AggreFlow: Achieving Power Efficiency, Load Balancing, and Quality of Service in Data Center Networks

Zehua Guo, *Senior Member, IEEE, Member, ACM*, Yang Xu, *Member, IEEE*, Ya-Feng Liu, *Senior Member, IEEE*, Sen Liu, *Member, IEEE*, H. Jonathan Chao, *Life Fellow, IEEE*, Zhi-Li Zhang, *Fellow, IEEE, Member, ACM*, and Yuanqing Xia, *Senior Member, IEEE*

***Abstract*— Power-efficient Data Center Networks (DCNs) have been proposed to save power of DCNs using OpenFlow. In these DCNs, the OpenFlow controller adaptively turns on/off links and OpenFlow switches to form a minimum-power subnet that satisfies the traffic demand. As the subnet changes, flows are dynamically routed and rerouted to the routes composed of active switches and links. However, existing flow scheduling schemes could cause undesired results: (1) power inefficiency: due to unbalanced traffic allocation on active routes, extra switches and links may be activated to cater to bursty traffic surges on congested routes, and (2) Quality of Service (QoS) fluctuation: because of the limited flow entry processing ability, switches may not be able to timely install/delete/update flow entries to properly route/reroute flows. In this paper, we propose AggreFlow, a dynamic flow scheduling scheme that achieves power efficiency and QoS improvement using three techniques: *Flow-set Routing*, *Lazy Rerouting*, and *Adaptive Rerouting*. Flow-set Routing achieves load balancing with a small number of flow entry operations by routing flows in a coarse-grained flow-set fashion. Lazy Rerouting spreads rerouting operations over a relatively long period of time, reducing the burstiness of entry operation on switches. Adaptive Rerouting selectively reroutes flow-sets to maintain load balancing. We built an NS3 based fat-tree network simulation platform to evaluate AggreFlow's performance. The simulation results show that AggreFlow reduces power consumption by about 18%, yet achieving load balancing and improved QoS (low packet loss rate and reducing the number of processing entries for flow scheduling by 98%), compared with baseline schemes.**

***Index Terms*— Flow scheduling, power-efficient data center networks, power saving, OpenFlow.**

Manuscript received September 5, 2019; revised March 22, 2020 and May 9, 2020; accepted August 22, 2020; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor E. Uysal. Date of publication November 10, 2020; date of current version February 17, 2021. The work was supported in part by the National Key Research and Development Program of China under Grant 2018YFB1003700, in part by the Natural Science Foundation of China under Grant 62002019, Grant 11688101, Grant 11671419, Grant 11991021, and Grant 62002066, in part by the Beijing Institute of Technology Research Fund Program for Young Scholars, in part by the Project "PCL Future Greater-Bay Area Network Facilities for Large-scale Experiments and Applications (LZC0019)", and in part by the U.S. NSF under Grant CNS-1618339, Grant CNS-1617729, and Grant CNS-1814322. This article was presented in part at IEEE/ACM IWQoS 2016. *(Corresponding author: Yang Xu.)*

Zehua Guo and Yuanqing Xia are with the Beijing Institute of Technology, Beijing 100081, China.

Yang Xu is with the School of Computer Science, Fudan University, Shanghai 200433, China, and also with the Peng Cheng Laboratory, Shenzhen 518066, China (e-mail: xuy@fudan.edu.cn).

Ya-Feng Liu is with the State Key Laboratory of Scientific and Engineering Computing, Institute of Computational Mathematics and Scientific/Engineering Computing, Academy of Mathematics and Systems Science, Chinese Academy of Sciences, Beijing 100190, China.

Sen Liu is with the School of Computer Science, Fudan University, Shanghai 200433, China.

H. Jonathan Chao is with the New York University Tandon School of Engineering, Brooklyn, NY 11201 USA.

Zhi-Li Zhang is with the Department of Computer Science and Engineering, University of Minnesota Twin Cities, Minneapolis, MN 55455 USA.

Digital Object Identifier 10.1109/TNET.2020.3026015

I. INTRODUCTION

THE popularity of cloud services accelerates the expanding of data centers. The high power consumption of data centers has become one of the most important concerns of their operators. Some recent studies present power-efficient DCNs, which enable network components (e.g., switches and links) to consume power proportionally to the varying traffic demand [2]–[5]. With ElasticTree [2], a key enabler of power-efficient Data Center Networks (DCNs), traffic flows are consolidated on a subnet of the DCN called the *minimum-power subnet*, which is composed of the minimum number of switches and links to sustain the current network traffic demand. Thus, unused network components are turned off or put into the sleep mode to save power [6]. When the traffic demand exceeds the current subnet's capacity, more switches and links will be powered on to expand the subnet for a larger capacity.

These power-efficient DCNs usually employ Software-Defined Networking (SDN) (e.g., OpenFlow [7]) to consolidate and schedule flows. While saving power is the first priority of the power-efficient DCNs, it can be argued that practically deploying power-efficient DCNs requires an efficient flow scheduling scheme to achieve good Quality of Service (QoS) and load balancing for a minimum-power subnet. Otherwise, additional power could be wasted. For example, when a bursty traffic surges on congested routes, extra switches and links may be activated to cater to the traffic growth, increasing the power consumption of DCNs. However, existing flow scheduling schemes fail to achieve the above two goals at the same time. ElasticTree proposes *balance-oblivious flow-level scheduling* schemes to consolidate flows in the DCN without the load balancing consideration [2].

To achieve load balancing on active routes, the SDN controller must take into account the load of each flow and conduct fine-grained flow-level scheduling. In particular, when some switches and links are about to be turned off to save power, the controller has to reroute many existing flows to

maintain reachability and load balancing [2]. Since rerouting an existing flow requires the controller to generate multiple control messages to set up flow tables in the switches along this flow's old and new routes, a *control message storm* occurs if a large number of flows are rerouted simultaneously. The control message storm could impose a high processing burden on switches to install/delete/update flow entries used for flow scheduling. Current OpenFlow switches suffer from traditional hardware design and have a limited processing ability (e.g., at most processing 200 entries per second) [8], [9]. Since the minimum-power subnet must change with the time-varying traffic demand [10]–[12], switches cannot timely update their flow tables to properly schedule a large number of flows, resulting in QoS fluctuation. The above problems will be detailed in Sections II-B and II-C.

In this paper, we propose a dynamic flow scheduling scheme named AggreFlow to achieve high power efficiency and load balancing in DCNs with improved QoS. AggreFlow mainly employs three techniques listed below:

1) Flow-set routing. It aggregates flows into a small number of *flow-sets* and achieves load balancing by conducting routing in a coarse-grained flow-set fashion, which thus reduces the number of control messages for routing flows.
2) Lazy rerouting. At each time when the minimum-power subnet changes, a flow-set will not be rerouted until a packet belonging to the flow-set enters the network. Lazy rerouting amortizes the rerouting operations on flow-sets over a relatively long time, relieving switches from the control message storms. In addition, lazy rerouting reroutes a few flow-sets to maintain load balancing and allows the majority of flows still to be forwarded on their original routes. Hence, the amount of control messages for rerouting operations is significantly reduced.
3) Adaptive rerouting. The traffic in DCNs exhibits high bursty, and the load of active routes could also dynamically change. Adaptive rerouting selects some flow-sets on unbalanced routes and adjusts their routes to achieve a good load balancing performance on active routes.

We built an NS3 based fat-tree network simulation platform to evaluate AggreFlow's performance. The simulation results show that AggreFlow reduces power consumption by about 18%, and achieves load balancing and improved QoS (i.e., low packet loss rate and reduced control messages by 98%), compared with baseline schemes.

The rest of this paper is organized as follows: Section II provides the background of power-efficient DCNs and their problems. In Section III, we give an overview of AggreFlow and exemplify how it works. Section V details modules of AggreFlow. In Section VI, we evaluate AggreFlow's performance with baseline schemes. In Section VII, we discuss several issues related to AggreFlow. Section VIII reviews the related work, and Section IX concludes the paper.

II. Background and Motivation

A. Power-Efficient DCNs

Fig. 1 shows the logical structure of a power-efficient DCN, which is composed of a DCN (including servers, switches, and links) and a power consumption adapting system. The power consumption adapting system enables network components to consume power proportionally to traffic demand in the DCN by using the following two components: power optimizer and flow scheduling [2], [3]. Both components reside in an OpenFlow controller with global network information. The power optimizer component calculates the number of active network components based on current network traffic demand, configures power status of switches and links in the DCN, and notifies the flow scheduling component of the current subnet structure [2], [3]. Upon receiving the subnet structure, the flow scheduling component consolidates flows by adaptively routing and rerouting flows in the given subnet.

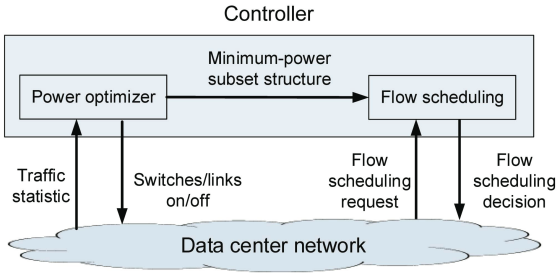


Fig. 1. Logical structure of a power-efficient DCN.

B. Imbalanced Loads on Active Routes

Fat-tree topology is the most representative topology of data center networks in real world (e.g., Baidu, Microsoft, DidiChuxing, Texas Advanced Computing Center, National University of Defense Technology, Oak Ridge National Laboratory, and Lawrence Livermore National Laboratory). ElasticTree [2] focuses on the fat-tree and proposes a simple balance-oblivious flow-level scheduling to consolidate flows. Typically, the load balancing is measured at the link level by calculating the maximum load of all links based on the traffic information, which is periodically pulled by OpenFlow from edge switches' meter entries. In a fat-tree network, the route of each flow is chosen in a deterministic left-to-right order. Only when the capacity of the leftmost route is insufficient for a flow, the second left route then will be evaluated for the flow, and so forth. Thus, the left routes could have more traffic loads than other routes, suffering from a higher chance of congestion. If a bursty traffic surges on highly loaded links, these links' loads may exceed the given threshold. Then, the controller is requested to turn on more switches and links to cater to the traffic growth, increasing the power consumption of DCNs, while other links are still under low utilization. Since the DCN traffic variation exhibits bursty [10]–[12], it could lead to unbalanced traffic allocation and degrade power efficiency. Our experiments in Section VI show the unbalanced load allocation needs about 18% more power consumption on average than the balanced load allocation.

C. QoS Fluctuation

In a subnet, we can achieve load balancing by rerouting flows to the least loaded route [13]: the OpenFlow controller uses its global network view to conduct flow-level scheduling. We name this scheme the *balance-aware flow-level scheduling*. However, flow-level scheduling schemes could impose a high burden on switches and the controller [48]. First, flow-level scheduling schemes require multiple control messages to reroute an existing flow by setting up flow tables of the

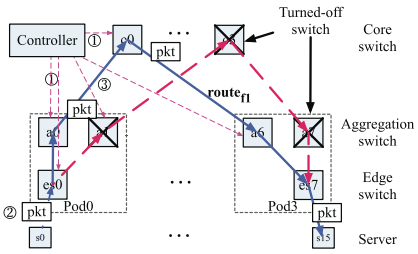


Fig. 2. Flow rerouting using the flow-level scheduling scheme in a fat-tree network. Flow $f1$ is forwarded on $route_{f1}$.

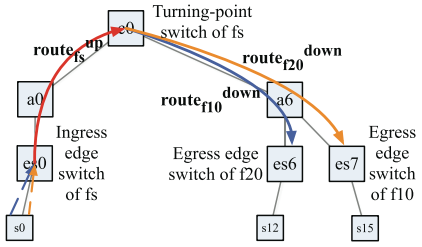


Fig. 3. An example of explanation term definitions. $f10$ and $f20$ denote two flows, and fs denotes the flow-set that includes the two flows.

switches along this flow's old and new routes. The scheduler in the SDN controller can calculate the number of control messages for flow scheduling. Fig. 2 shows an example to reroute an existing flow with flow-level scheduling schemes. Flow $f1$ is originally forwarded on route $es_0 \rightarrow a_1 \rightarrow c_3 \rightarrow a_7 \rightarrow es_7$ (red dash line). At time t_1, switches c_3 and a_7 are turned off to save power, and the controller immediately updates $f1$'s route to $route_{f1}$: $es_0 \rightarrow a_0 \rightarrow c_0 \rightarrow a_6 \rightarrow es_7$ (blue line). The rerouting operation consumes five control messages: one message to delete flow $f1$'s entry on switch a_1, one message to update flow $f1$'s entry on switch es_0, and three messages to install flow $f1$'s entries on switches a_0, c_0, and a_6. [1] At time t_2, the subsequent packets of flow $f1$ enter the DCN and are forwarded on $route_{f1}$. In the worst case (i.e., switch a_7 is not turned off in the above example), six control messages (i.e., control message to switches es_0, a_0, a_1, c_0, a_6, and a_7) are needed to reroute an existing flow.

Second, at every time the minimum-power subnet changes, flow-level scheduling schemes would reroute many flows (i.e., all flows on the soon-to-be-closed routes and many flows on routes that will remain open) to maintain reachability and load balancing. Reports show that a commercial data center can consist of millions of flows [14], [15]. To improve QoS, the rerouting operations require switches to install/delete/update entries in a very short period. We call this phenomenon the *control message storm*. The traffic variation in DCNs exhibits highly bursty [10], [11], [12], [16], [49], [50], and the subnet reconfiguration occurs consequently to save power or accommodate traffic demand variation, leading to frequent control message storms. However, current OpenFlow switches suffer from hardware design (e.g., flow entries must be organized in the TCAM in a priority order for correct and efficient matching; control messages must contend for limited bus bandwidth between a switch's CPU and ASIC [8]), and they have limited capacities to process entry update (e.g., the time of installing/updating/deleting a flow entry is usually in the order of milliseconds [8]). Therefore, the switches would not be able to timely update entries for all rerouting flows and thus degrade packet loss rate, which can be collected from servers.

D. Design Principles for Efficient Flow Scheduling Schemes

Based on the above analysis, we have the following considerations to design an efficient flow scheduling scheme for power-efficient DCNs:

1) High power efficiency: as traffic varies, flows should be dynamically consolidated and rerouted to as few links as possible so that unused switches and links can be turned off or put into to sleep mode for power saving.
2) Good load balancing: in the minimum-power subnet, a good load balancing among active routes can prevent activating extra switches and links to accommodate bursty traffic surges and save more power. Thus, we should take into account the traffic load of active routes to schedule flows.
3) Preventing control message storms: the main reason of the control message storm is that the flow-level scheduling scheme reroutes a large number of flows at the same time when the minimum-power subnet changes. To prevent control storms, we should (1) reduce the number of rerouted flows, (2) avoid conducting rerouting operations simultaneously, and (3) decrease the number of control messages for route configuration.

[1]The controller does not send control messages to delete flow $f1$'s entries at switches c_3 and a_7 because they are closed and their flow tables are emptied.

III. AggreFlow Overview

A. Term Definition

We first highlight some important terms used for AggreFlow and exemplify them in Fig. 3.

Minimum-power subnet: a subnet of the DCN that is composed of the minimum number of switches and links to sustain the current network traffic demand.

Flow-set fs: a set of flows that are aggregated together based on their hash values.

Ingress edge switch e^{in}: an edge switch that connects to a flow's source server.

Egress edge switch e^{out}: an edge switch that connects to a flow's destination server.

Forwarding route $route$: a route from a flow's ingress edge switch to its egress edge switch.

Turning-point switch ts: a switch that is at the turning point of a flow's (or flow-set's) forwarding route. In the fat-tree topology, a turning-point switch is either a core switch for inter-pod flows (which traverse different pods) or an aggregation switch for intra-pod flows (which only traverse aggregation switches in the same pod).

Upstream route $route^{up}$: an upstream route that is directed from a flow-set's ingress edge switch to its turning-point switch.

Downstream route $route^{down}$: a downstream route that is directed from a flow's turning-point switch to its egress edge switch.

Only the switches and links on the routes of flows should be activated to forward these flows. Fig. 3 shows an example. In this figure, the entire network only consists of two flows: yellow flow f10 from s_0 to s_{15} and blue flow $f20$ from s_0 to s_{12}. The two flows share the same source switch es_0, but $f10$

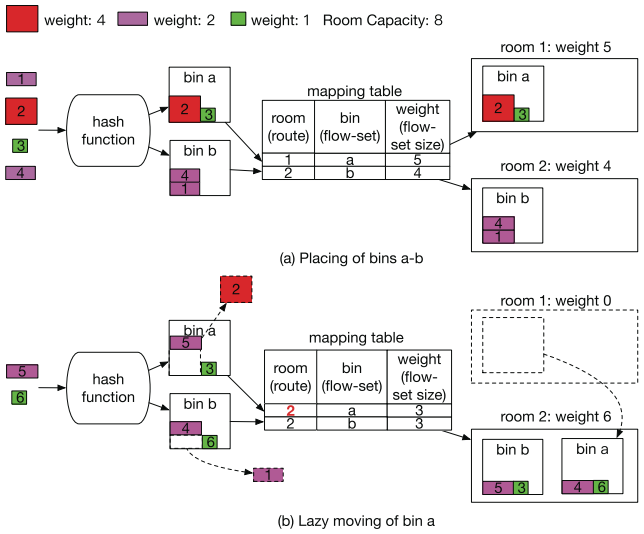


Fig. 4. Examples of flow-set routing and adaptive rerouting.

and $f20$'s destination switch is es_7 and es_6, respectively. The minimum-power subnet consists of switches es_0, a_0, c_0, a_6, es_6, and es_7, and links (es_0, a_0), (a_0, c_0), (c_0, a_6), (a_6, es_6), and (a_6, es_7). The two flows belong to the same flow-set with the turning-point switch c_0 and have the same red uplink route $route^{up}_{fs}$: $es_0 \rightarrow a_0 \rightarrow c_0$. Since the two flows are destined to different servers, $f10$'s downlink route is blue $route^{down}_{f10}$: $c_0 \rightarrow a_6 \rightarrow es_7$, and $f20$'s downlink route is yellow $route^{down}_{f20}$: $c_0 \rightarrow a_6 \rightarrow es_6$.

B. AggreFlow Techniques

In this paper, power saving has a higher priority than load balancing. To maintain high power-efficiency, a minimum subnet is first generated, which is composed of the minimum number of switches and links to sustain the current network traffic demand. For a newly generated subnet, load balancing is further achieved by dynamically rerouting certain flow-sets. When traffic demand changes, the minimum subnet may be updated to accommodate traffic variation, and then flow-sets need to be rerouted to maintain load balancing. AggreFlow employs three techniques to efficiently schedule flows: Flow-set Routing, Lazy Rerouting, and Adaptive Rerouting.

1) Flow-set Routing: Flow-set Routing conducts a coarse-grained control on flows to reduce the number of control messages for route configuration. In DCNs, we can aggregate flows with the same hash value into a flow-set, and conduct routing in a coarse-grained flow-set fashion. Once we select a route for a flow-set, the new coming flows that belong to the flow-set can be forwarded on the flow-set's route without querying the controller. In this paper, the main design trade-off is the load balancing performance and the number of flow-sets. Routing/rerouting a flow-set requires the flow entry operation of the controller to change the flow-set's route. If we use a huge number of small flow-sets, the controller can achieve better load balancing but with more flow entry operations.

Example of the Flow-set Routing: We use an example in Fig. 4 to illustrate the Flow-set Routing. Flow-set Routing can be simply viewed as a ball-bin-room mapping problem: a flow with a rate, a flow-set, and a route can be viewed as the ball with a weight, a bin, and a room, respectively; balls with different weights come and go; the goal is to maintain the balance of rooms. We first assign the balls into different bins using a hash function and then place bins into different rooms base on bin-room mappings in the mapping table to achieve load balancing of rooms. In Fig. 4, we have three types of balls (red balls with weight 4, purple balls with weight 2, and green balls with weight 1), bins a and b, and rooms 1 and 2 with the weight capacity of 8. Balls come and go at different time, and each ball arrives at a specific bin based on its hash value. In Fig. 4(a), the four balls are hashed into two bins. The weight of two bins is 9, which exceeds the room's weight capacity. Therefore, we need two rooms. To achieve load balancing of two rooms, we place bin a into room 1, and bin b into room 2. Thus, the weight of room 1 and room 2 is 5 and 4, respectively.[2]

2) Lazy Rerouting: Lazy Rerouting avoids conducting rerouting operations simultaneously. The DCN traffic analysis shows a flow's packet arrivals exhibit an ON/OFF pattern [11], [12], [15], [17]. For instance, in DCNs, there is an interval between a flow's two adjacent packets [15], [17]. Thus, every time the subnet changes, Lazy Rerouting updates the route of a ready-to-be-rerouted flow-set [3] only when a packet belonging to the flow-set enters the network. Such a rerouting spreads rerouting operations over a relatively long period of time, reducing the bursitness of flow entry operation on switches.

Example of the Lazy Rerouting: Fig. 4(b) illustrates the Lazy Rerouting. In the figure, purple ball 1 and red ball 2 leave. The two rooms' weights become 1 and 2, respectively. Since one room's weight capacity can fit all bins, we move bin a from room 1 to room 2. Purple ball 5 arrives and is placed into bin a. Sequentially, green ball 6 arrives and is assigned to bin b, and then bin a is lazily moved to room 2. Since room 1 is empty, we close it.

3) Adaptive Rerouting: Adaptive Rerouting maintains load balancing on active routes in the subnet. As flows enter and exit the network, flow-sets' sizes may vary randomly. We monitor active routes' loads and adaptively reroute some flow-sets from high-loaded routes to low-loaded routes to maintain load balancing.

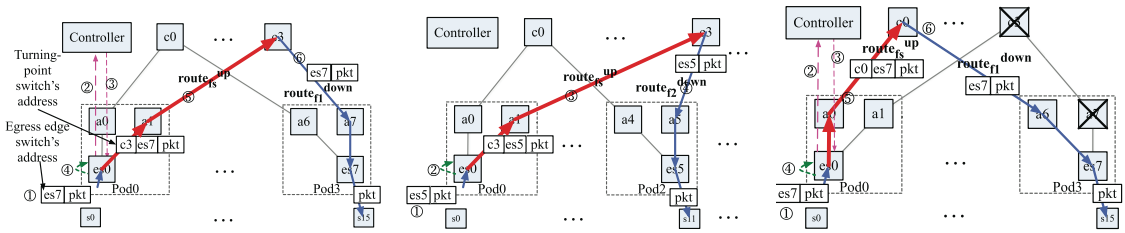
C. AggreFlow Processing Examples

In Fig. 5, we give an example that uses AggreFlow to schedule the same flow $f1$ as in Fig. 2. For simplicity, we use a switch ID to represent the switch's address in packets' headers.

1) New Flow Routing: AggreFlow routes new flows as follows: In Fig. 5(a), (1) the first packet of flow $f1$ enters the network from switch es_0. The packet's header is encapsulated with a blank header and address es_7, the address of its egress edge switch. (2) Switch es_0 cannot find flow $f1$'s flow-set, and then sends the routing request to the controller. (3) The controller informs switch es_0 of flow $f1's$ flow-set fs, which is associated with hash value h and route $es_0 \rightarrow c_3$. (4) Switch es_0 inserts address c_3 into the packet's header. (5) Using address c_3, the packet is forwarded on $route^{up}_{fs}$: $es_0 \rightarrow a_1 \rightarrow c_3$ to switch c_3. At switch c_3, the address c_3 is removed from the packet's header. (6) Using address es_7, the packet is forwarded on $route^{down}_{f1}$: $c_3 \rightarrow a_7 \rightarrow es_7$ to switch

[2] We just use the ball-bin-room example to help readers easily understand our problem. However, in some cases (e.g., different routes share some common links), our problem cannot be fully mapped to the ball-bin-room.

[3] In this paper, we use rerouting and route update interchangeably.



(a) AggreFlow routes a new flow $f1$ by creating a new flow-set fs.

(b) AggreFlow routes a new flow $f2$ by using an existing flow-set fs.

(c) AggreFlow reroutes an existing flow $f1$ by updating the route of the existing flow-set fs.

Fig. 5. Flow scheduling using AggreFlow in a fat-tree network. $route_{fs}^{up}$ is flow-set fs's route; $route_{f1}^{down}$ and $route_{f2}^{down}$ are flows $f1$ and $f2$'s downstream routes, respectively.

es_7. Switch es_7 removes the packet's header and forwards the packet to its destination server s_{15}.

In Fig. 5(b), new flow $f2$ with hash value h arrives at switch es_0. Switch es_0 finds that flow $f2$ belongs to flow-set fs and then encapsulates c_3 into the packet's header without querying the controller. After the encapsulation, flow $f2$ is forwarded on flow-set fs's $route_{fs}^{up}$ and its downstream route $route_{f2}^{down}$. In the routing procedure, the controller sends one control message to switch es_0 to initialize flow-set fs by configuring fs's route $route_{fs}^{up}$. After the initialization, no control messages are needed to configure routes for flows belonging to fs.

2) Existing Flow Rerouting: Fig. 5(c) shows a process that uses AggreFlow to reroute flow $f1$ in the same minimum-power subnet as Fig. 2. The process is explained below: At time t_1, switches c_3 and a_7 are put into sleep mode to save power, and the controller notifies all edge switches about the subnet change. (1) At time t_2, a packet of flow $f1$ enters the network (assume it is the first packet that belongs to flow-set fs and enters the network at the subnet change). (2) Switch es_0 finds that flow-set fs's route is closed and sends the rerouting request to the controller. (3) The controller tells switch es_0 to update flow-set fs's route to $es_0 \rightarrow c_0$. (4) Switch es_0 inserts address c_0 into the packet's header. (5) and (6) Packet is forwarded via updated routes $route_{fs}^{up}$ and $route_{f1}^{down}$ to destination server s_{15}.

In the rerouting procedure, AggreFlow reroutes flow-set fs at time t_2. Compared with the flow-level scheduling that conducts rerouting operation at time t_1, AggreFlow postpones the operation by a period of $t_2 - t_1$ and thus reduces switches' instant entry update overhead at t_1. Besides, the controller only sends one control message to switch es_0 to reroute flow-set fs. Assume flow-set fs contains N flows. For the worst case, flow-level scheduling consumes $N * 6$ control messages to reroute the N flows, while AggreFlow needs only one. Therefore, AggreFlow not only spreads rerouting operations over a relatively long time but also reduces the number of control messages for configuring rerouted flows.

IV. Problem Formulation

A. Network Description

The network can be described as a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ denotes the set of switches, and $\mathcal{E}$ denotes the set of directed links between switches in $\mathcal{V}$. The power consumption of link $e \in \mathcal{E}$ and switch $v \in \mathcal{V}$ are p_e and p_v, respectively. In the network operation duration $t \in [1, T]$, flow f has the following characteristics: flow rate $rate_f$, ingress edge switch's address es_f^{in}, egress edge switch's address es_f^{out}, starting time $st_f \in [1, T)$, and ending time $et_f \in (st_f, T]$. Hence, flow f can be expressed as $f = \left[rate_f, es_f^{in}, es_f^{out}, st_f, et_f\right]$. The set of edge switches is $\mathcal{ES}$. Edge switch $es \in \mathcal{ES}$ has L forwarding routes and K flow-sets. The set of es's forwarding routes is $route_{es} = \{route_{es}^1, route_{es}^2, \ldots, route_{es}^\ell, \ldots, route_{es}^L\}$, and the set of es's flow-sets is $FS_{es} = \{fs_{es}^1, fs_{es}^2, \ldots, fs_{es}^k, \ldots, fs_{es}^K\}$. Each route has the same of load, which is denoted as C. If node $v \in \mathcal{V}$ is on route $route_{es}^\ell$, $\alpha_{es}^{\ell,v} = 1$; otherwise, $\alpha_{es}^{\ell,v} = 0$. If link e is on route $route_{es}^\ell$, $\beta_{es}^{\ell,e} = 1$; otherwise, $\beta_{es}^{\ell,e} = 0$. We aggregate flows into flow-sets, and route/reroute flows in the flow-set fashion. Let $\mathcal{M}_{es}^k$ denote the set of flows assigned to flow-set fs_{es}^k. Then, at time slot t, flow-set fs_{es}^k's total rate is $rate(fs_{es}^k, t) = \sum_{f \in \mathcal{M}_{es}^k} rates_f * 1_{\{st_f \le t \le et_f\}}$. Here, $1\{.\}$ is the indicator function and is equal to one if the condition in the subscript is satisfied, otherwise zero. We use $x_{es}^{k,\ell} = 1$ to denote flow-set fs_{es}^k is assigned on $route_{es}^\ell$ (i.e., the ℓ-th route of edge switch es); otherwise, $x_{es}^{k,\ell} = 0$.

B. Constraints

1) Flow-Set Selection Constraint: Each flow-set can only be assigned to one route. Thus, we have

$$\sum_{\ell=1}^{L} x_{es}^{k,\ell} = 1, \forall\ k \in [1, K]. \tag{1}$$

2) Link Utilization Constraint: At any time slot, the load utilization of each route should not exceed its maximum utilization r. That is

$$\sum_{k=1}^{K} rate(fs_{es}^k, t) * x_{es}^{k,\ell} \le r * C,\ \forall\ \ell \in [1, L],\ \forall\ t\ \in [1, T]. \tag{2}$$

C. Objective Functions

We have three objective functions.

1) Power Consumption: The first goal is to achieve the minimum power consumption of a DCN, which consists of power consumption of links and switches. The power consumption of links/switches is proportional to the number of links/switches. Thus, we have

$$\begin{aligned} obj_1 &= p_{links} + p_{switches} \\ &= p_e * \sum_{k=1}^{K} \sum_{es \in \mathcal{ES}} \sum_{\ell=1}^{L} \sum_{e \in \mathcal{E}} \left(x_{es}^{k,\ell} * \beta_{es}^{\ell,e}\right) \\ &\quad + p_v * \sum_{k=1}^{K} \sum_{es \in \mathcal{ES}} \sum_{\ell=1}^{L} \sum_{v \in \mathcal{V}} \left(x_{es}^{k,\ell} * \alpha_{es}^{\ell,v}\right) \end{aligned}$$

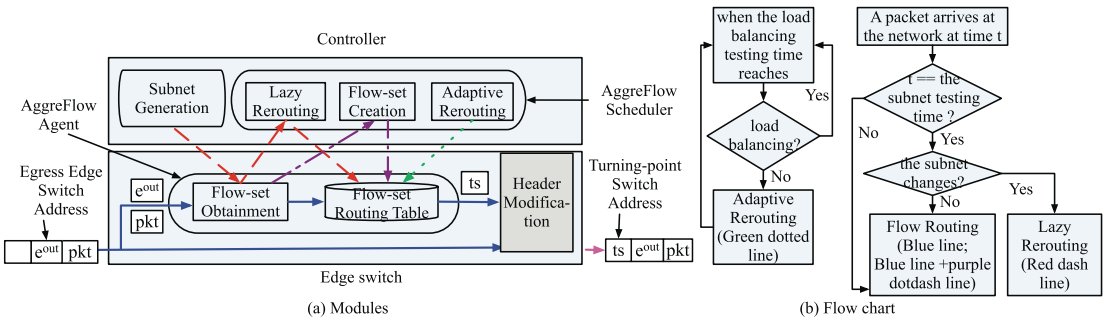


Fig. 6. AggreFlow processing procedure. Blue line: routing existing flows; blue line + purple dotdash line: routing new flows; red dash line: lazy flow-set rerouting; green dotted line: adaptive flow-set rerouting.

2) Load Balancing: The second goal is to achieve load balancing on active routes in the power-efficient DCN. The load balancing performance of an edge switch is decided by the maximum load of routes connected to the switch. Thus, our object is to minimize the maximum load utilization of routes connected to an edge switch by appropriately assigning flow-sets during at time $t \in [1, T]$. The maximum utilization of active routes at time t is

$$obj_2 = r$$

3) QoS: The QoS is measured by two metrics: the number of control messages and packet loss rate. The QoS degradation occurs when flow-sets are rerouted. Thus, if we can reduce the number of rerouting flow-sets, we can mitigate QoS degradation. We use $\tilde{x}_{es}^{k,\ell}$ to denote the selected route of flow-set fs_{es}^{k} in the last time slot. The number of changed route for all flow-sets can be formulated as follows:

$$obj_3 = \sum_{k=1}^{K} \sum_{es \in \mathcal{ES}} \sum_{\ell=1}^{L} \left| x_{es}^{k,\ell} - \tilde{x}_{es}^{k,\ell} \right|$$

D. Problem Formulation

The goals of our problem is three folded: (1) generating the minimum-power subnet, which minimizes the power consumption of DCN and accommodates to traffic load; (2) achieving load balancing on active routes of each edge switch in the minimum-power subnet; and (3) maintaining good QoS by appropriately selecting the minimum number of links and switches and assigning flow-sets to the active routes at each time $t \in [1, T]$. The problem is formulated as follows:

$$\min_{x,r} \ (w_1 * obj_1 + w_2 * obj_2 + w_3 * obj_3)$$

subject to

$$\text{Eqs.}(1), (2),$$
$$x_{es}^{k,\ell} \in \{0, 1\}, \ r \in [0, 1], \ k \in [1, K], \ \ell \in [1, L],$$

where all $\left\{x_{es}^{k,\ell}\right\}$ and r are design variables, $w_1, w_2, w_3 > 0$ are three constants that give different weights of the objectives. The problem is a mixed integer linear programming, which is generally NP-hard.

V. AggreFlow Design

In this section, we detail AggreFlow's processing procedure and its modules.

A. AggreFlow Structure and Processing Procedure

Fig. 6 shows AggreFlow's structure and processing procedure. AggreFlow consists of three components: subnet generation, AggreFlow scheduler, and AggreFlow agents. Subnet generation considers the topology and traffic to determine the number and location of active switches and links in the DCN. With the subnet's structure, AggreFlow scheduler and AggreFlow agents work together to efficiently schedule flows.

The input of an AggreFlow agent is an AggreFlow packet (including the address of the egress edge switch address e^{out} and a packet), and the output is the route of a flow-set that the packet belongs to. The route of a flow-set is stored in the flow-set routing table in the form of the address of the flow-set's turning-point switch ts. [4] The address of ts is then encapsulated into the AggreFlow packet. The addresses of two switches indicate the packet's upstream and downstream routes. Switches use the two headers to forward the packet in the DCN. In Fig. 6, each packet contains two headers when it enters a switch. In the DCN, each server is equipped with a system that stores the mapping relationship between servers and edge switches connected to the servers. Thus, before a packet leaves its source server, it is encapsulated with its header e^{out}. Considering MTU packets injected in the DCN cannot easily be expanded, we insert a blank header in advance and will change it to address ts after the processing. The header ts is selected by the AggreFlow scheduler. The details are shown in Sections V-C, V-E, and V-D.

In Fig. 6, AggreFlow has four processing cases: (1) blue line: routing existing flows; (2) blue line + purple dotdash line: routing new flows; (3) red dash line: lazily rerouting flows; (4) green dotted line: adaptively rerouting flows. In case (1), an AggreFlow agent calls FlowsetObtainment function to find the flow-set of a packet from its flow-set routing table and then encapsulates the route of the flow-set into the packet. In case (2), if the AggreFlow agent cannot find the flow-set for a packet, it will ask the controller, which will call FlowsetCreation function to create a new flow-set for the packet. In case (3), if a new subnet is generated, the scheduler conveys the change to AggreFlow agent in each edge switch, which will call LazyRerouting function via FlowsetObtainment function to change the route of the flow-set when it receives flows. In case (4), when the load balancing performance does not satisfy the requirement, the controller uses AdaptiveRerouting function to reroute some flow-sets to improve the performance.

B. Subset Generation

A fat-tree topology exhibits high regularity. Particularly, links have the same capacity, switches have the same size, and

[4] In this paper, we use a turning-point switch ts and a flow-set's route interchangeably.

Algorithm 1 FlowsetObtainment($packet_f^{AF}$)

Input: $packet_f^{AF}$: an AggreFlow packet of flow f that enters the network from edge switch es;
Output: $route(fs_{es}^k)$: the route of flow-set fs_{es}^k.
1: k = Hash($packet_f^{AF}$);
// $Table_{es}$: flow-set routing table of edge switch e;
2: fs_{es}^k = FlowsetSearch($k, Table_{es}$);
// $fs_{es}^k : \left[rate(fs_{es}^k, t), route(fs_{es}^k)\right]$;
// $route_{es}$: the set of activated routes connected to edge switch e;
3: **if** $fs_{es}^k == \varnothing$ **then**
4: $route(fs_{es}^k)$ = FlowsetCreation($k, Table_{es}, route_{es}$); // Algorithm 2
5: **else if** $fs_{es}^k \neq \varnothing$ and $IsSubnetChanged == TRUE$ and $ChangeRoute(fs_{es}^k) == TRUE$ **then**
6: $route(fs_{es}^k)$ = LazyRerouting($fs_{es}^k, Table_{es}, route_{es}$); // Algorithm 3
7: **end if**
8: return $route(fs_{es}^k)$.

the topology is regular. We can take advantage of the regularity of a fat-tree network to determine whether to turn on/off switches or links with much less computational burden. Specifically, in the fat-tree network, we activate all edge switches to accommodate traffic from servers. In each pod, the number of active aggregation switches equals to the number of active links that support the aggregated uplink and downlink traffic. For example, in Fig. 2, assume the rate of each link is 1 Gbps, and the link's safety margin (i.e., additional capacity beyond normal levels to handle unpredictable traffic surges) is 0.8 Gbps. If edge switch es_0 sends 1.5 Gbps of traffic up to the aggregation layer over two links, we must enable aggregation switches a_0 and a_1 to satisfy that demand. Similarly, the number of active core switches can be calculated based on the aggregated traffic between aggregation switch layer and core switch layer. Existing works (e.g., ElasticTree [2], Carpo [3]) proposed solutions to efficiently generate the minimum subnet, and we use the solution similar to ElasticTree in our work.

C. Flow-Set Routing

Flow-set Routing is achieved by Flow-set Obtainment and Flow-set Creation modules. Flow-set Obtainment module routes a packet by obtaining the flow-set that has the same hash value with the packet. If a packet does not belong to any exiting flow-sets, Flow-set Creation module will assign a flow-set to the flow, and the flow-set's route becomes the flow's route.

1) Flow-Set Obtainment: Algorithm 1 shows the pseudo code of Flow-set Obtainment. The algorithm works for each flow that arrives at network. In line 1, edge switch e receives an AggreFlow packet of flow f and uses hash function Hash() to compute flow f's hash value h_f. We use the hash function because of its consistency, randomness and efficiency. A hash function can always map packets of a flow to the same flow-set and provide the trade-off between search time and data storage space. The computation includes two steps: (1) using CRC32 checksum algorithm to hash flow f's packet $packet_f$'s five tuples (i.e., source IP address, destination IP address, source port number, destination port number, and protocol field), (2) doing the mod operation on the result of the first step with K, where K is the capacity of the flow-set routing table $Table_{es}$ at edge switch e, [5] and an entry stores the route and load of a flow-set. The route of a flow-set is stored in the form of its turning-point switch address, and the load of a flow-set records the number of packets that hit a flow-set entry in a period of time. It is used by Lazy Rerouting and Adaptive Rerouting modules to achieve load balancing. When the idle timeout of an entry in the flow-set routing table expires, it will be removed. Here we assume h_f equals k. In line 2, we use k to find flow f's flow-set fs_{es}^k from flow-set routing table $Table_{es}$.

Lines 3 to 4 handle the case that flow f is a new flow that does not belong to any existing flow-sets. If flow-set fs_{es}^k's route does not exist, switch e will request the scheduler. The scheduler calls Flow-set Creation module (Algorithm 2) to create a new flow-set fs_{es}^k for flow f and assigns a turning-point switch for the flow-set based on the current network status.

Lines 5 to 7 concern the case that the minimum-power subnet has changed, and the routes of flow-sets should be updated. In line 5, $IsSubnetChanged$ is a Boolean variable with default value $FALSE$, indicating an unchanged subnet. When the subnet changes, the scheduler sends the message $IsSubnetChanged = TRUE$ to each edge switch. Upon receiving the message, edge switches change Boolean variable $ChangeRoute$ of each flow-set to $TRUE$, which states a flow-set's route should be updated. If both Boolean variables $IsSubnetChanged$ and $ChangeRoute(fs_{es}^k)$ are $TRUE$, Lazy Rerouting module (Algorithm 3) is called to change its flow-set fs_{es}^k's route based on the current network status. In line 8, after the above process, flow-set fs_{es}^k's route $route(fs_{es}^k)$ is returned, and the turn-point switch in $route(fs_{es}^k)$ is encapsulated into $packet_f^{AF}$.

Algorithm 2 FlowsetCreation($k, Table_{es}, route_{es}$)

Input: k: $packet_f^{AF}$'s hash value;
$Table_{es}$: flow-set routing table of edge switch es;
$Route_{es}^{active}$: the set of active routes connected to edge switch es;
Output: $route(fs_{es}^k)$: the route of flow-set fs_{es}^k.
1: $route_f$ = LeastLoadRoute($Route_{es}^{active}$);
2: $rate(fs_{es}^k, t) = 0$, $route(fs_{es}^k) = route_f$, $fs_{es}^k = \left[rate(fs_{es}^k, t), route(fs_{es}^k)\right]$;
3: $Table_{es} \leftarrow Table_{es} \bigcup (k, fs_{es}^k)$;
4: $ChangeRoute(fs_{es}^k) = FALSE$;
5: return $route(fs_{es}^k)$.

2) Flow-Set Creation: Algorithm 2 describes the pseudo code of Flow-set Creation module. In line 1, the least loaded route in $Route_{es}^{active}$, the set of active routes connected to edge switch e, is selected as the route of flow f. In line 2, flow-set fs_{es}^k is generated. In fs_{es}^k, flow-set fs_{es}^k's rate counter $rate(fs_{es}^k, t)$ is initialized to 0, and flow-set fs_{es}^k's route equals to flow f's route. In line 3, flow f's hash value k is mapped to flow-set fs_{es}^k, and this mapping is stored in flow-set routing table $Table_{es}$. In line 4, $ChangeRoute(fs_{es}^k)$

[5] In this paper, we use a flow-set and an entry in the flow-set routing table interchangeably.

Algorithm 3 LazyRerouting($fs_{es}^k, Table_{es}, Route_{es}^{active}$)

Input: $Table_{es}$: flow-set routing table of edge switch es;
fs_{es}^k: the k-th flow-set of edge switch es;
$Route_{es}^{active}$: the set of active routes connected to switch es;
Output: ts_{es}^k: the turning-point switch address of flow-set fs_{es}^k that contains flow f.

1: $new\_route(fs_{es}^k) = \varnothing$;
// $route(fs_{es}^k) : es \rightarrow ts_{es}^k \rightarrow e_f^{out}$;
2: **if** ($route(fs_{es}^k) \notin Route_{es}^{active}$) **then** // close existing routes
3: $new\_route(fs_{es}^k) =$ LeastLoadRoute($route_{es}$);
// $new\_route(fs_{es}^k) : es \rightarrow new\_ts \rightarrow e_f^{out}$;
4: $load_{new\_route(fs_{es}^k)} = load_{new\_route(fs_{es}^k)} + rate(fs_{es}^k, t)$;
5: $ChangeRoute(fs_{es}^k) = FALSE$;
6: **else if** ($load_{route(fs_{es}^k)} > load_{es}^{ave}$ and ReroutingDecision($route(fs_{es}^k)$) $== TRUE$) **then** // open new routes
7: $new\_route(fs_{es}^k) =$ LeastLoadRoute($route_{es}$);
// $new\_route(fs_{es}^k) : es \rightarrow new\_ts \rightarrow e_f^{out}$;
8: $load_{new\_route(fs_{es}^k)} = load_{new\_route(fs_{es}^k)} + rate(fs_{es}^k, t)$,
$load_{route(fs_{es}^k)} = load_{route(fs_{es}^k)} - rate(fs_{es}^k, t)$;
9: $ChangeRoute(fs_{es}^k) = FALSE$;
10: **end if**
11: **if** $new\_route(fs_{es}^k) \neq \varnothing$ **then**
12: $route(fs_{es}^k) = new\_route(fs_{es}^k)$;
13: **end if**
14: return $route(fs_{es}^k)$.

changes to $FALSE$, indicating that flow-set fs_{es}^k's route is updated. In line 5, flow-set fs_{es}^k's route is returned to Flow-set Obtainment module.

3) Flow-set Routing's Advantages: Flow-set routing not only reduces the controller's routing load but also maintains low latency. First, it reduces the route establishment for multiple flows. The forwarding route of a flow-set is established only when the first packet of the first flow in the flow-set enters the network. Once the route is created, the rest packets of the flow and the following flows belonging to the flow-set can be immediately forwarded. Second, the cost of establishing a route is minimized. The route creation/update only requires an action on an edge switch, and other entries in aggregation and core switches are proactively installed. Third, flow-set routing requires a switch to do the hash and table lookup, which are basic packet processing functions in the switch and do not incur much latency.

D. Lazy Rerouting

1) Solution: Algorithm 3 describes the pseudo code of Lazy Rerouting module. In line 1, we initialize its new route $new\_route(fs_{es}^k)$ as empty. Lines 2 to 5 are concerned with the case that the existing route of flow-set fs_{es}^k is closed. When the minimum-power subnet changes to save power, the existing route does not exist in the set of active routes $Route_{es}^{active}$. Thus, in line 3, the least loaded route in $Route_{es}^{active}$ is selected as the flow-set's new route. In line 4, the traffic load of flow-set fs_{es}^k is added to the traffic load of its new route. In line 5, after the route update, $ChangeRoute(fs_{es}^k)$ changes to $FALSE$ to state the flow-set's route has been updated.

Lines 6 to 9 handle the case that one or more new routes are available for flow-set fs_{es}^k. When the minimum-power subnet changes to accommodate to the increasing traffic demand, flow-set fs_{es}^k's route will be updated if it meets two requirements listed below: (1) the load of $route(fs_{es}^k)$ exceeds the balancing counter threshold $load^{ave}$; (2) flow-set fs_{es}^k satisfies the rerouting probability: ReroutingDecision($route(fs_{es}^k)$) $== TRUE$. Requirement (1) ensures each active route with approximately equal traffic load. The balancing counter threshold $load^{ave}$ is the average traffic load of active routes connected to e, and it is calculated by the sum of traffic loads of the active routes connected to e divided by the number of the active connected to e.

Requirement (2) prevents traffic starvation on existing-activated routes during flow-set rerouting. If we only consider the first requirement to reroute flow-sets, flow-sets are kept rerouting to the newly activated route(s) until the traffic loads of those newly activated routes reach the balancing counter threshold. Under such a situation, a few existing-activated routes could have no traffic for a transient time and experience traffic starvation, leading to a short time load balancing performance degradation. In order to prevent the undesired situation, ReroutingDecision function sets the probabilities for rerouting flow-sets from their original existing-activated routes to newly activated routes. The rerouting probability of flow-sets on an existing-activated route equals the number of newly activated routes divided the total number of activated routes. A flow-set is rerouted only when it is selected by ReroutingDecision function. Fig. 7 shows examples that compare traffic on routes without and with the RerouteDecision function. In Fig. 7(a), edge switch e has one existing route $route0$ and a new route $route1$. After the new route is activated, four existing flows arrive the edge switch. In Fig. 7(b), all the four flows (and their flow-sets) are rerouted to $route1$, and $route0$ does not have traffic for a while, resulting in unbalancing traffic loads on the two routes. In Fig. 7(c), we use the RerouteDecision function and set each flow-set with 50% rerouting probability. Thus, two flows (and their flow-sets) are rerouted to $route1$ based on the result of ReroutingDecision function, and other two flows (and their flow-sets) are still forwarded on $route0$. Thus, the traffic loads on $route0$ and $route1$ are still statistically balanced during flow-set rerouting.

If flow-set fs_{es}^k meets all the two requirements, the least loaded route in $route_{es}$ is selected as the flow-set's new route. The load of flow-set fs_{es}^k is removed from $route(fs_{es}^k)$ to $new\_route(fs_{es}^k)$, and $ChangeRoutefs_{es}^k$ changes to $FALSE$ to state flow-set's route has been updated. In lines 11 to 13, if $new\_route(fs_{es}^k)$ is selected, it is updated to $route(fs_{es}^k)$. In line 14, $route(fs_{es}^k)$ is sent back to Flow-set Obtainment module.

2) Performance analysis: In this subsection, we first introduce one theorem for deterministic load balancing, and then introduce and prove one theorem for our lazy rerouting.

(1) Deterministic load balancing. For the load balancing with deterministic variables, Graham has proven the following rule [18]:

Graham's rule *All jobs are given size and arranged in an arbitrary order. These jobs are assigned one by one and allocated to the bin which has currently the smallest load. This scheme's performance is a* $2 - 1/L$ *approximation of the*

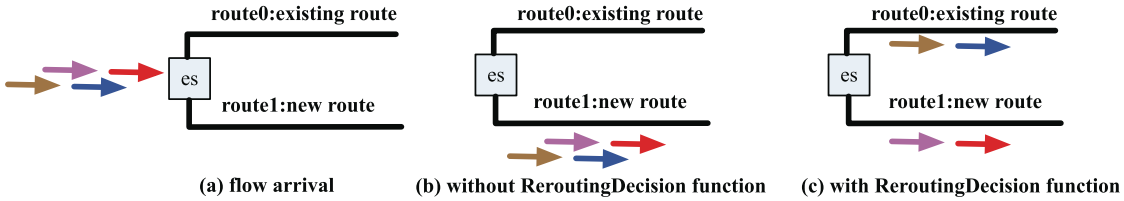


Fig. 7. Examples of not using and using ReroutingDecision function.

optimum value of deterministic load balancing problem, where L denotes the number of bins.

Replacing job and bin with flow-set and route in the above theorem, we can have Theorem 1.

Theorem 1 *All flow-sets are given size and arranged in an arbitrary order. These flow-sets are assigned one by one and allocated to the route which has currently the smallest load. This scheme's performance is a $2-1/L$ approximation of the optimum value of deterministic load balancing problem, where L denotes the number of routes.*

(2) Stochastic load balancing. Given the variation of flow's size, the goal of Lazy Rerouting is to realize the load balancing with variables following stochastic distribution. This problem is usually called *stochastic load balancing*. We assume the size of each flow follows Poisson distribution. A flow-set consists of multiple flows, and its size also follows Poisson distribution. Our stochastic load balancing problem can be stated as follows:

K flow-sets are allocated on L routes. The size of the k-th flow-set is $rate^k$, where $rate^k$s are independent random variables and follow Poisson distribution. $z_{kl}=1$ denotes k-th flow-set is assigned to $route_l$; otherwise $z_{kl}=0$. Our goal is to allocate flow-sets to routes such that the expected value of the maximum load on a route is minimized. The objective function can be mathematically formulated as follows:

$$\min \mathbf{E}\left[\max_{l\in[1,L]}\sum_{k=1}^{K}(rate_k * z_{kl})\right]$$

Lazy Rerouting is to place the flow-set one by one to the least loaded route to realize the stochastic load balancing with variables following Poisson distribution. The natural equivalent of Theorem 1 for Poisson variables is to replace each variable by a deterministic variable with the same mean. Here we first introduce one definition and prove some lemmas from [19] to analyze the performance of Lazy Rerouting.

Definition 1: X and Y are two random variables. X is said to stochastically dominate Y (i.e., $Y \leq_{\mathrm{sd}} X$) if for each x, $\mathbf{Pr}[Y \leq x] \geq \mathbf{Pr}[X \leq x]$.

Lemma 2.1: Let $x_1 > x_2$ *and* $2\delta \leq x_1 - x_2$. *Let* $M_1 = \max\left(\mathcal{P}\left(x_1-\delta\right), \mathcal{P}\left(x_2+\delta\right)\right)$ *and* $M_2 = \max\left(\mathcal{P}\left(x_1\right), \mathcal{P}\left(x_2\right)\right)$. *Then,* $M_1 \leq_{sd} M_2$.

Proof: Let $M(x,y) = \max(\mathcal{P}(x), \mathcal{P}(y))$ and $f_t(x) = \mathbf{Pr}[\mathcal{P}(x) \leq t]$. Then $\mathbf{Pr}[M(x,y) \leq t] = f_t(x)f_t(y) = e^{\log f_t(x) + \log f_t(y)}$. We now show that $\log f_t(x)$ is concave of x. Recall that $f_t(x) = e^{-x}\sum_{k=0}^{t}\frac{x^k}{k!}$. A simple computation shows that

$$\frac{\mathrm{d}\log f_t(x)}{\mathrm{d}x} = \frac{\frac{\mathrm{d}f_t(x)}{\mathrm{d}x}}{f_t(x)} = -\frac{x^t/t!}{\sum_{k=0}^{t}\left(x^k/k!\right)}$$

However, $\frac{x^t/t!}{\sum_{k=0}^{t}(x^k/k!)}$ is the Erlang-B formula [20], which is known to be monotonically increasing in x. Hence $\frac{\mathrm{d}\log f_t(x)}{\mathrm{d}x}$ is monotonically decreasing. Thus, $\log f_t(x)$ is concave of x. If $x+y$ is fixed, $\log f_t(x) + \log f_t(y)$ increases as $|y-x|$ decreases. Therefore, $\mathbf{Pr}\left[M_1 \leq t\right] \geq \mathbf{Pr}\left[M_2 \leq t\right]$ for all t. With Definition 1, we can have $M_1 \leq_{\mathrm{Sd}} M_2$. □

Let S_l be the sum of the means of the size of flow-set allocated to $route_l$. Let x_l be the size of the last flow-set allocated to $route_l$. Let $y_l = S_l - x_l$. Let M^* represent the optimum value of the objective function. Define $\bar{\mu} = \frac{\sum_{k=1}^{K}\mu_k}{L}$ where μ_k denotes the size of k-th flow-set. From the definition, it follows that $y_l \leq \bar{\mu}$.

Lemma 2.2: $\mathbf{E}\left[\max\left(\mathcal{P}\left(y_1\right), \ldots, \mathcal{P}\left(y_L\right)\right)\right] \leq M^*$

Proof: With Lemma 2.1 and Expectations of maximum order statistics, we can have $\mathbf{E}[\max(\mathcal{P}(\bar{\mu}), \mathcal{P}(\bar{\mu}), \ldots,$ repeated L times $)] \leq M^*$. Further, $y_l \leq \bar{\mu} \Rightarrow \mathcal{P}\left(y_l\right) \leq_{\mathrm{Sd}} P(\bar{\mu})$, or $\mathbf{E}\left[\max\left(\mathcal{P}\left(y_1\right), \ldots, \mathcal{P}\left(y_L\right)\right)\right] \leq \mathbf{E}[\max(\mathcal{P}(\bar{\mu}), \mathcal{P}(\bar{\mu}), \ldots$ repeated L times $)] \leq M^*$. □

Lemma 2.3: $\mathbf{E}\left[\max\left(\mathcal{P}\left(x_1\right), \ldots, \mathcal{P}\left(x_L\right)\right)\right] \leq M^*$

With the above lemmas, we can have the following theorem for the Lazy rerouting:

Theorem 2: Each flow-set's size follows Poisson distribution, and all flow-sets are arranged in an arbitrary order. These flow-sets are assigned one by one and allocated to the route which has currently the smallest load. The lazy rerouting's performance is a 2 approximation for the optimum value of the stochastic load balancing problem with Poisson variables.

Proof: $\mathbf{E}\left[\max\left(\mathcal{P}\left(S_1\right), \ldots, \mathcal{P}\left(S_L\right)\right)\right]$
$= \mathbf{E}\left[\max\left(\mathcal{P}\left(x_1+y_1\right), \ldots, \mathcal{P}\left(x_L+y_L\right)\right)\right]$
$\leq \mathbf{E}\left[\max\left(\mathcal{P}\left(x_1\right), \ldots, \mathcal{P}\left(x_L\right)\right)\right]$
$+\mathbf{E}\left[\max\left(\mathcal{P}\left(y_1\right), \ldots, \mathcal{P}\left(y_L\right)\right)\right]$
$\leq 2\ M^*$ □

3) Lazy Routing's Latency: When the minimum-power subnet changes, lazy rerouting sends the rerouting request for the first packet of a to-be-rerouted flow-set to the controller, which needs one control message to change the route. The rerouting request process increases latency. However, a flow-set can consist of hundreds or thousands of flows, each of which can have hundreds or thousands of packets, the increased latency due to lazy rerouting only applies to the first packet of each flow-set after rerouting and therefore is very minor.

4) Discussion: In our problem, we consider that the processing time of a flow is not known in advance, and flows come and go randomly. Since a flow-set consists of multiple flows, it may always have some flows for processing and does not have the end time of processing. Thus, we cannot use existing flow-based processing time algorithms (e.g., Longest Processing Time rule) to schedule flow-sets. Under this condition, lazy rerouting is a good solution since it greedily assigns flow-sets one by one to the route, which has currently the smallest load, and can achieve $2-1/L$ approximation of the optimum value of deterministic load balancing problem, where L denotes the number of routes.

E. Adaptive Rerouting

Adaptive Rerouting module monitors the traffic load of active routes and adaptively reroutes some flow-sets to main-

tain load balancing. This module solves the Flow-set Load Balancing Problem (FSLB) to find the mappings between flow-sets and routes. Next, we present the problem formulation and our solution.

1) Problem Formulation: The goal of FSLB problem is to achieve load balancing on active routes of each edge switch in the minimum-power subnet by appropriately assigning flow-sets to the active routes at each time $t \in [1, T]$. The load balancing performance of an edge switch is decided by the maximum load of routes connected to the switch. Thus, our object is to minimize the maximum load of routes connected an edge switch. The problem is formulated as follows:

$$\min_{x,r} r$$

subject to

$$\text{Eqs.}(1), (2)$$
$$x_{es}^{k,\ell} \in \{0, 1\},\ r \in [0, 1],\ k \in [1, K],\ \ell \in [1, L],$$

where all $\{x_{es}^{k,\ell}\}$ and r are design variables.

2) Problem Solution: The above FSLB problem is a binary linear program, which is generally NP-hard. To efficiently solve the problem, we can transform the problem into the Simplified FSLB (SFSLB) by using the linear programming relaxation technique.

The solution of the SFSLB problem are a set of decimal values, denoted x_{es}. We use a customized rounding technique to get the final binary solution. We first sort x_{es} in the descending order of its values. For each $x_{es}^n \in x_{es}$, we find its flow-set fs_{es}^k, the flow-set's previous route $route_{es}^{\ell_0}$, and its new route $route_{es}^{\ell_1}$. If the flow-set is not rerouted, and its flow-set's new route can accept the flow-set to improve its load balancing performance, the flow-set is decided to rerouted to the new route; otherwise, the flow-set is still forwarded on its previous route.

3) Adaptive Rerouting Algorithm: Algorithm 4 describes the pseudo code of the Adaptive Rerouting module. The traffic load of active route $route_{es}^{\ell}$, denoted by $load_{es}^{\ell}$, is periodically calculated and equals the total traffic load of flow-sets on the route. AggreFlow scheduler can use OpenFlow meters to periodically pull flow entries' meters from edge switches [21] to get the traffic.

In line 1, we sort $route_{es}$ in the ascending order of the route's load. Lines 2 to 13 evaluate the load balancing performance of routes in $route_{es}$. In lines 3 to 8, if the load of a route lies in the interval $[load_{es}^{Ave} - load_{es}^{Thd}, load_{es}^{Ave} + load_{es}^{Thd}]$, we think that the route has a good performance and will check the next route; otherwise, based on the route's load, we put the route into $Route_{es}^{OverThd}$, the set of routes whose loads exceed the average load, or $Route_{es}^{BelowThd}$, the set of routes whose loads are lower than the average load. Using $load_{es}^{Thd}$ can guarantee the performance in a small range and prevent the frequent flow-set rerouting, which comes from achieving the absolute load balancing on active routes.

In lines 9 to 10, if this switch's load balancing performance is good, we do not need to reroute any flow-sets on it. Otherwise, in lines 12 to 13, we generate Ω_{es}, the set of routes to be balanced. If the number of routes in $Route_{es}^{OverThd}$ and $Route_{es}^{BelowThd}$ are significantly different, it is difficute to reroute flow-sets to maintain load-balancing. Thus, we generate the auxiliary sets of routes to solve the problem. In line 12, we generate $Route_{es}^{*BelowThd}$ for $Route_{es}^{OverThd}$ and $Route_{es}^{*OverThd}$ for $Route_{es}^{BelowThd}$. Fig. 8 shows an example of generating the set of routes Ω. In the figure, if we do not have $Route_{es}^{*BelowThd}$, we only have one route and cannot reroute flow-sets in the route.

Algorithm 4 AdaptiveRerouting(es)

Input: es: edge switch es

1: sort $route_{es}$ in the ascending order of the route's load;
2: **for** $route_{es}^{\ell} \in route_{es}$ **do**
3: **if** ($load_{es}^{\ell} > load_{es}^{Ave}$ + $load_{es}^{Thd}$) **then**
4: $Route_{es}^{OverThd} \leftarrow Route_{es}^{OverThd} \cup route_{es}^{\ell}$;
5: **else if** $load_{es}^{\ell} < load_{es}^{ave} - load_{es}^{thd}$ **then**
6: $Route_{es}^{BelowThd} \leftarrow Route_{es}^{BelowThd} \cup route_{es}^{\ell}$;
7: **end if**
8: **end for**
9: **if** ($Route_{es}^{OverThd} == \emptyset$ and $route_{es}^{BelowThd} == \emptyset$) **then**
10: return;
11: **end if**
12: $Route_{es}^{*BelowThd} = \{route_{es}^{\ell} | \ell \in [1, |Route_{es}^{OverThd}|]\}$, $Route_{es}^{*OverThd} = \{route_{es}^{\ell} | \ell \in [|Route_{es}^{BelowThd}|, L]\}$;
13: $\Omega_{es} = Route_{es}^{OverThd} \cup Route_{es}^{BelowThd} \cup Route_{es}^{*BelowThd} \cup Route_{es}^{*OverThd}$;
14: $\Gamma_{es} = \{fs_{es}^{k} | fs_{es}^{k}$ on $route_{es}^{\ell} \in \Omega_{es})$, $ChangeRoute(fs_{es}^{k}) == True\ (fs_{es}^{k} \in \Gamma_{es})$;
15: $x_{es} = \{x_{es}^{n}, n \in [1, |\Gamma_{es}| * L]\}$ is obtained by solving the SFSLB problem for Γ_{es} and sorting the results in the descending order;
16: **for** $x_{es}^{n} \in x_{es}$ **do**
17: find x_{es}^{n}'s flow-set fs_{es}^{k}, new route $route_{es}^{\ell_1}$, and previous route $route_{es}^{\ell_0}$;
18: **if** $route_{es}^{\ell_1} == route_{es}^{\ell_0}$ **then**
19: $ChangeRoute(fs_{es}^{k}) = FALSE$, continue; // test the next route
20: **end if**
21: **if** $ChangeRoute(fs_{es}^{k}) == TRUE$ and ($load_{es}^{\ell_1} + rate(fs_{es}^{k}, t) < load_{es}^{ave}$ + $load_{es}^{thd}$) **then**
22: $load_{es}^{\ell_1} = load_{es}^{\ell_1} + rate(fs_{es}^{k}, t)$, $load_{es}^{\ell_0} = load_{es}^{\ell_0} - rate(fs_{es}^{k}, t)$,
23: **end if**
24: **end for**
25: update the routes of flow-sets in Γ_{es} on edge switch es;

After getting the set of routes for rerouting, we will decide how to reroute flow-sets to improve the load balancing performance. In line 14, we generate Γ_{es}, the set of flow-sets of routes in Ω_{es}, and each flow-set in Γ_{es} should update its corresponding route. In line 15, we generate the set of decimal results. In lines 16 to 24, we use a customized rounding technique to get the final rerouting result and update the load of related routes similar to the lines 7 to 9 in Algorithm 3. In line 25, the new routes of flow-sets are updated to the switch.

VI. Simulation

This section evaluates AggreFlow's performance in the fat-tree network simulation platform.

A. Comparison Schemes

We compare AggreFlow with two flow-level scheduling schemes.

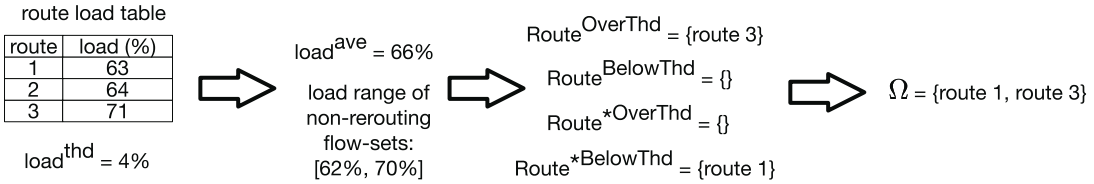


Fig. 8. An example of generating the set of routes Ω.

Balance-oblivious flow-level scheduling [2]: The controller consolidates every flow to the leftmost route with sufficient capacity for the flow. For a single flow, a routing operation or a rerouting operation requires multiple control messages from the controller.

Balance-aware flow-level scheduling [7]: In a given minimum-power subnet, the controller routes each new flow to the least loaded route. As the subnet changes, the controller reroutes existing flows to the least loaded route one by one. Section II-C explains its details.

AggreFlow: AggreFlow employs Flow-set Routing, Lazy Rerouting, and Adaptive Rerouting to efficiently schedule flows. The details are described in Section III-B. We let T(K) denote a flow-set routing table with K entries and AggreFlow(K) denote AggreFlow scheme using T(K).

B. Simulation Setup

We designed a packet-mode simulation platform on a NS-3 based fat-tree testbed. In the DCN, each link has the same rate, and each server sends a certain number of flows to all other servers. To emulate flow arrivals and terminations, each flow is given two states: ON and OFF. The ON status of a flow lasts a duration with an exponentially distributed random variable, which is determined when the flow is generated. The OFF status of a flow is the idle time of the flow and also lasts a duration of an exponentially distributed random variable, decided when the previous ON status finishes. The power of the DCN is the total power consumed by active switches and links/ports.

Generally speaking, DCNs usually incorporate some level of capacity safety margin to prepare for traffic surges [2]. In such cases, the network could allocate more capacity than essential for normal workload. To implement capacity safety margin ϕ, we monitor the utilization of each outgoing port/link of a switch. A new port on the same side of the switch will be enabled when the utilization exceeds 1-ϕ. Then, the corresponding port in another switch will be activated to establish the new link.

In our simulation, we use 3-layer 32-pod fat-tree network. Each link's rate is 1 Gbps, and the size of each packet is 1.5 KB. Each output port of a switch has a buffer space of 1,200 KB. The average ratio of ON period to OFF period is 5 [6]. Each flow is an inter-pod flow. The power status change of core switches causes flow rerouting. Traffic flows are generated in two separate slot intervals. In the first interval (0,60), each flow's arrival slot is a random variable in slot interval (0,40), and its termination slot is a random variable in slot interval (40,60). In the second slot interval (60,124), each flow's arrival slot is a random variable in interval (60,100), and its termination slot is a random variable in slot interval (100,124). No new flows are generated in slot intervals (40,60) and (100,124). Fig. 9 shows the DCN's utilization in our simulation. In slot intervals (0,40) and (60,100), the DCN's utilization grows as the number of new flows increases. In slot intervals (40,60) and (100,124), the DCN's utilization decreases as the existing flows terminate transmission. We take the power parameters of a switch from [2]. The capacity safety margin ϕ is set at 0.2. We test two AggreFlows: AF(160) with 160 entries and AF(40) with 40 entries. The installment/update/deletion process of a flow entry in existing SDN switches requires the time in the order of milliseconds [8]. Thus, the two AggreFlows do not exceed the processing ability of the switch, and the switch can timely process flow entries.

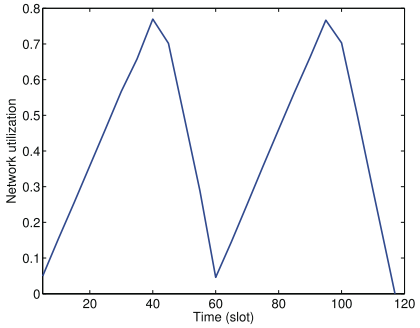


Fig. 9. Data center network's utilization.

C. Simulation Results

In our simulation, we evaluate three aspects for each scheme: load balancing performance, power consumption, and QoS performance.

1) Load balancing Performance: The load balancing performance is evaluated in the form of Root Mean Squared Error (RMSE) of active routes in the DCN [13].

$$RMSE = \sqrt{\frac{\sum_{i=1}^{N}(load_i - load_{ave})^2}{N}} \quad (3)$$

where i denotes the i-th link in the network. The network consists of N links, $load_i$ denotes the i-th link's load, and $load_{ave}$ denotes the average link load of N links in the network.

Fig. 10 shows the load balancing performance of different flow scheduling schemes. The smaller RMSE, the better performance. If all active routes have the same load, RMSE is 0. In the figure, the box represents the center half of the data, and the red line represents the median data. The whiskers include 1-25-50-75-95-th percentiles of the data, and red crosses are 5% outliers.

Balance-oblivious flow-level performs worst since it greedily consolidates flows to the left routes in each switch layer. Balance-aware flow-level presents the best performance because it conducts fine-grained flow-level routing and rerouting based on its global visibility. AggreFlow achieves the mean RMSE comparable to Balance-aware flow-level. AggreFlow's

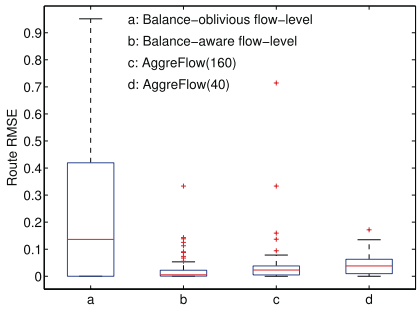


Fig. 10. Load balancing performance of different flow scheduling schemes.

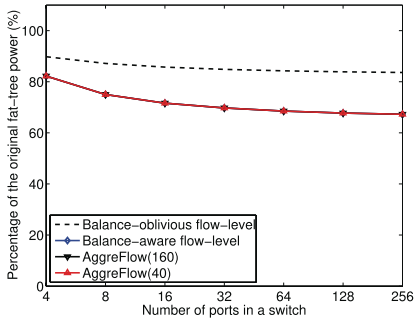


Fig. 11. Power saving. The number of ports in the switch is used to denote the scale of the DCN. Flow-level schemes achieve the optimal result.

load balancing performance can be further improved by using large flow-set routing tables. AggreFlow(160)'s load balancing performance is more close to Balance-aware flow-level than AggreFlow(40). However, the better performance is at cost of higher control messages for rerouting more flow-sets.

2) Power Consumption: In our simulation, every 1 slot, the scheduler collects traffic statistics from switches and decides power status of switches and links. Each scheme selects active switches and links in a deterministic left-to-right order, so that unused switches and links are then turned off in a deterministic right-to-left order to save power. With such an active switch selection order, a specific number of active switches and links is coupled with only one minimum-power subnet.

For each scheme, the power consumption includes the power consumption of active switches and links. In a fat-tree network with k port switches, the number of aggregation switches and core switches are $k^2/2$ and $k^2/4$, respectively. Thus, we use the number of ports in the switch to denote the scale of a DCN. Fig. 11 shows the power consumption of ports and switches of the schemes with the original fat-tree network using different switches. In our simulation, each scheme schedules flows and turns off the links/ports of corresponding switches to reduce power consumption. When all links/ports of a switch are turned off, this switch will be turned off to save power consumption.

For the metric of power consumption, we can divide flow scheduling schemes into two categories: the balance-oblivious scheme (i.e, Balance-oblivious flow-level) and the balance-aware scheme (i.e, all schemes except Balance-oblivious flow-level). In the entire simulation, both flow-level balance-aware schemes and AggreFlow consume the same power. In the figure, as the number of switches' ports increases, the power saving increases and reaches the threshold 67% for the balance-aware schemes and 85% for the balance-oblivious scheme. Good load balancing performance reduces power consumption by about 18% on average, while the unbalanced load allocation degrades power efficiency as the DCN's scale expands. Balance-oblivious flow-level scheduling scheme enables the controller to consolidate every flow to the leftmost route that has sufficient capacity for the flow. However, this scheme does not support load balancing and active routes could be allocated with unbalanced traffic. When bursty traffic surges on congested routes, this scheme will require extra switches and links to accommodate the traffic variation and thus consume more power than Balance-aware schemes. Balance-aware flow-level scheduling presents good performance of power saving. However, it works in a fine-grained per-flow fashion and thus has much higher complexity in terms of flow table storage and management than AggreFlow, which manage flows in the manner of sets, as shown in Fig. 17.

3) QoS Performance: We use four metrics to evaluate the QoS performace for each scheme: packet loss rate, routing overhead, rerouting overhead, and cumulative scheduling overhead.

(1) Packet Loss Rate. Packet loss comes from the procedure of the minimum-power subnet reconfiguration. As traffic load increases, links have to accommodate more flows and become congested. When the load on the most congested link exceeds a pre-determined threshold (1-ϕ), a new minimum-power subnet is generated to relieve the current network congestion.

In our simulation, packet loss mainly comes from slots 15 and 75. As shown in Fig. 9, traffic load increases equally at each slot interval. However, at the two slots, the subnet is small and thus performs worse than a large subnet to prevent packet loss when traffic surges occur. For the same subnet, the load balancing performance impacts packet loss rate. Compared with the imbalanced load allocation, a balanced traffic allocation not only postpones the time that links reach the pre-determined thresholds, but also reduces the number of packets lost on the most congested links. Packet loss rate of Balance-aware flow-level, AggreFlow(40) and AggreFlow(160) are 0.19%, 0.193% and 0.193%, respectively. Although these AggreFlow schemes do not achieve the same load balancing performance as Balance-aware flow-level does, their load balancing performance is good enough to handle traffic surges on congested links.

(2) Flow Routing Overhead. We define the flow routing overhead as the number of control messages used for routing new flows at a single slot. In the following figures, we do not present the results of Balance-oblivious flow-level since it has the same result of Balance-aware flow-level. Fig. 12 shows flow routing overhead of Balance-aware flow-level, AggreFlow(40), and AggreFlow(160). In the figure, Balance-aware flow-level consumes the highest overhead. In the two intervals (0,40) and (60,100), Balance-aware flow-level's overhead maintains about 1,500 control messages per slot since new flows arrive at the network in an approximate similar rate. In slot intervals (40,60) and (100,120), no new flows enter the network, and Balance-aware flow-level does not route any new flow.

AggreFlow(40)'s routing overhead is about 99% less than Balance-aware flow-level and only comes from the intervals (0,5) and (61,68). In intervals (0,5) and (61,68), AggreFlow creates new flow-sets as new flows enter the DCN. Specifically, in slot interval (0,5), the flow-set routing tables on different edge switches are initialized when the first flow of each flow-set enters the network. AggreFlow(40)'s highest overhead

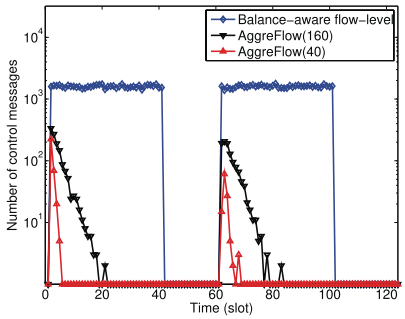


Fig. 12. Flow routing overhead of different flow scheduling schemes. The balancing-aware and balancing-oblivious flow-level schemes have the same performance.

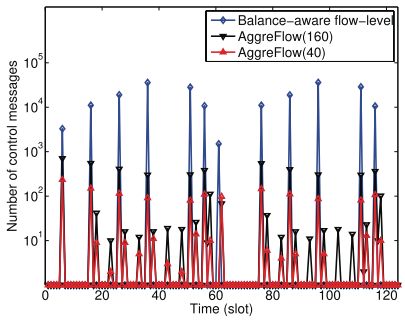


Fig. 13. Flow rerouting overhead of different flow scheduling schemes. The balancing-aware and balancing-oblivious flow-level schemes have the same performance.

comes from slot 1, the first slot to initialize flow-set routing tables, but it is still about 87.4% less than Balance-aware flow-level. As the number of initialized flow-sets increases, AggreFlow(40)'s overhead decreases. At the end of slot 5, the initialization completes. After the initialization, each flow-set is assigned with a route. Thus, in the slot interval (6,40), when an edge switch receives new flows, it finds a new flow's route from its flow-set routing table. In interval (40,60), no new flows enter the network, and some existing flows terminate transmission. As a result, a few flow-set entries are disabled from flow-set routing tables when their timeouts expire. Similarly, in slot interval (61,68), new flows enter the network and the DCN utilization increases. Under such a condition, some flow-set entries are initialized again in flow-set routing tables. AggreFlow(160)'s overhead is a little higher than that of AggreFlow(40) and comes from two longer slot intervals (0,18) and (61,83). AggreFlow(160) uses larger flow-set tables so that it requires more control messages and longer time to initialize its tables than AggreFlow(40) does.

(3) Flow Rerouting Overhead. The number of control messages used for rerouting existing flows at slot t is named flow rerouting overhead at slot t. Fig. 13 shows flow rerouting overhead of Balance-aware flow-level, AggreFlow(40) and AggreFlow(160). In the figure, the overheads of all schemes vary as the DCN utilization changes shown in Fig. 9. Balance-aware flow-level performs worst and consumes 36,067 messages at slot 96. This is because when each subnet changes, it reroutes a large number of flows, and each rerouting operation requires multiple control messages.

AggreFlow(40) reduces the overhead by about 99% compared with Balance-aware flow-level. Since the controller can reroute a set of flows by sending one control message to an edge switch, AggreFlow(40) requires much less control

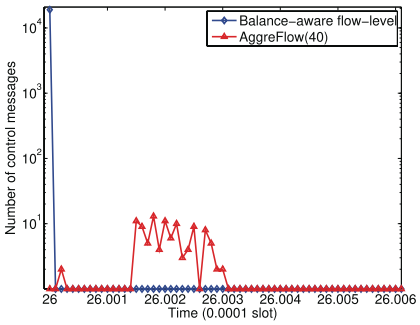


Fig. 14. Flow rerouting overhead in interval (26,26.006).

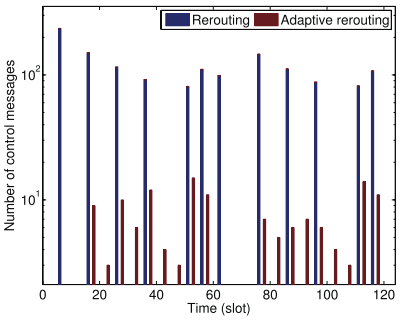


Fig. 15. Composition of AggreFlow(40)'s flow rerouting overhead.

messages to reroute the same number of flows than Balance-aware flow-level does.

AggreFlow conducts rerouting operations in a relative long time. There are two reasons. First, using Lazy Rerouting, an AggreFlow agent reroutes a flow-set when it receives a packet belonging to the flow-set. Fig. 14 shows the overhead consumed by rerouting operations in the interval (26,26.006). At slot 26, the minimum-power subnet changes, and Balance-aware low-level reroute flows immediately, whereas AggreFlow(40) spreads its rerouting operation over two separate slot intervals (26,26.0001) and (26.0011,26.0028). Second, using Adaptive Rerouting, as long as the scheduler detects imbalanced traffic loads on active routes, AggreFlow then will dynamically reroute some flow-sets to maintain load balancing. Compared with Balance-aware flow-level that only does the rerouting operations at a specific time, AggreFlow's rerouting operations are conducted as the network status changes.

Fig. 15 shows the composition of AggreFlow(40)'s flow rerouting communication overhead. In the figure, AggreFlow(40) conducts 18 adaptive rerouting operations, and each rerouting operation consumes 1-13 control messages. In Fig. 16, Adaptive Rerouting only consumes 9.2% of the total flow rerouting communication overhead.

In Fig. 13, AggreFlow(160)'s overhead is larger than that of AggreFlow(40). AggreFlow(160) uses large flow-set routing tables and requires more control messages in each rerouting operation than AggreFlow(40) does. Fig. 16 shows the flow rerouting overhead of AggreFlow(40) and AggreFlow(160) from Adaptive Rerouting. In the figure, AggreFlow(160) consumes more control messages than AggreFlow(40) in each adaptive rerouting operation. However, AggreFlow(160)'s overhead is still much less than Balance-aware flow-level.

(4) Cumulative Scheduling Overhead. Fig. 17 shows cumulative overhead of different flow scheduling schemes. Compared with Balance-aware flow-level, AggreFlow(40) and

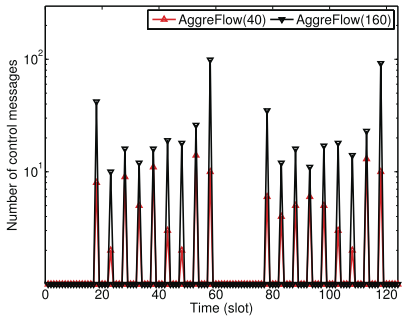


Fig. 16. Flow rerouting overhead from Adaptive Rerouting.

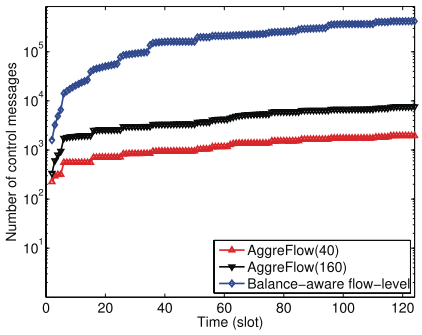


Fig. 17. Cumulative scheduling overhead of different flow scheduling schemes.

AggreFlow(160) reduce cumulative overhead by about 99% and 98% on average, respectively.

VII. Discussion

In this section, we discuss some issues related to AggreFlow.

1) Prevent out-of-order Packets: In practice, we apply some actions to ensure a flow's packets arrive in-order after each rerouting operation. For a to-be-rerouted flow, the ingress edge switch uses a bit to make a sign on the last packet of the flow and sets the packet with a high priority to make sure it is delivered to the egress switch. If some packets of the flow arrives earlier at egress edge switch from newly open routes, they are buffered until the signed packet is delivered. The actions prevent out-of-order packets and maintain good latency.

2) Header Encapsulation: Similarly to VL2 [11], AggreFlow can use an IP-in-IP encapsulation to insert the switch addresses. For a small DCN, we can give each switch a specific number to represent its address and reuse VLAN field to insert switch address [22]. Besides existing encapsulation techniques, we can also use state-of-the-art techniques (e.g., POF [23], PIF [24]) to design flexible packet headers.

3) AggreFlow Application Scenario: In DCNs, some network applications may pay attention to specific information contained in each packet of every flow. For example, in the applicaiton of network firewall, the packet is forwarded or dropped based on the rules that match the packet's source IP address, destination IP address and port numbers. AggreFlow can also be applied to such applications with small modification. For instance, we can conduct access control for each flow on edge switches (e.g., context-aware detection on packets of flows) before the flows are aggregated into flow-sets. We leave this issue for our future work.

VIII. Related Work

A. Data Center Cost Saving

In recent years, many studies have been conducted to optimize data center cost. Some studies propose to dynamic adjust devices to save the power consumption of a data center (e.g., servers [25], [51], cooling system [26], [27]). Some other works propose to reduce the electricity cost of distributed data centers considering practical factors, such as time-of-use electricity rates [28]–[33], renewable energy availability [34]–[36].

Some recent studies consider the power consumption of network devices. ElasticTree [2] turns on and off switches and links based on the current traffic demand and consolidates traffic on a minimum-power subnet. CARPO [3] consolidates low-correlation flows together to further save power based on an observation that bandwidth demands of low-correlation flows usually do not peak at the same time in real DCNs. Widjaja *et al.* [6] explore the impacts of stage and switch size on power saving of a DCN. In [37], the authors further reduce power by considering the correlation between the DCN and servers. PowerFCT [38] considers the requirements of flows and traffic consolidation with a component throttling to reduce Flow Completion Time (FCT) and energy consumption. FCTcon [5] works along with both a FCT control augment factor and a traffic consolidation module to dynamically adjust the FCT of delay-sensitive flows to meet their deadlines while saving energy. The similar idea is also proposed in [39]. SmartFCT [40] employs Deep Reinforcement Learning (DRL) to improve the power efficiency of DCNs and guarantee FCT. Our work considers the impact of load balancing on power efficiency and solves the scalability problem to deploy a power-efficient DCN with time-varying traffic loads.

B. Flow Scheduling in DCNs

In Hedera [10], new flows are recognized as mice flows and routed by edge switches with oblivious static schemes. When a flow's transmission rate grows past a threshold rate, it is detected as an elephant flow and rerouted to a new route with less load. In DevoFlow [41], flows are classified into mice and elephant flows based on their transferred volumes. Mice flow routing are achieved by matching the exact-match flow entries, whereas DevoFlow controller reroutes elephant flows to the least congested path between the flows' end-hosts. Mahout [42] also focuses scheduling elephant flows, which are detected at end-hosts by looking at the TCP buffer of outgoing flows. The above schemes are designed for static DCNs, where all switches and links are always turned on. As the minimum-power subnet changes frequently, they may lead to imbalanced load on active routes or frequent control message storms. DISCO [43] schedules flows in a distributed fashion and considers flow correlation and delay constraints.

Compared with the state-of-the-art load balancing schemes in DCNs (i.e., LetFlow [44], CONGA [45], Hermes [46], and HULA [47], AggreFlow has several different aspects. First, most of the aforementioned schemes (e.g., [44], [45], [47]) work on a granularity of flowlet, which requires dedicated hardware and is inflexible due to the fixed flowlet timeout (e.g., 500μs). By contrast, AggreFlow only relies on SDN to perform load balancing strategies. Second, thanks to the SDN controller, AggreFlow can perform much accurate load balancing strategies because of the global network traffic

information and global decisions compared with distributed schemes like [46].

IX. Conclusion and Future Work

In this paper, we identify two practical issues for deploying power-efficient DCNs: unbalanced traffic allocation of active routes could degrade power efficiency; frequent control message storms would overwhelm OpenFlow switches. To achieve power saving and load balancing with a low overhead, we propose a dynamic flow scheduling scheme named AggreFlow. AggreFlow schedules flows in a coarse-grained flow-set fashion, employs lazy rerouting to amortize a huge number of simultaneous rerouting operations over a relatively long period of time, and adaptively reroutes flow-sets to maintain load balancing on active routes. Simulation results show that, compared with baseline schemes, AggreFlow achieves a good power efficiency and a good load balancing performance with a lower overhead. Sophisticated aggregating flow design can construct better flow-sets at the cost of higher processing load and delay on switches. In future, we will consider the efficient aggregating flow design.

Acknowledgment

The authors would like to thank Shufeng Hui's early contribution for this paper. The work was partly conducted while Zehua Guo worked at the New York University and University of Minnesota.

References

[1] Z. Guo, S. Hui, Y. Xu, and H. J. Chao, "Dynamic flow scheduling for power-efficient data center networks," in *Proc. IEEE/ACM IWQoS*, Jun. 2016, pp. 1–10.

[2] B. Heller *et al.*, "Elastictree: Saving energy in data center networks," in *Proc. USENIX NSDI*, 2010, pp. 249–264.

[3] X. Wang, Y. Yao, X. Wang, K. Lu, and Q. Cao, "Carpo: Correlation-aware power optimization in data center networks," in *Proc. IEEE INFOCOM*, Mar. 2012, pp. 1125–1133.

[4] X. Wang, X. Wang, K. Zheng, Y. Yao, and Q. Cao, "Correlation-aware traffic consolidation for power optimization of data center networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 4, pp. 992–1006, Apr. 2016.

[5] K. Zheng and X. Wang, "Dynamic control of flow completion time for power efficiency of data center networks," in *Proc. ICDCS*, 2017, pp. 340–350.

[6] I. Widjaja, A. Walid, Y. Luo, Y. Xu, and H. J. Chao, "Small versus large: Switch sizing in topology design of energy-efficient data centers," in *Proc. IEEE/ACM IWQoS*, Jun. 2013, pp. 1–6.

[7] N. McKeown *et al.*, "OpenFlow: Enabling innovation in campus networks," *ACM CCR*, vol. 38, no. 2, pp. 69–74, 2008.

[8] K. He *et al.*, "Measuring control plane latency in SDN-enabled switches," in *Proc. ACM SIGCOMM SOSR*, 2015.

[9] A. Wang, Y. Guo, F. Hao, T. Lakshman, and S. Chen, "Scotch: Elastically scaling up SDN control-plane using vswitch based overlay," in *Proc. ACM CoNext*, 2014, pp. 403–414.

[10] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks," in *Proc. USENIX NSDI*, 2010, pp. 89–92.

[11] A. Greenberg and *et al.*, "VL2: A scalable and flexible data center network," *ACM CCR*, vol. 39, no. 4, pp. 51–62, 2009.

[12] T. Benson, A. Anand, A. Akella, and M. Zhang, "Understanding data center traffic characteristics," in *Proc. ACM WREN*, 2009, pp. 65–72.

[13] Z. Guo *et al.*, "Improving the performance of load balancing in software-defined networks through load variance-based synchronization," *Elsevier Comput. Netw.*, vol. 68, pp. 95–109, Aug. 2014.

[14] (2013). *Extremetech*. [Online]. Available: http://www.extremetech.com/extreme/161772-microsoft-now-has-one-million-servers-less-than-google-but-more-than-amazon-says-ballmer

[15] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proc. ACM SIGCOMM IMC*, 2010, pp. 267–280.

[16] B. Yan, Y. Xu, H. Xing, K. Xi, and H. J. Chao, "Cab: A reactive wildcard rule caching system for software-defined networks," in *Proc. ACM HotSDN*, 2014, pp. 163–168.

[17] A. Vishnoi, R. Poddar, V. Mann, and S. Bhattacharya, "Effective switch memory management in openflow networks," in *Proc. ACM DEBS*, 2014, pp. 177–188.

[18] R. L. Graham, "Bounds for certain multiprocessing anomalies," *Bell Syst. Tech. J.*, vol. 45, no. 9, pp. 1563–1581, 1966.

[19] A. Goel and P. Indyk, "Stochastic load balancing and related problems," in *Proc. 40th Annu. Symp. Found. Comput. Sci.*, 1999, pp. 579–586.

[20] D. P. Bertsekas, R. G. Gallager, and P. Humblet, *Data Networks*, vol. 2. Upper Saddle River, NJ, USA: Prentice-Hall, 1992.

[21] *Openflow Switch Specification V1.3.1*, Open Netw. Found., Menlo Park, CA, USA, 2012.

[22] A. S. Iyer, V. Mann, and N. R. Samineni, "Switchreduce: Reducing switch state and controller involvement in openflow networks," in *Proc. IFIP Netw.*, 2013, pp. 1–9.

[23] H. Song, "Protocol-oblivious forwarding: Unleash the power of SDN through a future-proof forwarding plane," in *Proc. ACM HotSDN*, 2013, pp. 127–132.

[24] P. Bosshart and *et al.*, "P4: Programming protocol-independent packet processors," *ACM CCR*, vol. 44, no. 3, pp. 87–95, 2014.

[25] J. Heo, D. Henriksson, X. Liu, and T. Abdelzaher, "Integrating adaptive components: An emerging challenge in performance-adaptive systems and a server farm case-study," in *Proc. IEEE RTSS*, Dec. 2007, pp. 227–238.

[26] L. A. Barroso and U. Hölzle, "The case for energy-proportional computing," *Computer*, vol. 40, no. 12, pp. 33–37, Dec. 2007.

[27] C. Bash and G. Forman, "Cool job allocation: Measuring the power savings of placing jobs at cooling-efficient locations in the data center," in *Proc. USENIX ATC*, 2007, p. 140.

[28] A. Qureshi, R. Weber, H. Balakrishnan, J. Guttag, and B. Maggs, "Cutting the electric bill for Internet-scale systems," *ACM CCR*, vol. 39, no. 4, pp. 123–134, 2009.

[29] L. Rao, X. Liu, L. Xie, and W. Liu, "Minimizing electricity cost: Optimization of distributed Internet data centers in a multi-electricity-market environment," in *Proc. INFOCOM*, Mar. 2010, pp. 1–9.

[30] Z. Guo, Z. Duan, Y. Xu, and H. J. Chao, "Cutting the electricity cost of distributed datacenters through smart workload dispatching," *IEEE Commun. Lett.*, vol. 17, no. 12, pp. 2384–2387, Dec. 2013.

[31] Y. Zhang, Y. Wang, and X. Wang, "Electricity bill capping for cloud-scale data centers that impact the power markets," in *Proc. IEEE ICPP*, Sep. 2012, pp. 440–449.

[32] J. Li, Z. Li, K. Ren, and X. Liu, "Towards optimal electric demand management for Internet data centers," *IEEE Trans. Smart Grid*, vol. 3, no. 1, pp. 183–192, Oct. 2012.

[33] Z. Guo, Z. Duan, Y. Xu, and H. J. Chao, "JET: Electricity cost-aware dynamic workload management in geographically distributed datacenters," *Comput. Commun.*, vol. 50, pp. 162–174, Sep. 2014.

[34] Y. Zhang, Y. Wang, and X. Wang, "Greenware: Greening cloud-scale data centers to maximize the use of renewable energy," in *Proc. ACM/IFIP/USENIX Middleware*, 2011, pp. 143–164.

[35] Y. Zhang, Y. Wang, and X. Wang, "TEStore: Exploiting thermal and energy storage to cut the electricity bill for datacenter cooling," in *Proc. IFIP CNSM*, 2012, pp. 19–27.

[36] Z. Liu, M. Lin, A. Wierman, S. H. Low, and L. L. Andrew, "Greening geographical load balancing," in *Proc. ACM SIGMETRICS*, 2011, pp. 233–244.

[37] K. Zheng, X. Wang, L. Li, and X. Wang, "Joint power optimization of data center network and servers with correlation analysis," in *Proc. IEEE INFOCOM*, Apr. 2014, pp. 2598–2606.

[38] K. Zheng, X. Wang, and X. Wang, "PowerFCT: Power optimization of data center network with flow completion time constraints," in *Proc. IEEE IPDPS*, May 2015, pp. 334–343.

[39] G. Xu, B. Dai, B. Huang, J. Yang, and S. Wen, "Bandwidth-aware energy efficient flow scheduling with SDN in data center networks," *Elsevier FGCS*, vol. 68, pp. 163–174, Mar. 2017.

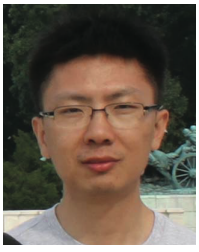
[40] P. Sun, Z. Guo, S. Liu, J. Wang, J. Lan, and Y. Hu, "SmartFCT: Improving power-efficiency for data center networks with deep reinforcement learning," *Comput. Netw.*, vol. 179, Oct. 2020, Art. no. 107255.

[41] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "DevoFlow: Scaling flow management for high-performance networks," *Comput. Commun. Rev.*, vol. 41, no. 4, pp. 254–265, Aug. 2011.

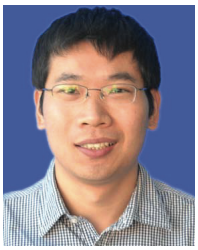
[42] A. R. Curtis, W. Kim, and P. Yalagandula, "Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection," in *Proc. IEEE INFOCOM*, Apr. 2011, pp. 1629–1637.

[43] K. Zheng, X. Wang, and J. Liu, "Disco: Distributed traffic flow consolidation for power efficient data center network," in *Proc. IFIP Netw.*, 2017, pp. 1–9.
[44] E. Vanini, R. Pan, M. Alizadeh, P. Taheri, and T. Edsall, "Let it flow: Resilient asymmetric load balancing with flowlet switching," in *Proc. NSDI*, 2017, pp. 407–420.
[45] M. Alizadeh *et al.*, "CONGA: Distributed congestion-aware load balancing for datacenters," *ACM SIGCOMM CCR*, vol. 44, no. 4, pp. 503–514, 2014.
[46] H. Zhang, J. Zhang, W. Bai, K. Chen, and M. Chowdhury, "Resilient datacenter load balancing in the wild," in *Proc. ACM Sigcomm*, 2017, pp. 253–266.
[47] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford, "HULA: Scalable load balancing using programmable data planes," in *Proc. Symp. SDN Res.*, 2016, p. 10.
[48] Y. Xu *et al.*, "Dynamic switch migration in distributed software-defined networks to achieve controller load balance," *IEEE J. Sel. Areas Commun.*, vol. 37, no. 3, pp. 515–529, 2019.
[49] Y. Xu *et al.*, "RAPID: Avoiding TCP incast throughput collapse in public clouds with intelligent packet discarding," *IEEE J. Sel. Areas Commun.*, vol. 37, no. 8, pp. 1911–1923, 2019.
[50] D. Li, J. Zhu, J. Wu, J. Guan, and Y. Zhang, "Guaranteeing heterogeneous bandwidth demand in multitenant data center networks," *IEEE/ACM Trans. Netw.*, vol. 23, no. 5, pp. 1648–1660, 2014.
[51] D. Li, Y. Shang, W. He, and C. Chen, "EXR: Greening data center network with software defined exclusive routing," *IEEE Trans. Comput.*, vol. 64, no. 9, pp. 2534–2544, 2014.

Zehua Guo (Senior Member, IEEE) received the B.S. degree from Northwestern Polytechnical University, the M.S. degree from Xidian University, and the Ph.D. degree from Northwestern Polytechnical University. He was a Research Fellow at the Department of Electrical and Computer Engineering, New York University Tandon School of Engineering, a Post-Doctoral Research Associate with the Department of Computer Science and Engineering, University of Minnesota, Twin Cities, MN, USA, and a Visiting Associate Professor with the Singapore University of Technology and Design. He is currently an Associate Professor with the Beijing Institute of Technology. His research interests include software-defined networking, network function virtualization, machine learning, data center network, cloud computing, content delivery network, network security, and Internet exchange. He is a member of the ACM. He was the Session Chair for the IEEE ICC 2018 and is the Technical Program Committee Member of Computer Communications and prestigious conferences, such as AAAI, ICC, ICCCN, and ICA3PP. He is an Associate Editor of the IEEE SYSTEMS JOURNAL, IEEE ACCESS, and *EURASIP Journal on Wireless Communications and Networking* (Springer), an Editor of the *KSII Transactions on Internet and Information Systems*, and a Guest Editor of the *Journal of Parallel and Distributed Computing*.



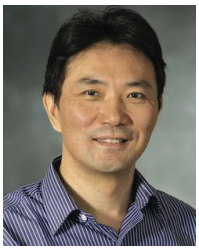
Yang Xu (Member, IEEE) received the B.Eng. degree from the Beijing University of Posts and Telecommunications, in 2001, and the Ph.D. degree in computer science and technology from Tsinghua University, China, in 2007. He is currently the Yaoshihua Chair Professor with the School of Computer Science, Fudan University. Prior to joining Fudan University, he was a Faculty Member with the Department of Electrical and Computer Engineering, New York University Tandon School of Engineering. His research interests include software-defined networks, data center networks, distributed machine learning, edge computing, network function virtualization, and network security. He has published more than 80 journal and conference papers and holds more than ten U.S. and international granted patents on various aspects of networking and computing. He has served as a TPC member for many international conferences. He has also served as an Editor for the *Journal of Network and Computer Applications* (Elsevier), and as a Guest Editor for the IEEE JOURNAL ON SELECTED AREAS IN COMMUNICATIONS–SPECIAL SERIES ON NETWORK SOFTWARIZATION & ENABLERS and *Wiley Security and Communication Networks Journal–Special Issue on Network Security and Management in SDN*.



Ya-Feng Liu (Senior Member, IEEE) received the B.Sc. degree in applied mathematics from Xidian University, Xi'an, China, in 2007, and the Ph.D. degree in computational mathematics from the Chinese Academy of Sciences (CAS), Beijing, China, in 2012. During his Ph.D. study, he was supported by the Academy of Mathematics and Systems Science (AMSS), CAS, to visit Professor Zhi-Quan (Tom) Luo at the University of Minnesota, Twin Cities, MN, USA, from 2011 to 2012. After his graduation, he joined the Institute of Computational Mathematics and Scientific/Engineering Computing, AMSS, CAS, in 2012, where he became an Associate Professor in 2018. His main research interests are nonlinear optimization and its applications to signal processing, wireless communications, and machine learning. He is an elected member of the Signal Processing for Communications and Networking Technical Committee (SPCOM-TC) of the IEEE Signal Processing Society. He received the Best Paper Award from the IEEE International Conference on Communications (ICC) in 2011, the Best Student Paper Award from the International Symposium on Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks (WiOpt) in 2015, the Chen Jingrun Star Award from the AMSS in 2012, and the Science and Technology Award for Young Scholars from the Operations Research Society of China in 2018. He currently serves as an Editor for the IEEE TRANSACTIONS ON WIRELESS COMMUNICATIONS, and as an Associate Editor for the IEEE SIGNAL PROCESSING LETTERS and the *Journal of Global Optimization*.

Sen Liu (Member, IEEE) received the B.S. degree from Northeastern University, the M.S. degrees from the South China University of Technology, and the Ph.D. degree from Central South University. He also worked as a Visiting Scholar with the Department of Computer Science and Engineering, University of Minnesota, Twin Cities, MN, USA, from 2018 to 2019. He is currently a Post-Doctoral Research Associate with Fudan University. His research interests include congestion control, network traffic load balancing in data center networks, and performance optimization in software-defined networks.

H. Jonathan Chao (Life Fellow, IEEE) received the B.S. and M.S. degrees in electrical engineering from National Chiao Tung University, Taiwan, in 1977 and 1980, respectively, and the Ph.D. degree in electrical engineering from The Ohio State University, Columbus, OH, USA, in 1985. He was the Head of the Electrical and Computer Engineering (ECE) Department, New York University (NYU), from 2004 to 2014. From 2000 to 2001, he was the Co-Founder and a CTO of Coree Networks, Tinton Falls, NJ, USA. From 1985 to 1992, he was a Member of Technical Staff at Bellcore, Piscataway, NJ, where he was involved in transport and switching system architecture designs and application-specified integrated circuit implementations, such as the world's first SONET-like framer chip, ATM layer chip, sequencer chip (the first chip handling packet scheduling), and ATM switch chip. He is currently a Professor of ECE at NYU, New York City, NY, USA. He is also the Director of the High-Speed Networking Lab. He has coauthored three networking books: *Broadband Packet Switching Technologies—A Practical Guide to ATM Switches and IP Routers* (New York: Wiley, 2001); *Quality of Service Control in High-Speed Networks* (New York: Wiley, 2001); and *High-Performance Switches and Routers* (New York: Wiley, 2007). He holds 63 patents and has published more than 260 journals and conference papers. He has been doing research in the areas of software-defined networking, network function virtualization, datacenter networks, high-speed packet processing/switching/routing, network security, quality-of-service control, network on chip, and machine learning for networking. He is a Fellow of the National Academy of Inventors. He was a recipient of the Bellcore Excellence Award in 1987. He was a co-recipient of the 2001 Best Paper Award from the IEEE TRANSACTION ON CIRCUITS AND SYSTEMS FOR VIDEO TECHNOLOGY.



Zhi-Li Zhang (Fellow, IEEE) received the B.S. degree in computer science from Nanjing University, China, in 1986, and the M.S. and Ph.D. degrees in computer science from the University of Massachusetts, in 1992 and 1997, respectively. In 1997, he joined the Computer Science and Engineering Faculty, University of Minnesota, where he is currently a Qwest Chair Professor and Distinguished McKnight University Professor. His research interests lie broadly in computer communication and networks, Internet technology, content distribution systems, cloud computing, and emerging IoT applications. He has co-chaired several conferences/workshops (as general or TCP chairs), including the IEEE INFOCOM, IEEE ICNP, ACM CONEXT, IFIP Networking, and has served on the TPC for numerous conferences/workshops. He is a member of the ACM. He is a co-recipient of several Best Paper Awards and has received several other awards.

Yuanqing Xia (Senior Member, IEEE) was appointed as the Xu Teli Distinguished Professor at the Beijing Institute of Technology in 2012, and was made the Chair Professor in 2016. In 2012, he obtained the National Science Foundation for Distinguished Young Scholars of China. In 2016, he was honored as the Yangtze River Scholar Distinguished Professor and was supported by the National High Level Talents Special Support Plan ("Million People Plan") by the Organization Department of the CPC Central Committee. He is currently the Dean and Professor of the School of Automation, Beijing Institute of Technology. He has published ten monographs in Springer, John Wiley, and CRC, and more than 400 articles in international scientific journals. His research interests include cloud control systems, networked control systems, robust control and signal processing, active disturbance rejection control, unmanned system control, and flight control. He obtained the Second Award of the Beijing Municipal Science and Technology (No. 1), in 2010 and 2015, respectively; the Second National Award for Science and Technology (No. 2) in 2011; and the Second Natural Science Award of the Ministry of Education (No. 1), in 2012 and 2017, respectively. He is the Deputy Editor of the *Journal of Beijing Institute of Technology* and an Associate Editor of *Acta Automatica Sinica*; *Control Theory and Applications*; *International Journal of Innovative Computing, Information, and Control*; and the *International Journal of Automation and Computing*.