A_b} \times 100, \quad (4)$$

$$w_b = 100 - w_i. \quad (5)$$

In this way, s_a will forward each of f_k's packet via its ports p_i and p_b, with probabilities of $w_i/100$ and $w_b/100$, respectively. Thus, the traffic load of link $l_{i,j}$ (via port p_i) can be shared by the backup route (via port p_b).

S5. Update the load-deviation parameter L_D in Eq. 2. If L_D is still above the threshold δ, we return to step 1 to find another busy link. Otherwise, the algorithm terminates.

Let us give an example of weight adjustment in step S4 by Table 5. Suppose that $A_i = 4203952$ (for port 1) and $A_b = 1326821$ (for port 3). Then, the bucket weight w_i for port 1 will be $\frac{1326821}{4203952+1326821} \times 100 = 24$ and the bucket weight w_b for port 3 will be $100 - 24 = 76$. Therefore, around 24% and 76% of f_k's packets will be sent through ports 1 and 3, thereby alleviating the load of link $l_{i,j}$.

We discuss the rationale of this stage. Since the backup route via port p_b has a lighter load than the primary route via port p_i, we adjust the bucket weights w_i and w_b to make them inversely proportional to the amount of data transmitted through p_i and p_b, respectively. Thus, the switch can send more packets of flow f_k through the backup route whose current load is not heavy. Only when the backup route is also busy, will the controller finds a new route (through port $p_{b'}$) in step S3 to share the load of the primary route. In this way, we can greatly reduce the computation cost and message overhead of the controller, as it simply adjusts the bucket weights of existing routes in a switch, instead of frequently consulting the path load to install new routes in multiple switches. Theorem 5 analyzes the time complexity of the rerouting stage.

Theorem 5. *Given n_l links in the DCN, the worst-case time complexity of the rerouting stage is $O(n_l^2)$.*

Proof: In step S1, we have to pick out the busiest link from $\mathcal{L}$. To facilitate this procedure, we can build a maximum heap to store all links by their $\zeta_{i,j}(t)$ values, which spends $O(n_l)$ time. Then, it takes $O(\lg n_l)$ time to pick the busiest link from the heap. In step S2, suppose that at most n_f flows can be transmitted through a port, where $n_f \ll n_l$. It will take $O(n_f)$ time to find flow f_k from port p_i. In step S3, since each switch has k ports, we can find the backup route (via port b') with no more than $O(k)$ time. Obviously, we have $k \ll n_l$. Then, it takes constant time to do the calculation of Eqs. 4 and 5 in step S4. Besides, we will update the data structures of links $l_{b,x}$ and $l_{i,j}$ in the heap, which takes $2O(\lg n_l)$ time. In step S5, we need to compute L_D by Eq. 2, where it takes $O(n_l)$ time to find $\zeta_{\text{avg}}(t)$ and also $O(n_l)$ time to do the summation. Therefore, each iteration of the rerouting stage will spend time of $O(\lg n_l) + O(n_f) + O(k) + 2O(\lg n_l) + 2O(n_l) = O(n_l)$. The worst case occurs when we check every link in $\mathcal{L}$, which results in n_l iterations. Thus, the overall time complexity will be $O(n_l) + n_l \cdot O(n_l) = O(n_l^2)$. □

2. Obviously, if we do the replacement of backup route in step S3, it will be port $p_{b'}$ in step S4.

TABLE 6: Example of adjusting the polling period by the DIP mechanism.

successive time	polling period (seconds)
$\alpha < 3$	$T_{\text{poll}} = 5 \times 2^0 = 5$
$3 \le \alpha < 6$	$T_{\text{poll}} = 5 \times 2^1 = 10$
$6 \le \alpha < 9$	$T_{\text{poll}} = 5 \times 2^2 = 20$
$9 \le \alpha < 12$	$T_{\text{poll}} = 5 \times 2^3 = 40$
$12 \le \alpha < 15$	$T_{\text{poll}} = 5 \times 2^4 = 80$
$15 \le \alpha < 18$	$T_{\text{poll}} = 5 \times 2^5 = 160$
$\alpha \ge 18$	$T_{\text{poll}} = 5 \times 2^6 = 320$

4.4.3 Reaction Stage

The controller sends Packet_Out messages to switches to modify their flow entries (e.g., updating bucket weights). Since these messages are transmitted through the same network links with ordinary packets, the update of routes in different switches may not be consistent. In this case, some switches will not have correct flow entries to deal with the transmission of a flow, thereby causing the event of table miss and dropping its packets. To solve this problem, we suggest updating the flow tables of switches in a *reverse* manner. In particular, suppose that the new route of a flow is $s_1 \to s_2 \to \cdots \to s_n$. The controller then updates the flow tables of switches following the sequence of s_n, s_{n-1}, $\cdots$, and s_1. In this way, we can avoid the occurrence of a situation where the flow's packets come to a switch before the switch gets the Packet_Out message to update its route.

In the reaction stage, the number of Packet_Out messages that the controller has to send depends on the number of switches whose flow entries are updated in the rerouting stage. From Theorem 5, we observe that the ARM mechanism may spend more computational time. That is why we develop the method in Section 4.3 to significantly reduce the frequency of invoking the ARM mechanism, which avoids ARM becoming a bottleneck of our L2RM framework and also saves the computational resource of the controller.

4.5 The DIP Mechanism

To keep correct information of the path, load, and entry tables, the controller will periodically poll switches for their information to update these tables. Specifically, the polling period T_{poll} decides not only the correctness of table content but also the message overhead for the controller to get information of switches. Moreover, the polling period should be *dynamic*, since it is not necessary for the controller to frequently query switches in some cases. For example, when the network is stable and most links are load-balanced, there is no need to invoke the ARM mechanism to reroute packets. In this case, the controller can actually update its tables at a later time and save the message overhead accordingly.

Based on this observation, the DIP mechanism adjusts the polling period by borrowing the notion of *exponential backoff* in IEEE 802.11 [31] as follows:

$$T_{\text{poll}} = T_{\text{check}} \times 2^{\alpha/\beta}, \quad (6)$$

where T_{check} is the minimum time interval for the controller to compute the load-deviation parameter L_D by Eq. 2, α is the successive time that the condition of $L_D \le \delta$ holds (i.e., the ARM mechanism is not invoked), and $\beta \in \mathbb{N}$ is a coefficient. In case of $L_D > \delta$, α is reset to 0. Besides, the maximum value of α is limited to 6β. Thus, the value of T_{poll} will not grow too large, and the controller can still update its tables when the network keeps stable for a long time. Table 6 gives an example,

where we set T_{check} to 5 seconds and β to 3. In Eq. 6, the coefficient β provides flexibility in adjusting the polling period. Specifically, when most links are load-balanced (i.e., L_D keeps below δ) in the current T_{poll} period, we can double the value of T_{poll} to further reduce the message overhead incurred by the controller.

5 Performance Evaluation

We implement our L2RM framework on the Mininet simulator [32], which supports the OpenFlow protocol. The controller is implemented by Ryu [33], which is a popular component-based framework for SDN implementation. On the other hand, switches are implemented by the Open vSwitch module [34] to simulate the behavior of OpenFlow switches. Besides, we use the iPerf tool [35] to generate real traffic flows in a network.

In our simulations, we consider a DCN whose topology is a 4-ary fat tree, as shown in Fig. 1. Moreover, there are three scenarios used to model the generation of traffic flows:

- *Multi-destination (MD) scenario:* We choose two servers linked by the same edge switch to be sources. Each of them will send packets to five servers.
- *Uniform traffic (UT) scenario:* There are ten servers to generate flows, where each one has a different destination. Thus, flows may be distributed among links.
- *Concentrated traffic (CT) scenario:* We select twelve servers as sources and three servers as destinations. This scenario can be used to model the traffic pattern where many users query resource possessed by just few servers.

Each flow is a TCP connection and produces packets in every [10, 15] seconds. The simulation time is 300 seconds.

We compare L2RM with three SDN-based methods discussed in Section 3, including RR [14], LABERIO [27], and DLPO [28]. They also aim to balance traffic loads of links in a fat-tree DCN. As mentioned in Section 4.3, we set the threshold δ to 1.3 in L2RM. Besides, we also set $T_{\text{idle}}^{\min} = 10$ and $T_{\text{idle}}^{\max} = 50$ in Eq. 3. Below, we measure the performance of each method in terms of link utilization, table overflow, and message overhead. Then, we evaluate the effects of different parameters, including δ, CW, the number of backup routes, $T_{\text{idle}}^{\min}$, and $T_{\text{idle}}^{\max}$ on L2RM performance.

5.1 Link Utilization

We compare the average link utilization of each method, as shown in Fig. 5. Since the amount of data generated by traffic flows are fixed, high link utilization implies that the method could make good use of different routes to send data. There is a similar periodicity in changes of link utilization, where the period length is about 10 to 15 seconds (i.e., consistent with the time interval that each flow produces packets). Besides, most methods have lower link utilization in the MD scenario, as there are only two servers to generate flows and they are connected by the same edge switch.

The RR method asks a switch to send packets of each flow via two fixed links in turns. When these links become congested, packet loss may frequently occur. That is why RR results in the lowest link utilization in Fig. 5. By dynamically changing routes, LABERIO, DLPO, and L2RM can increase their link utilization accordingly. Our L2RM framework significantly outperforms LABERIO and DLPO in the UT scenario.

TABLE 7: Confidence intervals of average link utilization in Fig. 5.

method	MD scenario	UT scenario	CT scenario
RR	20.13 ± 2.27	24.87 ± 2.81	26.70 ± 3.02
LABERIO	26.74 ± 3.02	31.99 ± 3.61	30.42 ± 3.44
DLPO	24.26 ± 2.74	29.71 ± 3.36	30.31 ± 3.42
L2RM	28.35 ± 3.20	35.17 ± 3.97	34.37 ± 3.88

The reason is that each source server has its own destination. Thus, traffic flows may be evenly distributed among links, and L2RM can better use group tables to balance loads of links.

Table 7 presents the confidence intervals of average link utilization by different methods in Fig. 5, where the confidence level is 95%. The RR method has higher link utilization in the CT scenario than in other scenarios, since there are the most source servers (i.e., 12 servers) in this scenario. On the other hand, LARBERIO, DLPO, and L2RM perform better in the UT scenario, because there are more source servers (i.e., 10 servers) and each of them has its own destination. Thus, there are more choices for these methods to pick backup routes to balance link loads. From Table 7, we observe that L2RM always has the best performance among all methods in different scenarios, which demonstrates its effectiveness on improving link utilization.

We also evaluate the standard deviation of link loads, as given in Fig. 6. Specifically, the lower the standard deviation, the more balanced the link loads. Since the MD scenario is a multicast scenario with only two sources, all methods have lower standard deviation in the MD scenario than in other scenarios. In the RR method, as each switch uses two fixed links to transmit packets of a flow by turns, the loads of these links may become similar. That is why it results in the lowest standard deviation. On the other hand, L2RM can have lower standard deviation than both LARBERIO and DLPO, especially in the UT and CT scenarios. That is because L2RM adopts group tables for the routing purpose and adaptively adjusts the bucket weights of routes to decide the percentages of packets to be sent via these routes. In this way, L2RM can further balance link loads.

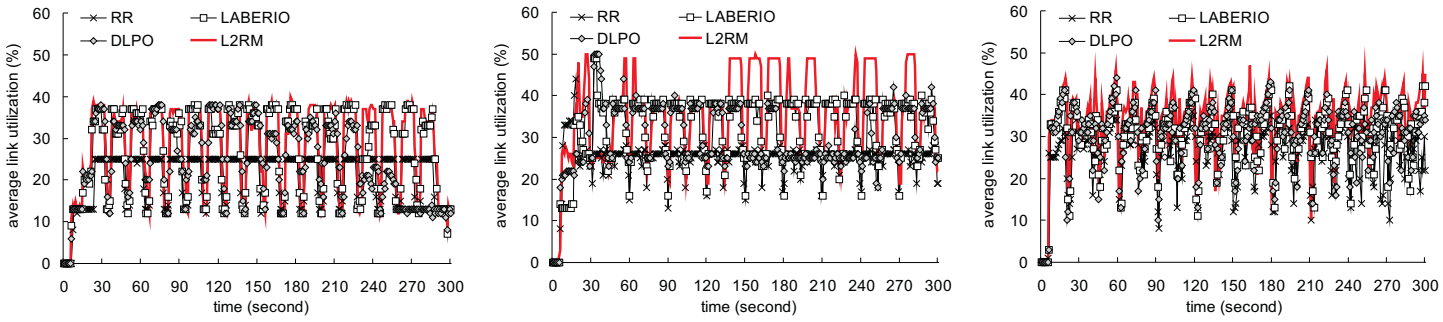
5.2 Table Overflow

We then measure the degree of table overflow by different methods, which can be evaluated by two metrics:

- *Blocked packet:* When a switch has no idea to handle a packet, if no additional flow entries can be installed (e.g., due to running out of table space), the packet is called *blocked* [30] and has to be discarded by the switch.
- *Table-full event:* When a switch uses up its table space but the controller seeks to install new flow entries (e.g., due to blocked packets), an event of table full will occur.

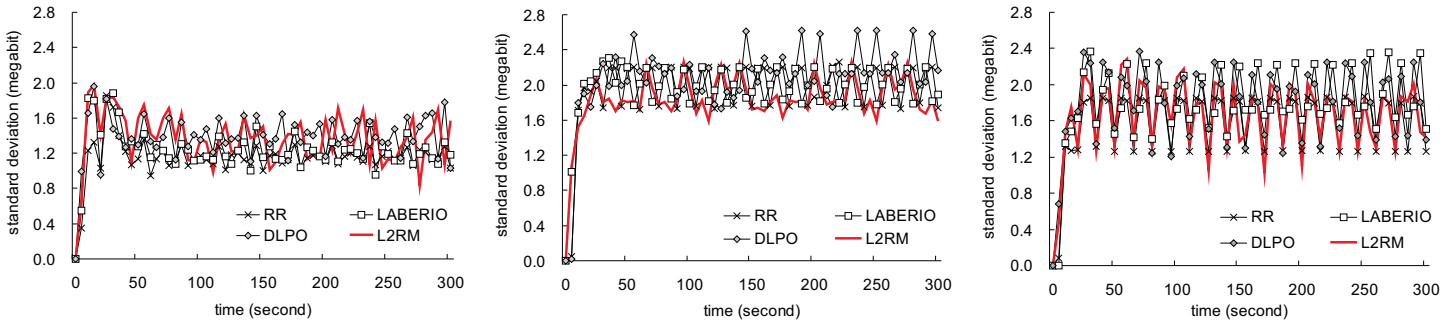
The number of table-full events will be larger than the number of blocked packets, because each blocked packet will be also sent to the controller for processing. In fact, when a table-full event occurs, it means that the controller tries in vain to install flow entries in the switch, which wastes resource. Therefore, we can also use the number of table-full events to measure the amount of "useless" load by the controller.

Fig. 7 gives the number of blocked packets generated by each method, while Fig. 8 shows the number of table-full events caused by each method, where we increase table size from 10 to 50 flow entries. When the DCN becomes more congested, both blocked packets and table-full events will increase, because the controller has to install more flow entries



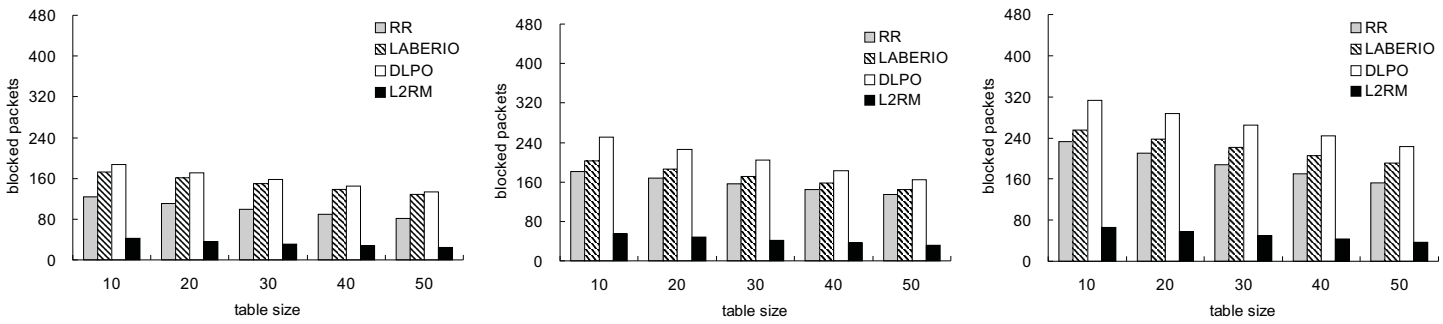
(a) MD scenario (b) UT scenario (c) CT scenario

Fig. 5: Comparison on the average link utilization.



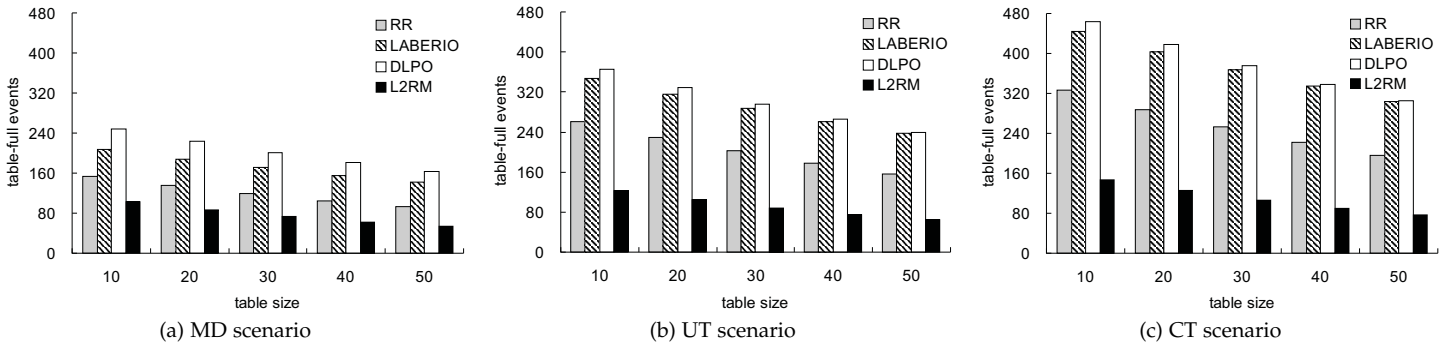
(a) MD scenario (b) UT scenario (c) CT scenario

Fig. 6: Comparison on the standard deviation of link loads.



(a) MD scenario (b) UT scenario (c) CT scenario

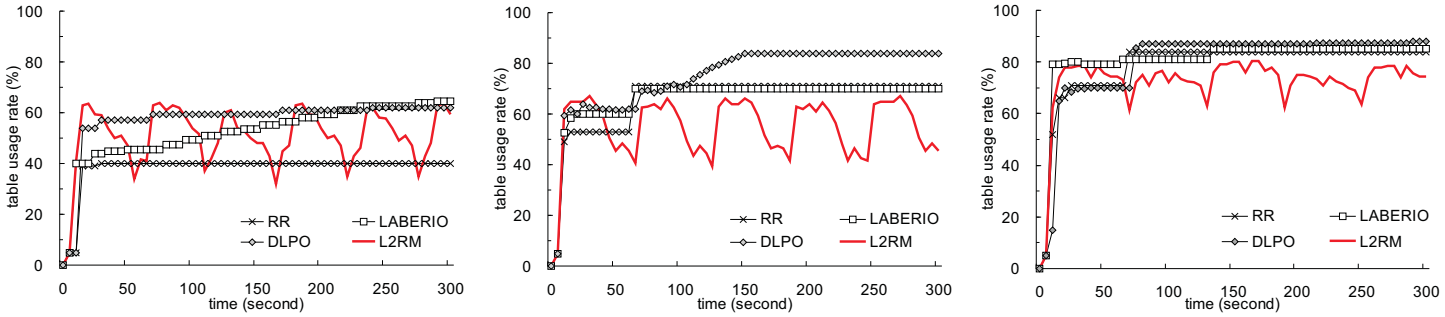
Fig. 7: Comparison on the number of blocked packets.



(a) MD scenario (b) UT scenario (c) CT scenario

Fig. 8: Comparison on the number of table-full events.

to deal with packet routes for such a situation. Consequently, all methods result in the smallest (respectively, largest) blocked packets and table-full events in the MD (respectively, CT) scenario. Besides, increasing table size can help reduce both blocked packets and table-full events, as each switch has more memory space to store more flow entries. From both Fig. 7



(a) MD scenario (b) UT scenario (c) CT scenario

Fig. 9: Comparison on the average usage rate of flow tables.

TABLE 8: Reduction ratio of message overhead by L2RM in Fig. 10.

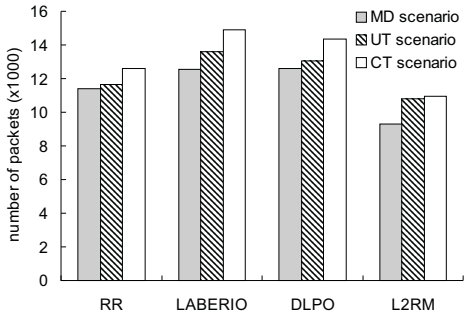
method	MD scenario	UT scenario	CT scenario
RR	18.75%	5.79%	5.63%
LABERIO	49.29%	45.10%	50.96%
DLPO	24.50%	22.51%	24.26%

and 8, we observe that DLPO has the most blocked packets and table-full events, followed by LABERIO and RR. On the other hand, our L2RM framework can greatly reduce blocked packets and table-full events, as comparing with the above three methods. That is because L2RM allows each switch to adaptively remove old records from its flow table by using Eq. 3 to set an idle-timeout for every flow entry, which helps save its table space.

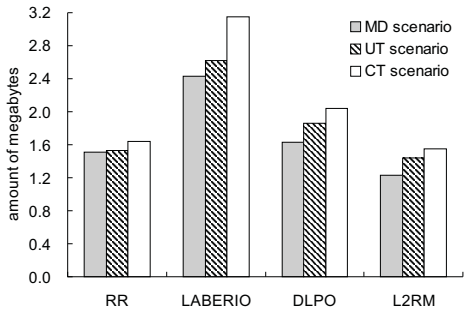
We also study the usage rates of flow tables by different methods. In the experiment, we set table size to 20 flow entries. When a flow table is full but the controller still asks its switch to install new flow entries, these flow entries will be discarded accordingly. Fig. 9 gives the average usage rate of the top 50% of table tables (in terms of their usage rates). Since RR, LABERIO, and DLPO do not consider removing old flow entries, their usage rates will increase and then keep in high values as time goes by. On the contrary, L2RM uses Eq. 3 to decide the lifetime of each flow entry. Thus, a switch can automatically remove stale ones from its flow table. That is why the usage rate of L2RM periodically decreases and then increases. In the MD scenario, since there are only two sources, RR will install the fewest flow entries among all methods. L2RM can regularly weed out unused flow entries, so it has a lower usage rate than LARBERIO and DLPO. On the other hand, in both UT and CT scenarios, as more switches are involved in relaying packets, more flow entries have to be installed. Thanks to the setting of idle-timeout by Eq. 3, L2RM can conserve table space and thus result in the lowest table usage rate in these two scenarios.

5.3 Message Overhead

Then, we evaluate the message overhead for the controller to query and command switches, whose experimental results are given in Fig. 10. On the whole, each method requires more SDN messages in the CT scenario, followed by UT and MD scenarios. The reason is that there are the most sources that produce packets in the CT scenario, while only two sources are used in the MD scenario. For the RR method, since each switch sends packets of a flow via two routes in turns, the controller need not exchange many messages with switches. On the contrary, the controller will incur the highest message



(a) measured in packets



(b) measured in megabytes

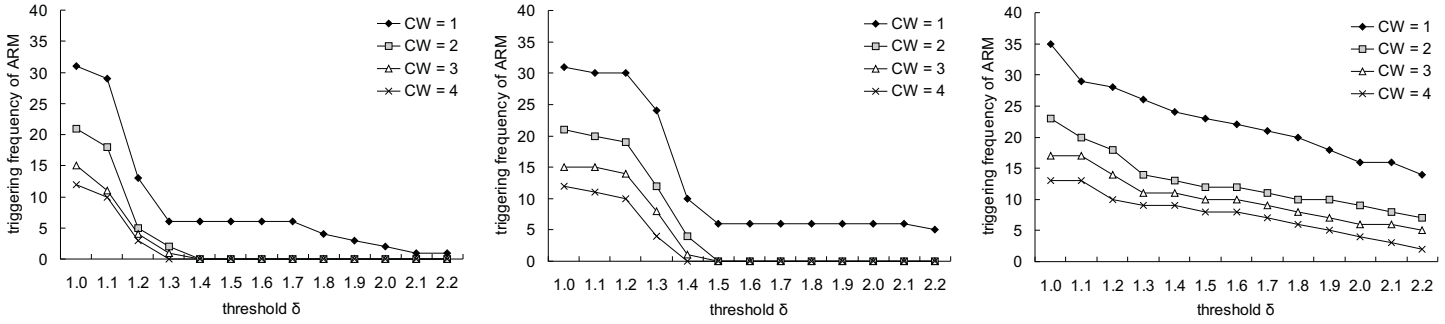
Fig. 10: Comparison on the message overhead.

overhead in LABERIO, as it frequently asks the controller to do the load-balancing job. As discussed in Section 3.2, DLPO uses simple moving average to reduce its triggering frequency. Thus, it will have a lower message overhead than LABERIO. Among all methods, L2RM can always result in the lowest message overhead due to the design of the DIP mechanism in Section 4.5.

Table 8 presents the reduction ratio of message overhead by L2RM, as comparing with other methods. On the average, our L2RM framework can save 10.06%, 48.45%, and 23.76% of SDN message overheads as comparing with RR, LABERIO, and DLPO, which verifies its message efficiency.

5.4 Effect of δ and CW

In Section 4.3, we use both parameters δ and CW to control when to invoke the ARM mechanism to do the load-balancing job in L2RM. As discussed in Theorem 5, the rerouting stage of ARM is a time-consuming operation. Thus, we investigate how these two parameters affect the triggering frequency of ARM. In Theorem 3, we show that the range of the load-deviation parameter L_D is between 1 and 2.25. Therefore, starting from



(a) MD scenario (b) UT scenario (c) CT scenario

Fig. 11: Effect of parameters δ and CW on the triggering frequency of ARM.

$\delta = 1$, we iteratively add it by 0.1, until $\delta = 2.2$ in the experiment.

Fig. 11 shows the experimental results. Generally speaking, the triggering frequency decreases when δ grows, because only when $L_D > \delta$ will we seek to invoke the ARM mechanism. In addition, the ARM mechanism will be executed only when the condition of $L_D > \delta$ holds for CW successive slots. Thus, a larger CW value, a lower triggering frequency. In the MD scenario, since there are only two sources that generate flows, the triggering frequency drastically drops when $\delta \geq 1.3$. In the UT scenario, we increase more sources, so the triggering frequency significantly decreases when δ exceeds 1.3. On the other hand, since there are the most sources that send packets to only three destinations, the network will easily become congested in the CT scenario. Thus, the triggering frequency gradually decreases when δ increases in the UT scenario.

Observing from Fig. 11, a better choice is to set $\delta = 1.3$ and $CW = 2$, because this setting still allows us to invoke the ARM mechanism to balance traffic loads among flows in the MD scenario. Besides, it can greatly reduce the triggering frequency of ARM in both UT and CT scenarios, so as to save the computational resource of the controller.

5.5 Effect of Backup Routes

Next, we evaluate the effect of different numbers of backup routes on both the average flow throughput and the overflow probability. Fig. 12 presents the experimental results, where we consider 0, 1, 2, and 3 backup routes for each flow. Broadly speaking, increasing the number of backup routes can improve flow throughput, because there are more choices for a switch to relay the packets of each flow. However, it also substantially increases the overflow probability of the flow table, since the switch needs to store more routing information. Such a trend is more obvious in the CT scenario, as there are the most sources that transmit packets to just three destinations. From Fig. 12, we suggest using one backup route for the three traffic scenarios, since it can increase flow throughput but will not drastically increase the overflow probability.

5.6 Effect of $T_{\text{idle}}^{\min}$ and $T_{\text{idle}}^{\max}$

To conserve table space, L2RM adopts Eq. 3 to compute the idle-timeout T_{idle} for each flow entry, which are decided by $T_{\text{idle}}^{\max}$, $T_{\text{idle}}^{\min}$, and the number of used flow entries N_{used}. Fig. 13 shows how T_{idle} changes when we vary the value of $T_{\text{idle}}^{\min}$, where we set table size to 30 flow entries and $T_{\text{idle}}^{\max}$ to 50. The value of T_{idle} will decrease linearly as the value of N_{used} grows.

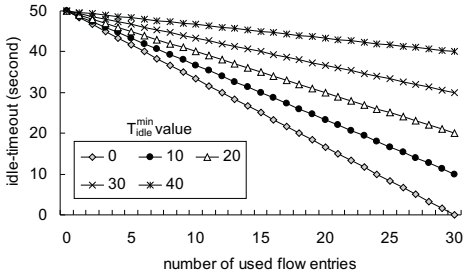
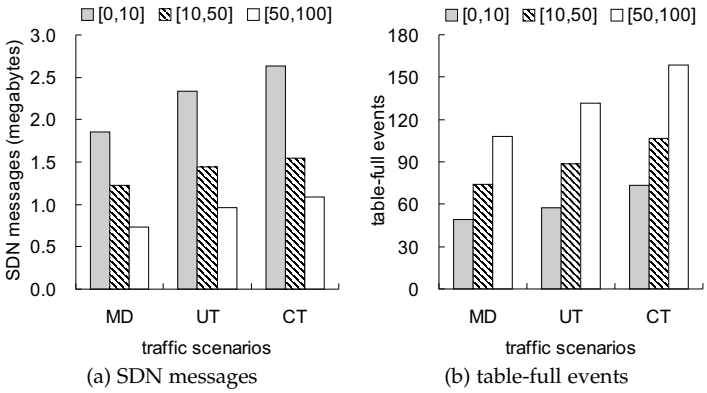


Fig. 13: Effect of $T_{\text{idle}}^{\min}$ value on the idle-timeout T_{idle}.

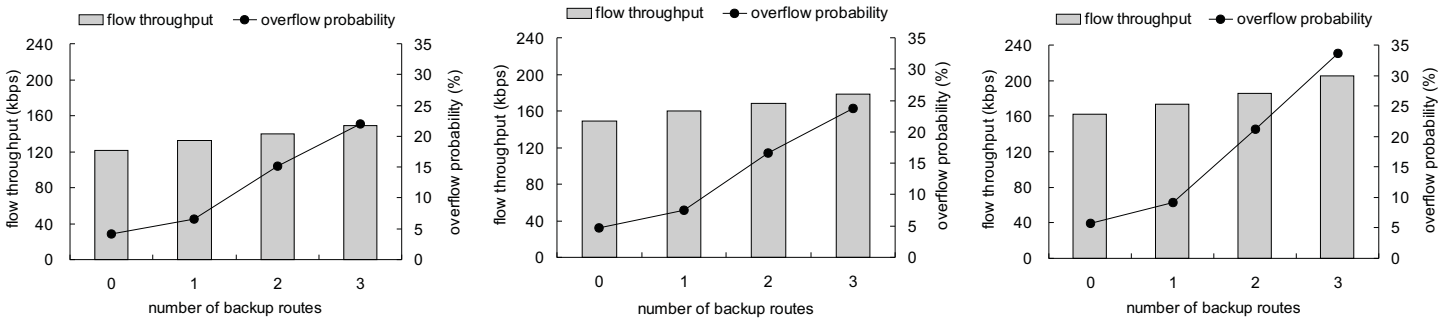
Besides, the slope of T_{idle}'s result depends on the gap between $T_{\text{idle}}^{\max}$ and $T_{\text{idle}}^{\min}$. Specifically, the larger the gap, the larger the slope. In other words, a new flow entry will be given a shorter lifetime when $(T_{\text{idle}}^{\max} - T_{\text{idle}}^{\min})$ increases. In this way, we can speed up removing flow entries when there remains less space in a flow table.



(a) SDN messages (b) table-full events

Fig. 14: Effect of different $[T_{\text{idle}}^{\min}, T_{\text{idle}}^{\max}]$ ranges.

We finally study the effect of different $[T_{\text{idle}}^{\min}, T_{\text{idle}}^{\max}]$ ranges on SDN messages and table-full events in L2RM, where we set $[T_{\text{idle}}^{\min}, T_{\text{idle}}^{\max}]$ to $[0, 10]$, $[10, 50]$, and $[50, 100]$. Fig. 14 gives the experimental results. In particular, when the $[T_{\text{idle}}^{\min}, T_{\text{idle}}^{\max}]$ range is smaller, each flow entry will be given a shorter idle-timeout by Eq. 3. In this case, since a switch will fast remove its flow entries, the controller has to reinstall the removed flow entries (for packet routes). On the other hand, a shorter idle-timeout means that the flow table will not be occupied by old flow entries for long time. Thus, the table-full events will decrease accordingly. From Fig. 14, a better choice is to set $[T_{\text{idle}}^{\min}, T_{\text{idle}}^{\max}]$ to $[10, 50]$, since it can balance between SDN



(a) MD scenario (b) UT scenario (c) CT scenario

Fig. 12: Effect of the number of backup routes on the average flow throughput and the overflow probability.

messages and table-full events.

6 Conclusion

DCNs have been widely deployed to support the growing demand of cloud services. It is important to manage packet routes in a DCN to avoid network congestion by sharing loads among links. We thus propose the L2RM framework to improve performance of fat-tree DCNs by exploiting SDN with a low cost, which keeps monitoring network condition and uses the ARM mechanism to reroute traffic flows only if necessary. Besides, the DIP mechanism is also developed for the controller to adaptively query switches about their information with fewer SDN messages. By considering different traffic scenarios in simulations, we show that L2RM can increase link utilization, balance traffic loads, conserve table space, reduce blocked packets, and alleviate table-full events, as comparing with the RR, LABERIO, and DLPO methods. Moreover, we also evaluate the effects of parameters such as δ, CW, backup routes, and $[T_{\text{idle}}^{\min}, T_{\text{idle}}^{\max}]$ on L2RM performance.

In this paper, we aim to balance traffic loads of links with the minimum SDN overhead. When there are also congested links in backup routes or failures in switches in primary or backup routes, the controller may react to such events only after it queries these switches (e.g., by the SHC mechanism). Thus, it deserves further investigation on how to reduce the response time of the controller when failure occurs by, for example, dynamically adjusting the polling period of the SHC mechanism or developing a fast error-reporting mechanism. Moreover, we will consider implementing L2RM in a large-scale DCN in the future.

References

[1] P. Lu, L. Zhang, X. Liu, J. Yao, and Z. Zhu, "Highly efficient data migration and backup for big data applications in elastic optical inter-data-center networks," *IEEE Network*, vol. 29, no. 5, pp. 36–42, 2015.

[2] W. Xia, P. Zhao, Y. Wen, and H. Xie, "A survey on data center networking (DCN): infrastructure and operations," *IEEE Comm. Surveys & Tutorials*, vol. 19, no. 1, pp. 640–656, 2017.

[3] T. Wang, Z. Su, Y. Xia, and M. Hamdi, "Rethinking the data center networking: architecture, network protocols, and resource sharing," *IEEE Access*, vol. 2, pp. 1481–1496, 2014.

[4] D. Kreutz, F.M.V. Ramos, P.E. Verissimo, C.E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: a comprehensive survey," *Proc. IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[5] S.Q. Zhang, Q. Zhang, H. Bannazadeh, and A. Leon-Garcia, "Routing algorithms for network function virtualization enabled multicast topology on SDN," *IEEE Trans. Network and Service Management*, vol. 12, no. 4, pp. 580–594, 2015.

[6] T. Zhu, D. Feng, F. Wang, Y. Hua, Q. Shi, J. Liu, Y. Cheng, and Y. Wan, "Efficient anonymous communication in SDN-based data center networks," *IEEE/ACM Trans. Networking*, vol. 25, no. 6, pp. 3767–3780, 2017.

[7] J.H. Cox, R. Clark, and H. Owen, "Leveraging SDN and WebRTC for rogue access point security," *IEEE Trans. Network and Service Management*, vol. 14, no. 3, pp. 756–770, 2017.

[8] Y.C. Wang and H.Hu, "An adaptive broadcast and multicast traffic cutting framework to improve Ethernet efficiency by SDN," *J. Information Science and Engineering*, 2018.

[9] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: enabling innovation in campus networks," *ACM SIGCOMM Computer Comm. Review*, vol. 38, no. 2, pp. 69–74, 2008.

[10] ONF. [Online]. Available: https://www.opennetworking.org/

[11] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," *Proc. ACM SIGCOMM Conf. Data Comm.*, 2008, pp. 63–74.

[12] K. Garala, N. Goswami, and P.D. Maheta, "A performance analysis of load balancing algorithms in cloud environment," *Proc. Int'l Conf. Computer Comm. and Informatics*, 2015, pp. 1–6.

[13] R. Lee and B. Jeng, "Load-balancing tactics in cloud," *Proc. Int'l Conf. Cyber-Enabled Distributed Computing and Knowledge Discovery*, 2011, pp. 447–454.

[14] H. Zhang and X. Guo, "SDN-based load balancing strategy for server cluster," *Proc. IEEE Int'l Conf. Cloud Computing and Intelligence Systems*, 2014, pp. 662–667.

[15] M. Qilin and S. Weikang, "A load balancing method based on SDN," *Proc. Int'l Conf. Measuring Technology and Mechatronics Automation*, 2015, pp. 18–21.

[16] A.C.S. Araujo, L.N. Sampaio, and A.Ziviani, "BEEP: balancing energy, redundancy, and performance in fat-tree data center networks," *IEEE Internet Computing*, vol. 21, no. 4, pp. 44–53, 2017.

[17] L. Chen, K. Chen, W. Bai, and M. Alizadeh, "Scheduling mix-flows in commodity datacenters with Karuna," *Proc. ACM SIGCOMM Conf.*, 2016, pp. 174–187.

[18] W. Bai, L. Chen, K. Chen, and H. Wu, "Enabling ECN in multi-service multi-queue data centers," *Proc. USENIX Symp. Networked Systems Design and Implementation*, 2016, pp. 537–550.

[19] S. Ghorbani, Z. Yang, P.B. Godfrey, Y. Ganjali, and A. Firoozshahian, "DRILL: micro load balancing for low-latency data center networks," *Proc. ACM SIGCOMM Conf.*, 2017, pp. 225–238.

[20] A. Chatzieleftheriou, S. Legtchenko, H. Williams, and A. Rowstron, "Larry: practical network reconfigurability in the data center," *Proc. USENIX Symp. Networked Systems Design and Implementation*, 2018, pp. 141–156.

[21] D. Adami, G. Antichi, R.G. Garroppo, S. Giordano, and A.W. Moore, "Towards an SDN network control application for differentiated traffic routing," *Proc. IEEE Int'l Conf. Comm.*, 2015, pp. 5827–5832.

[22] D.B. Johnson, "A note on Dijkstra's shortest path algorithm," *J. ACM*, vol. 20, no. 3, pp. 385–388, 1973.

[23] M.F. Ramdhani, S.N. Hertiana, and B. Dirgantara, "Multipath routing with load balancing and admission control in software-defined networking (SDN)," *Proc. Int'l Conf. Information and Comm. Technology*, 2016, pp. 1–6.

[24] A. Laszka and A. Gueye, "Network topology vulnerability/cost trade-off: model, application, and computational complexity," *Internet Mathematics*, vol. 11, no. 6, pp. 588–626, 2015.

[25] J. Liu, J. Li, G. Shou, Y. Hu, Z. Guo, and W. Dai, "SDN based load balancing mechanism for elephant flow in data center networks,"

Proc. Int'l Symp. Wireless Personal Multimedia Comm., 2014, pp. 486–490.
[26] D. Adami, S. Giordano, M. Pagano, and G. Portaluri, "A novel SDN controller for traffic recovery and load balancing in data centers," *Proc. IEEE Int'l Workshop on Computer Aided Modelling and Design of Comm. Links and Networks*, 2016, pp. 77–82.
[27] H. Long, Y. Shen, M. Guo, and F. Tang, "LABERIO: dynamic load-balanced routing in OpenFlow-enabled networks," *Proc. IEEE Int'l Conf. Advanced Information Networking and Applications*, 2013, pp. 290–297.
[28] Y.L. Lan, K. Wang, and Y.H. Hsu, "Dynamic load-balanced path optimization in SDN-based data center networks," *Proc. Int'l Symp. Comm. Systems, Networks and Digital Signal Processing*, 2016, pp. 1–6.
[29] D.C. Montgomery, G.C. Runger, and N.F. Hubele, *Engineering Statistics*, 4th ed, New York: John Wiley & Sons, 2009.
[30] L. Zhang, S. Wang, S. Xu, R. Lin, and H. Yu, "TimeoutX: an adaptive flow table management method in software defined networks," *Proc. IEEE Global Comm. Conf.*, 2015, pp. 1–6.
[31] G. Bianchi, "Performance analysis of the IEEE 802.11 distributed coordination function," *IEEE J. Selected Areas in Comm.*, vol. 18, no.3, pp. 535–547, 2000.
[32] Mininet. [Online]. Available: http://mininet.org/
[33] Ryu. [Online]. Available: https://osrg.github.io/ryu/
[34] Open vSwitch. [Online]. Available: https://www.openvswitch.org/
[35] iPerf. [Online]. Available: https://iperf.fr/