Maintaining Control Resiliency and Flow Programmability in Software-Defined WANs During Controller Failures

Zehua Guo, *Senior Member, IEEE, Member, ACM*, Songshi Dou, *Graduate Student Member, IEEE, ACM*, Sen Liu, *Member, IEEE*, Wendi Feng, Wenchao Jiang, Yang Xu, *Member, IEEE*, and Zhi-Li Zhang, *Fellow, IEEE, Member, ACM*

***Abstract*— Providing resilient network control is a critical concern for deploying Software-Defined Networking (SDN) into Wide-Area Networks (WANs). For performance reasons, a Software-Defined WAN is divided into multiple domains controlled by multiple controllers with a logically centralized view. Under controller failures, we need to remap the control of offline switches from failed controllers to other active controllers. Existing solutions have three limitations: (1) the least flow programmability (e.g., the ability to change paths of flows) cannot be maintained; (2) active controllers could be overloaded, interrupting their normal operations; (3) network performance could be degraded because of the increasing controller-switch communication overhead. In this paper, we propose RetroFlow+ to recover the flow programmability and achieve low communication overhead during controller failures. By intelligently configuring a set of selected offline switches working under the legacy routing mode and several active controllers releasing a few control resources, RetroFlow+ enables active controllers to use the minimum control resource to sustain the flow programmability. RetroFlow+ also smartly transfers the control of offline switches with the SDN routing mode to active controllers to minimize the communication overhead from these offline switches to the active controllers. Simulation results show that RetroFlow+ realizes low communication overhead, recovers all offline flows under one and two controller failures, and improves the flow recovery percentage up to 70% under three controller failures, compared with the state-of-the-art solution.**

***Index Terms*— Software-defined networking, control plane, resiliency, programmability, wide area networks, hybrid routing.**

I. Introduction

SOFTWARE-DEFINED Networking (SDN) has been deployed in real networks [2], [3]. One critical scenario for the SDN is Wide-Area Networks (WANs), known as the SD-WANs. For instance, as one of the world's largest Internet service providers, AT&T has softwarized 65% of its WAN with programmable devices (e.g., SDN switches) by 2018 and plans to improve the ratio to 75% by 2020 [4]. In the future, most of the network infrastructure in WANs (e.g., switches and routers) will be replaced by programmable devices.

In SD-WANs, the network is usually divided into multiple domains to achieve low latency control given the large scale of a WAN and the huge number of SDN switches in it [5]. Each domain usually has an SDN controller that can quickly reply to requests from SDN switches within the domain. The controllers from different domains are physically distributed but can achieve a logically centralized control by the synchronization among them, which maintains a consistent network view [6].

Control resiliency is a critical concern for SD-WANs. Essentially, an SDN controller is a network software installed on a physical server or a virtual machine. Due to some unexpected issues (e.g., hardware/software bugs, power failure), one controller could accidentally fail, and then all of its connected switches are out of control, which we refer to as the *offline switches*. Existing solutions to maintain the control resiliency of SDN can be categorized into two classes: (1) controller placement and (2) switch remapping. Solutions in the former category carefully choose physical locations

Manuscript received March 23, 2020; revised July 7, 2021 and August 28, 2021; accepted October 17, 2021; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor Y. Chen. Date of publication January 5, 2022; date of current version June 16, 2022. The work of Zehua Guo was supported in part by the National Natural Science Foundation of China under Grant 62002019 and in part by the Beijing Institute of Technology Research Fund Program for Young Scholars. The work of Sen Liu was supported in part by the National Natural Science Foundation of China under Grant 62002066 and in part by the China Postdoctoral Science Foundation under Grant 2021M690705. The work of Wenchao Jiang was supported in part by the Ministry of Education (MOE) Start-Up Research Grant SRG ISTD 2020 159. The work of Yang Xu was supported in part by the Key-Area Research and Development Program of Guangdong Province under Grant 2021B0101400001, in part by the National Natural Science Foundation of China under Grant 62172108, in part by the Shanghai Pujiang Program under Grant 2020PJD005, and in part by the Science and Technology Commission of Shanghai Municipality under Grant 20S31903800. The work of Zhi-Li Zhang was supported in part by U.S. NSF under Grant CNS-1618339, Grant CNS-1814322, Grant CNS-1836772, Grant CNS-2106771, and Grant CCF-2123987. This paper was presented in part at IEEE/ACM IWQoS 2019 [1] [DOI: 10.1145/3326285.3329036]. *(Corresponding author: Sen Liu.)*

Zehua Guo and Songshi Dou are with the Beijing Institute of Technology, Beijing 100081, China.

Sen Liu is with the School of Computer Science, Fudan University, Shanghai 200433, China (e-mail: senliu@fudan.edu.cn).

Wendi Feng is with the State Key Laboratory of Network and Switching Technology, Beijing University of Posts and Telecommunications, Beijing 100876, China.

Wenchao Jiang is with the Pillar of Information Systems Technology and Design, Singapore University of Technology and Design, Singapore 487372.

Yang Xu is with the School of Computer Science, Fudan University, Shanghai 200433, China, and also with the Peng Cheng Laboratory, Shenzhen 518000, China.

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.2021.3128771

of controllers to optimize the control performance under controller failures, such as minimizing the latency between backup controllers and switches [7], [8] and/or minimizing the latency among the main controller, backup controllers, and switches [9]. These solutions are usually based on some unrealistic assumptions, such as the same control cost and the unlimited capability of controllers, which are far from practical. In contrast, the switch remapping approaches propose to dynamically shift the control of offline switches to other active controllers [10]. However, in an almost saturated SDN, controllers almost reach their processing limits. There will be little room left in active controllers to accept offline switches from the failed controllers without overloading them, which otherwise can degrade the performance (e.g., increasing the communication overhead) or even cause cascading controller failure [10]–[12].

In this paper, we present a feature available in existing commercial SDN switches that shed light on this issue. Existing commercial SDN switches (e.g., Brocade MLX-8 PE [13]) can freely change between two routing modes, the SDN mode and the legacy mode. The former relies on the SDN controller's decision to process flows while the latter processes flows using its traditional routing table without consulting the controller. Inspired by the feature in commercial devices, we propose to configure switches in hybrid modes so that we can enjoy the flow programmability (e.g., the ability to change the paths of flows) brought by the SDN mode while avoiding the out-of-control disasters coming with the offline switches during the controller failures.

To get the best of both worlds, we have overcome three main challenges. First, configuring SDN switches to work in the legacy mode will decrease the flow programmability of SDN. We carefully choose a set of offline switches working in the legacy mode to maximize the number of programmable flows[1] that have at least one alternative path to forward. Second, the limited control resource of active controllers becomes the key bottleneck factor for recovering offline flows. To increase the control resource for recovering flows, we choose some active controllers to release a few control resources for controlling normal flows by reducing the programmability of these normal flows. Third, the switch-controller mapping affects the communication overhead from offline switches to active controllers. For the remaining offline switches that still work in the SDN mode, we carefully remap them to active controllers, which is a complex optimization problem restricted to the switches' control cost (e.g., the per-flow state pulling [14], [15]) and the controller's real-time workload. Since these three problems are coherent, we approach the optimal results with joint optimization.

In summary, our paper makes the following contributions:

- We formulate the joint optimization problem as the *Optimal Configuration and Mapping for Switch and Controller (OCMSC)* problem, which aims to improve the number of recovered flows and keep low communication overhead of controllers by rationally configuring the control resource of active controllers and establishing the mapping between offline switches and active controllers based on the switch and controller states in real time.
- We provide a rigorous proof of the OCMSC problem for its NP-hardness and propose a heuristic solution named *RetroFlow+* to efficiently solve the problem.
- We evaluate the performance of RetroFlow+ under a real topology. Simulation results show that RetroFlow+ realizes low communication overhead, recovers all offline flows under one and two controller failures, and improves the flow recovery percentage up to 70% under three controller failure, compared with the state-of-the-art solution RetroFlow [1].

The rest of the paper is organized as follows. In Section II, we introduce the background of SDN and the motivation of this paper. Section III introduces our design considerations, and Section IV mathematically formulates our design as the OCMSC problem. Section V proves the OCMSC problem's complexity and proposes RetroFlow+ to efficiently solve the problem. We evaluate and analyze the performance of RetroFlow+ in Section VI. Section VII introduces related works, and Section VIII concludes this paper.

II. Background and Motivation

In this section, we introduce the background of SDN, analyze the limitation of SDN under controller failures, and present opportunities to solve the problem using features available in commercial SDN switches.

A. Blessing of the SDN

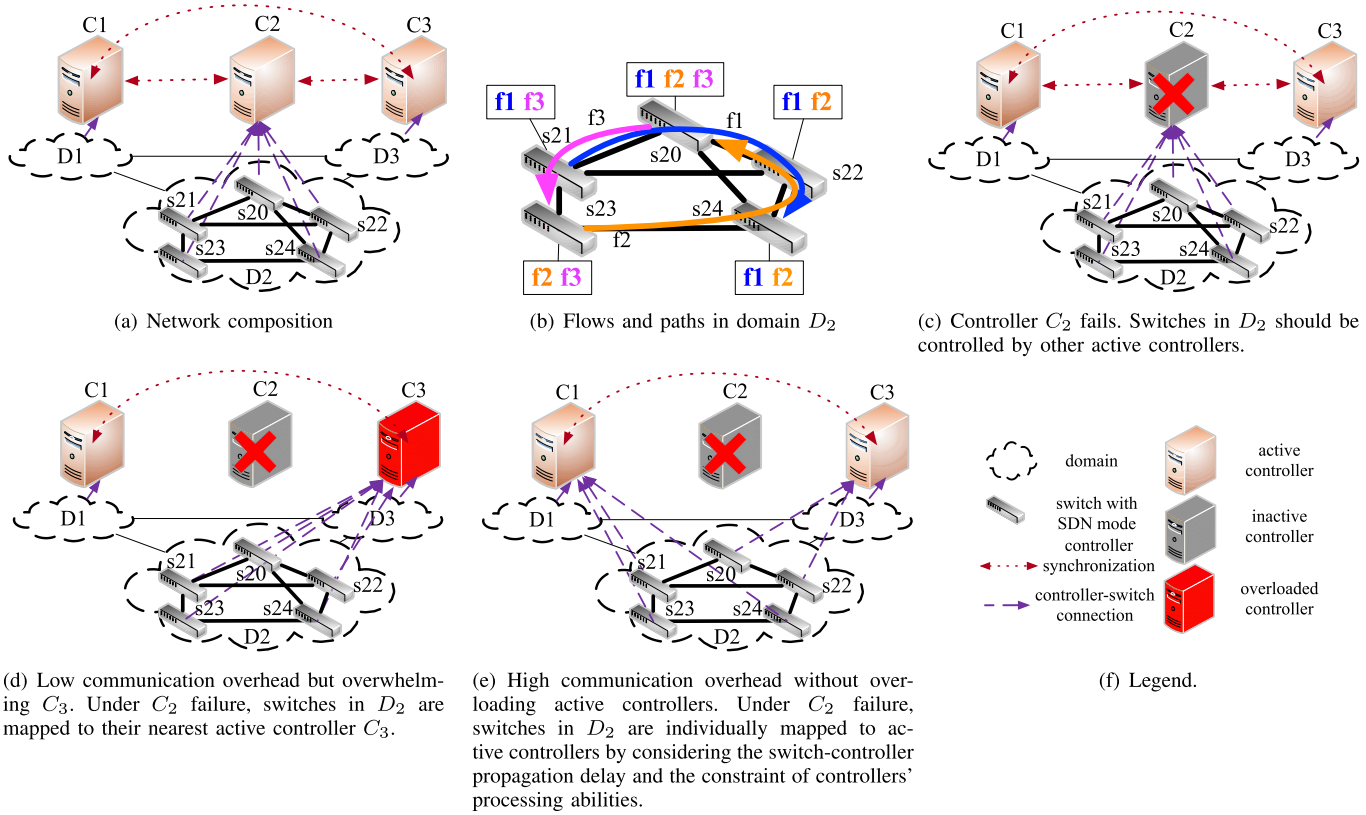
One big benefit of SDN is to provide flexible control on traffic flows based on the global state of the network. To achieve this benefit, the SDN controller can establish forwarding paths for individual flows reactively when they enter the network for the first time or proactively before they arrive at the network. During the network operation, the controller periodically pulls flow state information from the controlled switches to update its global network view and dynamically changes some flows' paths to improve network performance. These unique features and advantages called *flow programmability*, help SDN to prevail over traditional network techniques. Therefore, many networks start to deploy SDN [2]–[4].

For an SD-WAN that consists of many switches, we usually divide the WAN into multiple domains with a different number of SDN switches and use a logically centralized control on domains with distributed controllers [6]. In each domain, its controller can quickly reply to requests from a switches and synchronize with other controllers to maintain a consistent network view.

B. Curse of the SDN

The normal operations of an SD-WAN rely on the controllers' decision and the communication between controllers and switches for conducting the decision and pulling flow state information. The controller becomes the Achilles Heel

[1]In this paper, we use recovered flow and programmable flow interchangeably.



(a) Network composition

(b) Flows and paths in domain D_2

(c) Controller C_2 fails. Switches in D_2 should be controlled by other active controllers.

(d) Low communication overhead but overwhelming C_3. Under C_2 failure, switches in D_2 are mapped to their nearest active controller C_3.

(e) High communication overhead without overloading active controllers. Under C_2 failure, switches in D_2 are individually mapped to active controllers by considering the switch-controller propagation delay and the constraint of controllers' processing abilities.

(f) Legend.

Fig. 1. A motivation example of switch remapping under the controller failure. Switches in D_2 are closer to C_3 than C_1.

of SDN. In other words, an SDN switch will be out of control if its controller fails. In order to provide a resilient control of the network, an SDN switch usually connects to a master controller and several backup controllers [16]. When the master controller of a switch crashes, its connection to the switch becomes inactive, and the switch will request one of its backup controllers to become its new master controller. We call the switches previously controlled by failed controllers the *offline switches*. The problem of remapping the control of offline switches to other active controllers is called *SDN switch remapping under controller failures*. This switch remapping has two impacts on other active controllers:

- **Overloading controllers**: A master controller mainly has two types of operations on switches: (1) flow entry operations to establish/update flows' forwarding paths and (2) flow state pulling operations to get the network state variation. Both of the two operations consume the control resource of the master controller. Backup controllers only maintain the connection to their switches without operations until they become master controllers. Becoming the new master controller of some switches from remote domains increases the processing load of a controller, potentially overloading the controller [10], [11]. Existing studies show the switches' requests handled by an overloaded controller could experience long-tail latency [17], which could degrade the network performance significantly.
- **Increasing the communication overhead of controllers**: The communication overhead of a controller is proportional to two factors: (1) the propagation latency between the controller and its controlled switches and (2) the number of flows in the switches. In WANs, the propagation latency is the dominant factor among all latencies because the propagation latency bounds a controller's control reactions that can be executed at a reasonable speed [18], [19]. A long propagation latency could limit convergence time (e.g., routing convergence, network update). Thus, remapping offline switches to active controllers should consider the propagation latency among the switches and controllers. Otherwise, the reaction of the controllers to dynamic network changes could be delayed.

To better illustrate our view, we use a motivation example in Fig. 1 to show that existing solutions under controller failures suffer from the two impacts. In Fig. 1(a), an SDN consists of three domains, and each domain is controlled by one master controller and connected to two backup controllers. In this example, we mainly focus on domain D_2 and do not show details of the other two domains. In domain D_2, controller C_2 is the master controller that controls five SDN switches $s20$-$s24$. Controllers C_1 and C_3 are backup controllers of D_2. The three controllers synchronize with each other to maintain the consistent network information. Both C_1 and C_3 know that D_2 has five switches and the flow information of D_2. We denote the control resource of a controller as the number of flows. In this example, without interrupting a controller's normal operations, C_1 can pull the state of ten flows, but C_3 can only pull the state of at most five flows. Fig. 1(b)

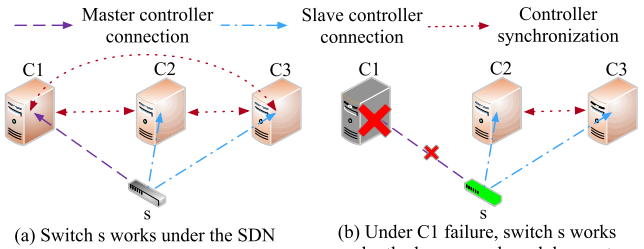


Fig. 2. Switch routing mode configuration.

shows the flows in D_2 and their paths. The flows in a switch are shown in the rectangle connecting the switch.

An SDN is vulnerable. In Fig. 1(c), controller C_2 fails, and the control of the five switches in D_2 should be transferred to active controllers C_1 and C_3. As summarized above, two problems are raised in the following cases:

1) Controller overload: Fig. 1(d) shows that remapping switches in D_2 to active controllers only considers the switch-controller propagation delay. In this figure, the five switches are remapped to their nearest, active controller C_3. This solution minimizes the communication overhead of the entire network but controller C_3 has to pull the states of eleven flows (i.e., $f1$, $f2$, and $f3$ from $s20$, $f1$ and $f3$ from $s21$, $f1$ and $f2$ from $s22$, $f2$ and $f3$ from $s23$, $f1$ and $f2$ from $s24$), which interrupts its normal operations by introducing queueing delays [17].
2) High controller communication overhead: Fig. 1(e) shows that remapping switches in D_2 to active controllers considers the switch-controller propagation delay and controllers' processing abilities. In this figure, switches $s20$ and $s22$ are remapped to controller C_3 while switches $s21$, $s23$, and $s24$ are remapped to controller C_1. This solution prevents controllers from being overloaded but incurs a higher communication overhead than the solution in Fig. 1(d) due to the long propagation delay among offline switches and controllers.

The above two examples show that existing solutions either suffer from the controller overload or high communication overhead, and they cannot solve the problem of SDN switch remapping under controller failures well.

C. Opportunities

We note that many commercial SDN switches today are hybrid SDN switches (e.g., Brocade MLX-8 PE [13]) and support two routing modes: legacy mode and SDN mode. In the legacy mode, switches route flows with the destination-based entries in the routing table generated from legacy routing protocols (e.g., OSPF), whereas, in the SDN mode, they route flows using OpenFlow table managed by the controller. In other words, a switch with the legacy mode can work without the controller. Thus, we can reduce the controller's processing load by configuring some switches working under the legacy mode.

A controller can dynamically set the routing mode of a hybrid SDN switch by sending a control message. Fig. 2 shows an example of changing the routing mode under a controller failure. In Fig. 2(a), switch s connects to its master controller C_1 and backup controllers C_2 and C_3. C_1 sets s to work under the SDN mode, and the three controllers synchronize with others. In Fig. 2(b), when C_1 fails, s identifies its connection to C_1 become inactive and then sends a master controller selection request to C_1, C_2, and C_3. Both C_2 and C_3 reply a rejection message to switch s's request with a legacy mode configuration, and then s starts to work under the legacy mode. s can go back to the SDN mode when one controller wants to become its new master controller. This dynamic routing mode configuration feature offers us more flexibility to tackle the problem of switch remapping under controller failures.

III. Design Considerations

In this section, we propose to relieve the controller's control load for offline switches and reduce the communication overhead during controller failures by introducing the optimal configuration and mapping for switch and controller problem.

A. Switch Mode Configuration Problem

Inspired by the hybrid SDN switches available in the market, we propose to relieve the controller's control load of offline switches by *degrading* a pure SDN of the offline switches to a hybrid SDN that consists of switches with the SDN mode and legacy mode. In other words, we configure a subset of offline switches to run under the SDN mode and others to run under the legacy mode without the management of controllers. However, when configuring the routing mode on switches, we should guarantee the programmability of flows in the hybrid SDN.

The programmability of flows is an essential feature of SDN [20], [21] [22]. Taking the routing problem as an example, the programmability of a flow is the ability to change the flow's path. Existing pure SDN designs realize the network programmability with *a per-hop programmable routing*, which enables the controller to change each flow's path from any switches[2] on the flow's path. In contrast, by maintaining the basic programmability, we keep only *1-hop programmable routing* for flows on offline switches. In other words, we can still change the path of a flow from offline switches by controlling one switch on the flow's path while letting other switches on the flow's path just forward the flow using the legacy destination-based routing. Thus, the switches using the legacy routing do not need to consult the controllers for routing flows.

Fig. 3 shows an example that under the same failure case of Fig. 1(c), we achieve the programmability of flows $f1$-$f3$ using switches with SDN and legacy modes. In this figure, switches $s21$ and $s23$ work under the SDN mode, and switches $s20$, $s22$, and $s24$ work under the legacy mode. Flows $f1$, $f2$, and $f3$'s forwarding paths can be changed at switches $s21$, $s23$, and $s21$, respectively. Note flow $f3$ cannot change its path at switch $s23$ since $s23$ is the destination of $f3$.

[2]If a switch cannot reach a flow's destination through two paths, we will not change the flow's path from this switch.

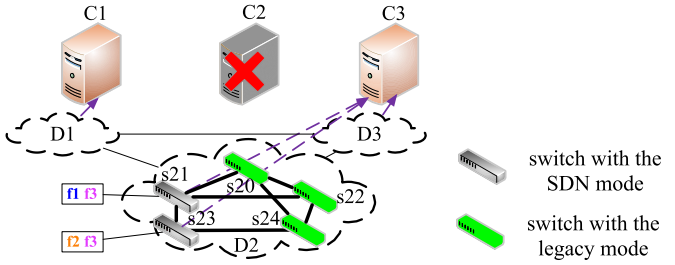


Fig. 3. Switch remapping under single controller failure using SDN and legacy modes. Realizing the programmability of offline flows $f1$-$f3$. Under C_2 failure, $s20$, $s22$, and $s24$ are configured with the legacy mode, and $s21$ and $s23$ with the SDN mode are mapped to controller C_3.

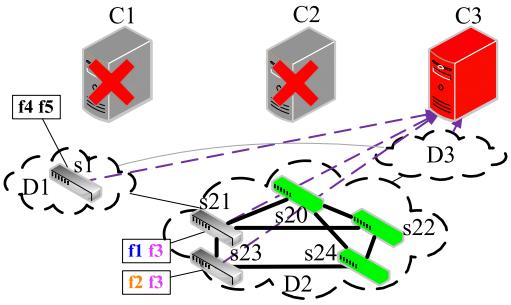
B. Controller Control Resource Configuration Problem

With the reduced number of switches under the SDN mode, we have a higher probability to recover offline flows. However, under the serious controller failure, the rest control resource of active controllers could not be enough to recover each offline flow to become programmable even if degenerating some switches working under the legacy mode. The load of a controller can be measured as the number of flows the controller can control, and it is restricted to the control resource upper bound. As explained in Section II-B, the load consists of two parts: flow entry operations and flow state pulling operations. The former one has been considered in many existing works [23], while we argue the latter one is more important in the WAN scenario. In WANs, each flow's path is usually proactively configured since each flow is an aggregated large flow of multiple flows and always has a traffic rate. Only a limited number of flow entry operations are conducted to reroute some flows occasionally (e.g., congestion). However, each controller conducts the flow state pulling operations periodically (e.g., every few seconds [24]) to maintain the updated network view. Per-flow pulling [14], [15] are popular state pulling methods. The controller sends a request to a switch to pull the state of one flow. The number of flow state pulling requests is usually proportional to the number of flows in a switch. Thus, in WANs, the flow state pulling is a big overhead for controllers and is our main concern in this paper.

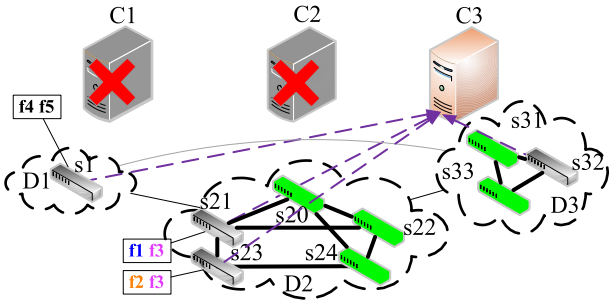
Fig. 4(a) shows an example of overloading the active controller to recover offline flows during two controller failures. In this figure, both controllers C_1 and C_2 fail, and the two domains have offline flows $f1$-$f5$. To recover the five flows, switches $s1$, $s21$, and $s23$ are configured with the SDN mode and mapped to controller C_3, which is overloaded for handling these flows. Under the condition, one potential solution is to increase the rest control resource of C_3 by releasing its control resource used for enabling the programmability of normal flows in $D3$. Fig. 4(b) shows an example using this idea. In this figure, under controllers C_1 and C_2 failure, C_3 releases its control resource by configuring $s31$ and $s33$ with the legacy mode. Thus, it has enough control resources to recover offline flows $f1$-$f5$ from switches $s1$, $s21$, and $s23$.

C. Switch-Controller Remapping Problem

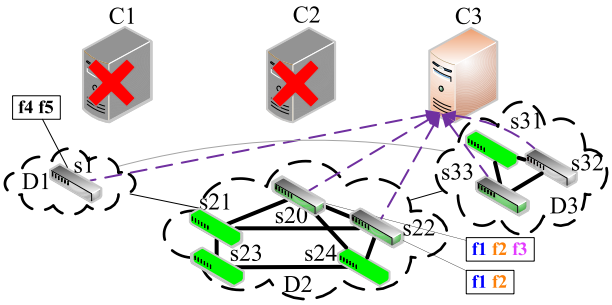
By reducing the number of switches under the SDN mode and increasing the rest control resource from active controllers, we have more freedom to remap these switches to active controllers. The switch remapping should maintain a low communication overhead between these switches and active controllers without exceeding the controller's rest control resource. We name the solution that sequentially solves the above three problems one by one the three-phase solution. Fig. 4(b) shows the result of the three-phase solution under C_1 and C_2 failure.



(a) Realizing the programmability of offline flows $f1$-$f5$ by overloading the active controller. Under C_1 and C_2 failure, mapping switches $s1$, $s21$ and $s23$ with the SDN mode overloads controller C_3.



(b) Realizing the programmability of offline flows $f1$-$f5$ by releasing control resource from active controllers. Under C_1 and C_2 failure, C_3 reduces its control load by configuring $s31$ and $s33$ with the legacy mode. Switches $s1$, $s21$, $s23$, and $s32$ with the SDN mode are mapped to controller C_3 to recover the five flows.



(c) Joint realizing the programmability of offline flows $f1$-$f5$ and low communication overhead. Under C_1 and C_2 failure, RetroFlow+ configures switch $s31$ with the legacy mode to release C_3's control resource and maps switches $s1$, $s20$, and $s22$ with the SDN mode to C_3 since recovering these switches can recover the programmability of all offline flows with low communication overhead.

Fig. 4. Switch remapping under single and two controller failures using the SDN and legacy modes. Switches $s21$ and $s23$ are much far from controllers C_3 than switches $s20$, $s22$, and $s24$.

D. A Joint Optimization Problem

However, it is not enough to consider the above three problems independently because these problems are correlated.

Recall the communication overhead of a controller equals to the propagation latency between the controller and its controlled switches multiplied with the number of flows in the switches. Thus, to achieve the low communication overhead, it may be better to choose more switches (or switches with more flows) with lower delays than to choose fewer switches (or switches with fewer flows) with higher delays.

Fig. 4(c) shows the result of RetroFlow+ under the same failure case of Fig. 4(a). In this figure, RetroFlow+ configures switch $s31$ with the legacy mode to release C_3's control resource and maps switches $s1$, $s20$, and $s22$ with the SDN mode to C_3. Flows $f1$, $f2$, $f3$, $f4$, and $f5$'s forwarding paths can be changed at $s20$ and $s22$, at $s22$, at $s20$, at $s1$, and at $s1$, respectively. RetroFlow+ outperforms the three-phase solution in Fig. 4(b) in three aspects. First, RetroFlow+ has a higher flow programmability since flow $f1$ traverses two switches with the SDN mode. Second, RetroFlow+ releases the minimum control resource by only configuring one switch in $D3$, and thus tries its best to maintain the normal operation in $D3$. Third, RetroFlow+ has a lower communication overhead. RetroFlow+ pulls flow $f1$ twice from two switches, but switches $s20$ and $s22$ are much closer to controller C_3 than switches $s21$, $s23$, and $s24$. Thus, for the communication overhead, the significant decrease of propagation delay compromises the increased number of flows.

Therefore, we should jointly consider the number of flows in offline switches and the propagation delay among the switches and active controllers to configure switches' routing mode, configure active controllers' resource, and establish the controller-switch mapping. We name this problem *Optimal Configuration and Mapping for Switch and Controller (OCMSC)* problem and literally explain it as follows: *under controller failures, we need to maintain the programmability of flows from offline switches with the minimum communication overhead among offline switches and active controllers by efficiently configuring each offline switch with a routing mode, rationally releasing some control resource from active controllers, and mapping the offline switches with the SDN mode to the active controllers*.

IV. Problem Formulation

In this section, we introduce how to optimally configure controllers and switches and remap switches by modeling the system, introducing constraints and the objective function, and finally formulating an optimization problem. For simplicity, in the rest of this paper, we use a switch instead of an offline switch.

A. System Description

Typically, an SD-WAN consists of H controllers at H locations, and each controller controls a domain of switches. Controllers $C_{M+1}, \ldots, C_H$ fail, and they control N switches in total. The set of active controllers is $\mathcal{C} = \{C_1, \ldots, C_j, \ldots C_M\}$, and the set of switches controlled by the failed controllers are $\mathcal{S} = \{s_1, \ldots, s_i, \ldots, s_N\}$. We need to select some switches from $\mathcal{S}$, configure them with the SDN mode, and map these selected switches to controllers in $\mathcal{C}$; the rest of switches in $\mathcal{S}$ are configured with the legacy mode. If switch $s_i \in \mathcal{S}$ is configured with the SDN mode, $x_i = 1$; otherwise, the switch is configured with the legacy mode and does not rely on any controller, and thus $x_i = 0$. We use $z_{ij} = 1$ to denote that switch $s_i \in \mathcal{S}$ under the SDN mode is mapped to controller $C_j \in \mathcal{C}$; otherwise $z_{ij} = 0$. Since both x_i and z_{ij} are binary variables, and a feasible switch-controller mapping requires that a switch is under the SDN mode and mapped to an active controller, we can have

$$x_i * z_{ij} = z_{ij}, \quad \forall i \in [1, N], \ \forall j \in [1, M]. \tag{1}$$

The set of flows from switches $\mathcal{S}$ is $F = \{f^1, f^2, \ldots, f^l, \ldots, f^L\}$. If flow f^l's forwarding path traverses switch s_i, and s_i has at least two paths to f^l's destination, we have $\beta_i^l = 1$, otherwise $\beta_i^l = 0$. If flow f^l is a programmable flow, we have $y^l = 1$, otherwise $y^l = 0$.

B. Constraints

1) Switch-Controller Mapping Constraint: If switch s_i is configured with the SDN mode, it must be controlled by only one controller; if switch s_i is configured with the legacy mode, it is not controlled by any controller. Thus, we have

$$\sum_{j=1}^{M} z_{ij} = x_i, \quad \forall i \in [1, N]. \tag{2}$$

2) Controller Control Resource Constraint: If some controllers fail, active controllers should try their best to control offline switches. The state pulling operations of a switch equal to the total number of flows in the switch's flow table. We measure a controller's control resource as the number of flows that the controller can normally pull from its controlled switches without introducing extra delays (e.g., queueing delay). The processing load of a controller should not exceed the controller's available control resource A_j. It can be written as

$$\sum_{i=1}^{N} (\sum_{l=1}^{L} \beta_i^l * x_i * z_{ij}) \leq A_j, \quad \forall j \in [1, M],$$

Bringing Eq. (1) into the above inequality and letting g_i denote the number of flows in switch s_i

$$g_i = \sum_{l=1}^{L} \beta_i^l, \quad \forall i \in [1, N], \tag{3}$$

we can reformulate the above nonlinear constraints as the following linear constraints:

$$\sum_{i=1}^{N} (g_i * z_{ij}) \leq A_j, \quad \forall j \in [1, M]. \tag{4}$$

In our previous work [1], the controller's rest control resource A_j equals A_j^{rest}, the rest control resource of the controller without interrupting its normal operations. However, A_j^{rest} could not be enough to recover offline flows to be programmable. Thus, to increase the number of recovered flows, we should find a way to get some extra control resources from active controllers and use the extra control resources to recover more offline flows.

Thanks to hybrid routing mode, we can retrieve extra control resources from an active controller by configuring some switches controlled by this controller to work under the legacy mode. Using this solution, the programmability of normal flows, which traverse these switches with the legacy mode, is reduced. We ensure that after releasing control resources, each normal flow is maintained at its minimum programmability with at least two paths.

We use A_j^{avail} to show the list of all available control resource of controller c_j. $A_j \in A_j^{avail} = \{A_j^{rest}, A_j^1, A_j^2, \ldots, A_j^k, \ldots, A_j^{K_j}\}$, where $A_j^{rest} \leq A_j^1 \leq A_j^2 \leq \ldots \leq A_j^k \leq \ldots \leq A_j^{K_j}$. We can only select one control resource for each controller and use $u_j^k = 1$ to denote A_j^k is selected. Thus, we have

$$\sum_{k=1}^{K_j} u_j^k = 1, \quad \forall j \in [1, M]. \tag{5}$$

Eq. (4) can be rewritten as follows:

$$\sum_{i=1}^{N} (g_i * z_{ij}) \leq \sum_{k=1}^{K_j} (A_j^k * u_j^k), \quad \forall j \in [1, M]. \tag{6}$$

3) Flow Programmability Constraint: If a switch works under the SDN mode, the flows in the switch become programmable. The flow f^l's programmability can be expressed as follows:

$$y^l \leq \sum_{i=1}^{N} (\beta_i^l * x_i), \quad \forall l \in [1, L].$$

In the above inequality, the equal sign comes when there is only one offline switch with the SDN mode that contains flow f^l. If multiple switches contain this flow, the inequality sign is used. Since all flows from offline switches should be programmable, we have $y^l = 1$ for all $l \in [1, L]$. Bringing Eq. (2), we have

$$1 \leq \sum_{i=1}^{N} (\beta_i^l * x_i), \quad \forall l \in [1, L]. \tag{7}$$

C. Objective Function

The objective is to minimize the communication overhead of active controllers to pull flow state from offline switches, which equals the total propagation delay of programmable flows between the switches with the SDN mode and their newly mapped controllers. We use D_{ij} $(D_{ij} \geq 0)$ to denote the propagation delay between switch s_i and controller C_j and formulate the overhead as follows:

$$obj = \sum_{j=1}^{M} \sum_{i=1}^{N} (g_i * D_{ij} * z_{ij}).$$

If we use w_{ij} to denote controller C_j's communication overhead to switch s_i:

$$w_{ij} = g_i * D_{ij}, \quad \forall i \in [1, N], \ \forall j \in [1, M],$$

we can write the objective function as follows

$$obj = \sum_{j=1}^{M} \sum_{i=1}^{N} (w_{ij} * z_{ij}). \tag{8}$$

D. OCMSC Problem Formulation

The goal of our problem is to minimize the communication overhead between active controllers in $\mathcal{C}$ and offline switches in $\mathcal{S}$ and provide the programmability for flows in F by smartly configuring controllers in $\mathcal{C}$ and switches in $\mathcal{S}$ and mapping switches with the SDN mode to active controllers in $\mathcal{C}$. Therefore, we formulate the OCMSC problem as follows:

$$\begin{aligned} \min_{z,y} \ & \sum_{j=1}^{M} \sum_{i=1}^{N} (w_{ij} * z_{ij}) \\ \text{s.t. } & (2)(5)(6)(7), \\ & x_i, z_{ij}, y^l, u_j^k \in \{0, 1\}, \\ & \forall i \in [1, N], \ \forall j \in [1, M], \ \forall l \in [1, L], \ \forall k \in [1, K_j], \end{aligned} \tag{P}$$

where $\{w_{ij}\}$, $\{g_i\}$, $\{\beta_i^l\}$, and $\{A_j^{rest}\}$ are constants, and $\{x_i\}$, $\{z_{ij}\}$, $\{y^l\}$, and $\{u_j^k\}$ are design variables. In the OCMSC problem, the objective function is linear, and variables are binary integers. Thus, this problem is an integer programming.

V. RetroFlow+

In this section, we first analyze the complexity of the OCMSC problem and then propose our RetroFlow+ algorithm for solving the problem.

A. Complexity Analysis of the OCMSC Problem

Theorem 1: For a special case with two conditions: (1) each controller releases the maximum control resource, and (2) each flow traverses only two switches and has different source and destination switches with others, the OCMSC problem is NP-hard.

Proof: We first introduce the *Generalized Assignment Problem (GAP)* [25]. The GAP aims to minimize the cost assignment of n tasks to m agents such that each task is precisely assigned to one agent subject to capacity restrictions on the agents. A typical formulation of the GAP is shown below:

$$\begin{aligned} \min_{x} \ & \sum_{j=1}^{m} \sum_{i=1}^{n} (c_{ij} * x_{ij}) \\ \text{s.t. } & \sum_{i=1}^{n} (a_{ij} * x_{ij}) \leq b_j, \quad \forall j \in [1, m], \\ & \sum_{j=1}^{m} x_{ij} = 1, \quad \forall i \in [1, n], \\ & x_{ij} \in \{0, 1\}, \quad \forall i \in [1, n], \ \forall j \in [1, m], \end{aligned}$$

where c_{ij} is the cost of assigning task i to agent j, a_{ij} is the capacity of task i when the task is assigned to agent j, and b_j is the available capacity of agent j. Binary variable x_{ij} equals 1 if task i is assigned to agent j; otherwise it equals 0.

It has been proved when assigning multiple tasks to an agent and ensuring each task is performed exactly by one agent, the GAP is NP-hard [25].

We then prove for a special case of conditions (1) and (2), problem (P) and the GAP are equivalent problems. Given condition (1) that each controller releases the maximum control resource, $u_j^{K_j} = 1$. Eqs. (5) and (6) can be combined as follows

$$\sum_{i=1}^{N}(g_i * z_{ij}) \leq A_j^{K_j}, \quad \forall j \in [1, M]. \tag{9}$$

Recall a flow cannot change its path at its destination switch. Given condition (2) that each flow traverses only two switches and has different source and destination switches with others, we have that each offline switch has a unique flow, and the number of offline switches equals the number of unique flows. That is $\beta_{i_0}^{i_0} = 1$ for a specific $i_0 \in [1, N]$. Thus, we can change Eq. (7) as the following equation

$$1 = \sum_{i=1}^{N}(\beta_i^l * x_i) = \beta_{i_0}^{i_0} * x_{i_0} = x_{i_0}, \quad \forall i_0 \in [1, N].$$

Bringing the above equation into (2), we have

$$\sum_{j=1}^{M} z_{ij} = 1, \quad \forall i \in [1, N]. \tag{10}$$

Following the above two conditions, our OCMSC problem can be reformulated as follows:

$$\begin{aligned} &\min_{z} \sum_{j=1}^{M}\sum_{i=1}^{N}(w_{ij} * z_{ij}) \\ &\text{s.t. (9)(10),} \\ &\quad z_{ij} \in \{0, 1\}, \quad \forall i \in [1, N], \ \forall j \in [1, M]. \end{aligned} \tag{P'}$$

Problem (P') aims to minimize the communication cost of N switches to M controllers such that each switch is precisely assigned to one controller subject to control resource restrictions on the controllers. We can treat switch s_i and controller C_j in problem (P') as task i and agent j in the GAP. By this construction, it is easy to prove that there exists the minimum communication cost by mapping switches in $\mathcal{S}$ to controllers $\mathcal{C}$, if and only if there exists the optimal solution of the GAP by assigning n tasks to m agents. The construction can be done in polynomial time. In problem (P'), the mapping between switches and controllers could be many to one. Since the GAP is NP-hard when multiple tasks are assigned to an agent, and each task is performed exactly by one agent [25], problem (P') is NP-hard. □

Problem (P') is a special case of the OCMSC problem and is NP-hard. Therefore, we can have the following conclusion:

Theorem 2: The OCMSC problem is NP-hard.

B. RetroFlow+ Algorithm

Typically, we can use existing integer program optimization solvers to obtain the OCMSC problem's optimal solution. However, for the problem with a large network, the solver could require a very long time, or sometimes it is impossible to find a feasible solution. Therefore, we propose a heuristic algorithm called RetroFlow+ for solving the problem to achieve the trade-off between the performance and time complexity.

TABLE I
NOTATIONS

Notation	Meaning
$\mathcal{S}$	the set of offline switches, $\mathcal{S} = \{s_i \mid i \in [1, N]\}$
$\mathcal{W}(i)$	the communication overhead of switch s_i, $\mathcal{W}(i) = \{w_{i1}, ..., w_{ij}, ..., w_{iM}\}, i \in [1, N]$
$\mathcal{C}(i)$	the set of active controllers by sorting $\mathcal{C} = \{C_j \mid j \in [1, M]\}$ following the ascending order of $\mathcal{W}(i)$, $i \in [1, N]$
$\mathcal{A}^{rest}$	the set of rest control resource of active controllers for recovering offline flows, $\mathcal{A}^{rest} = \{A_j^{rest} \mid j \in [1, M]\}$
$\mathcal{A}^{avail}$	the set of available control resource of active controllers for recovering offline flows, $\mathcal{A}^{avail} = \{\mathcal{A}_j^{avail} \mid j \in [1, M]\}$, $\mathcal{A}_j^{avail} = \{A_j^{rest}, A_j^1, A_j^2, ..., A_j^k, ..., A_j^{K_j}\}$
$\mathcal{G}$	the set of the number of flows in switches, $\mathcal{G} = \{g_i \mid i \in [1, N]\}$
$\mathcal{B}$	the set of flow-switch relationship, $\mathcal{B} = \{B_1, ..., B_i, ...B_N \mid i \in [1, N]\}$, $B_i = \{\beta_i^1, ..., \beta_i^l, ..., \beta_i^L\}$
$\mathcal{X}$	the set of offline switches with the SDN mode, $\mathcal{X} = \{i \in [1, N] \mid x_i = 1\}$
$\mathcal{Z}$	the set of the mapping relationship between offline switches with the SDN mode and active controllers, $\mathcal{Z} = \{(i, j) \in [1, N] \times [1, M] \mid z_{ij} = 1\}$
$\mathcal{Y}$	the set of recovered flows, $\mathcal{Y} = \{l \in [1, L] \mid y_l = 1\}$
δ	a number that indicates the maximum number of flows different from recovered flows
$\mathcal{U}$	the set of variables that record the index of the used control resource of controllers in $\mathcal{A}^{avail}$, $\mathcal{U} = \{u_1, ..., u_j, ..., u_M\}$

The main idea behind RetroFlow+ is to recover as many offline flows as possible by releasing as few control resource as possible. During the flow recovery, we select and test variables based on their importance. The first priority of our problem is to enable many unique flows from offline switches to become programmable flows. Thus, we first select a switch that has the maximum number of flows different from recovered flows. This switch selection method helps us to efficiently rescue as many unique flows as possible in each iteration. For this selected switch, we choose a controller in the ascending order of the communication overhead and then test whether the controller's rest control resource is enough to recover flows from this switch. If yes, the controller is selected; otherwise, we release extra control resources from this tested controller by reducing the programmability of flows controlled by this controller and use the released resource to recover this selected switch. The next controller will be tested if releasing the maximum control resource of the tested controller still cannot recover this selected switch. This controller selection method aims to recover flows in an offline switch by releasing as few control resource as possible. After selecting the controller, a mapping between this controller and the selected switch is established, and all flows in the switch are recovered. This established mapping enables low communication overhead.

Details of RetroFlow+ are summarized in Algorithm 1, and Table I shows the notations used in the algorithm. In line 1, the sets $\mathcal{X}$, $\mathcal{Z}$, and $\mathcal{Y}$ are first set to be empty. In line 2, we start iteratively to select switches and controllers and establish the mappings. In line 4, for each iteration, we set δ to 0, and set i_0 and j_0 to NULL. In lines 5-14, we find

Algorithm 1 RetroFlow+

Input: $\mathcal{S}$, $\mathcal{C}(i)$, $\mathcal{A}^{rest}$, $\mathcal{A}^{avail}$, $\mathcal{G}$, $\mathcal{B}$;
Output: $\mathcal{X}$, $\mathcal{Z}$, $\mathcal{Y}$;

1: $\mathcal{X} = \emptyset$, $\mathcal{Z} = \emptyset$, $\mathcal{Y} = \emptyset$;
2: **while** True **do**
3: //select switch s_{i_0}, which has the maximum number of flows different from recovered flows
4: δ = 0, i_0 = NULL, j_0 = NULL;
5: **for** $s_i \in \mathcal{S}$ **do**
6: **for** $l \in \{\beta_i^l = 1, l \in [1, L]\}$ **do**
7: **if** $l \in \mathcal{Y}$ **then**
8: $\beta_i^l = 0$;
9: **end if**
10: **end for**
11: **if** $|\sum_{l=1}^{L} \beta_i^l| > \delta$ **then**
12: $\delta = |\sum_{l=1}^{L} \beta_i^l|$, $i_0 = i$; //select a switch
13: **end if**
14: **end for**
15: //select controller C_{j_0}, which has low communication overhead and enough control resource for switch s_{i_0}
16: **if** $\max \mathcal{A}^{rest} < g_{i_0}$ **then**
17: //release some control resource to recover switch s_{i_0} when existing control resource are not enough
18: j_0, $A_{j_0}^{rest}$ = RelaxedControllerSelection($\mathcal{A}^{rest}$, $\mathcal{A}^{avail}$, $\mathcal{S}$, $\mathcal{C}(i)$, g_{i_0}, $\mathcal{B}$, $\mathcal{X}$, $\mathcal{Z}$, $\mathcal{Y}$)
19: **else**
20: **for** $C_j \in \mathcal{C}(i_0)$ **do**
21: **if** $A_j^{rest} - g_{i_0} \geq 0$ **then**
22: $j_0 = j$; //select a controller
23: $A_{j_0}^{rest} = A_{j_0}^{rest} - g_{i_0}$; //update $A_{j_0}^{rest}$
24: break;
25: **end if**
26: **end for**
27: **end if**
28: //establish the mapping between s_{i_0} and C_{j_0}
29: **if** $j_0 \neq$ NULL **then**
30: $\mathcal{X} \leftarrow \mathcal{X} \cup i_0$, $\mathcal{Z} \leftarrow \mathcal{Z} \cup (i_0, j_0)$;
31: **for** $k \in [0, K_j]$ **do**
32: $A_{j_0}^k = A_{j_0}^k - g_{i_0}$; //update $A_{j_0}^{avail}$
33: **end for**
34: **for** $l \in \{\beta_i^l = 1, l \in [1, L]\}$ **do**
35: $\mathcal{Y} \leftarrow \mathcal{Y} \cup l$;
36: **end for**
37: **end if**
38: $\mathcal{S} \leftarrow \mathcal{S} \setminus s_{i_0}$;
39: **if** $|\mathcal{S}| == \emptyset$ or $|\mathcal{Y}| == L$ **then**
40: break;
41: **end if**
42: **end while**
43: return $\mathcal{X}, \mathcal{Z}, \mathcal{Y}$;

Algorithm 2 RelaxedControllerSelection()

Input: $\mathcal{A}^{rest}$, $\mathcal{A}^{avail}$, $\mathcal{S}$, $\mathcal{C}(i)$, g_{i_0}, $\mathcal{B}$, $\mathcal{X}$, $\mathcal{Z}$, $\mathcal{Y}$;
Output: j_0, $A_{j_0}^{rest}$;

1: **for** $C_j \in \mathcal{C}(i_0)$ **do**
2: $k_0 = u_j$;
3: **for** $k \in [k_0, K_j]$ **do**
4: **if** $A_j^k - g_{i_0} \geq 0$ **then**
5: **if** $k > k_0$ **then**
6: $k_0 = k$, $u_j = k_0$;
7: $j_0 = j$; //select a controller
8: $A_{j_0}^{rest} = A_{j_0}^{k_0} - g_{i_0}$ //update $A_{j_0}^{rest}$
9: break;
10: **end if**
11: **end if**
12: **end for**
13: **end for**
14: return j_0, $A_{j_0}^{rest}$;

the required switch. We first remove recovered flows from each switch in $\mathcal{S}$ (lines 6-10) and find the required switch and update this switch's index to i_0 (lines 11-13). In lines 16-27, we select controller C_{j_0} for switch s_{i_0}. In lines 16-17, when each controller's rest control resource is not enough to recover s_{i_0}, we call function RelaxedControllerSelection() in Algorithm 2 to recover s_{i_0} by releasing control resource from active controllers. If controller C_j's rest control resource is able to control switch s_{i_0}, we select it as C_{j_0} and update the rest control resource of C_j (lines 20-26). After selecting the controller, we establish the mapping between switch s_{i_0} and controller C_{j_0}, update the set of available control resources of C_{j_0}, and upgrade the set of recovered flows (lines 29-37). In line 38, we remove the tested switch from the set of offline switches $\mathcal{S}$. In lines 39-41, if all switches are tested or all offline flows are recovered, the algorithm jumps out of the iterations. In line 43, the result returns.

Algorithm 2 shows details of function RelaxedControllerSelection(). This function aims to select a controller for switch s_{i_0} by reducing the programmability of flows controlled by this controller and use the released resource to recover switch s_{i_0}. In line 1, we start iteratively to test controllers. In line 2, we find k_0 from $\mathcal{U}$. For controller C_j, we use A_j^{avail} to denote the set of available control resources of this controller by reducing the programmability of flows controlled by this controller. Specifically, $\mathcal{A}_j^{avail} = \{A_j^{rest}, A_j^1, A_j^2, \ldots, A_j^k, \ldots, A_j^{K_j}\}$. We also use $\mathcal{U}$ to denote the used control resource from $\mathcal{A}^{avail}$. If $A_j^{k_0}$ is used by the controller, we have $u_j = k_0$. In lines 3-12, available control resources in A_j^{avail} from k_0 to K_j are tested. The least available control resource, which is able to recover s_{i_0} is used, and then the iteration stops after $\mathcal{U}_j$ and the rest control resource of C_j are updated. In line 14, the selected controller and its rest control resource are returned.

VI. Simulation

A. Simulation Setup

We evaluate the performance of RetroFlow+ with a real backbone topology named AT&T from Topology Zoo [26]. The AT&T topology is a national topology of the US with 25 nodes and 112 links. In this topology, each node is given a unique ID with a latitude and a longitude. We calculate

TABLE II
DEFAULT RELATIONSHIP BETWEEN CONTROLLERS, SWITCHES, AND THE NUMBER OF FLOWS IN THE SWITCHES UNDER AT&T TOPOLOGY

Controller ID	2				5				6				13					20		22					
Switch ID	2	3	9	16	4	5	8	14	0	1	6	7	10	11	12	13	15	19	20	17	18	21	22	23	24
Number of flows	143	71	107	55	49	143	53	61	81	49	89	97	63	59	71	213	67	49	63	125	49	81	111	49	57

TABLE III
DEFAULT RELATIONSHIP BETWEEN CONTROLLERS, SWITCHES, AND THE NUMBER OF FLOWS IN THE SWITCHES UNDER BELNET TOPOLOGY

Controller ID	1				4					11			17				18				
Switch ID	0	1	3	6	4	5	8	9	12	2	10	11	7	13	17	20	14	15	16	18	19
Number of flows	65	89	53	113	117	91	63	49	61	65	113	89	87	59	209	55	43	59	91	243	59

the distance between two nodes using Haversine formula [27] and use the distance divided by the propagation speed (i.e., 2×10^8 m/s) [28] to represent the propagation delay between the two nodes. In our simulation, each node is an SDN switch, and some selected nodes are further deployed controllers. Any two nodes have a traffic flow, and each flow is forwarded on its shortest path. We set the control resource upper bound of a controller to 500. The default selection of controllers and default mapping between controllers and switches are obtained by solving an optimization problem, which aims to minimize the communication overhead among all switches and controllers. Tables II and III show the default relationship of controllers, switches, and the number of flows in the switches under AT&T and Belnet topologies.

AT&T is one of the world's largest Internet service providers, and it has started to softwarize its WAN with programmable devices. Belnet is the national topology of Belgium. Some existing works [8], [9], [29]–[33] study the problem of maintaining network resiliency in WAN using AT&T and Belnet topologies as the simulation topology. Thus, following existing works, we use the AT&T and Belnet topology as representative WAN topologies.

Our paper presents the results of flow-level simulations using Python 2.7. Our solution is designed for maintaining control resiliency and flow programmability in WANs. Typically, a WAN consists of two-tuple source-destination flows. Many existing works [33]–[37] on WANs use a similar simulation to study the controller failure problem.

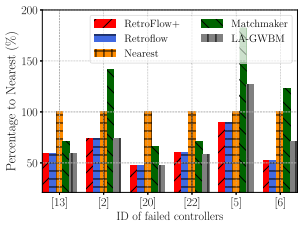
B. Comparison Algorithms

1) Nearest: during controller failures, each offline switch maps to its nearest controller. This solution can minimize the propagation delay but could overload active controllers.
2) RetroFlow [1]: this algorithm configures a set of offline switches working under legacy routing modes to relieve controllers from controlling these offline switches, and transfers the control of offline switches with SDN mode to active controllers to recover offline flows and minimize the communication overhead from these offline switches to controllers.
3) RetroFlow+: this algorithm is shown in Algorithm 1.
4) Matchmaker [33]: this algorithm adaptively adjusts the control cost of offline switches based on the limited control resource by changing the paths of flows to realize proper offline switches remapping.
5) LA-GWBM [37]: this is a greedy algorithm to assign master and slave controllers to switches against multiple controller failures.

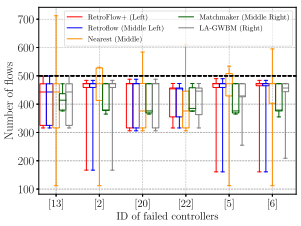
C. Simulation Results

In our simulation, the SDN control plane consists of six controllers under AT&T topology and five controllers under Belnet topology. Many existing works consider only one controller failure [10], [11]. RetroFlow+ can cover a wide range of multiple controller failure scenarios. We compare RetroFlow+ with other algorithms under three scenarios: (1) one controller failure, (2) two controller failures, and (3) three controller failures. Scenario (1) is a moderate controller failure that active controllers usually have enough rest control resources to recover all offline switches. Scenarios (2) and (3) are serious controller failures, where active controllers are not always able to handle all offline switches with their given rest control resource. The scenario that multiple controllers fail simultaneously is not as common as single controller failure. Our simulations totally use six controllers under AT&T topology, and three controllers are 50% of the total controllers. For Belnet topology, the scenario that two of five controllers fail simultaneously is sufficient for the evaluation. Thus, the simulation results prove the robustness of RetroFlow+. Our performance metrics are the percentage of recovered flows from offline switches, communication overhead, and processing load of active controllers. We use Nearest as the baseline algorithm and normalize the metric of each algorithm to that of Nearest.

1) Under AT&T Topology: ***One controller failure:*** Fig. 5 shows the results of five algorithms when one of the six controllers fail. All five algorithms recover 100% flows from offline switches and remap the same number of offline switches to active controllers. In Fig. 5(a), RetroFlow+ and RetroFlow outperform Nearest, LA-GWBM, and Matchmaker in terms of the communication overhead. This is because RetroFlow+ and RetroFlow remap offline switches to their closest controllers with enough control resource to recover switches but Nearest only considers the propagation delay to remap offline switches to controllers and thus could overload controllers, leading to long queueing delay for pulling flow state. Matchmaker performs worse than Nearest because it



(a) Communication overhead. The lower, the better.



(b) Processing load of active controllers. The black dash line indicates the controller's control resource upper bound.

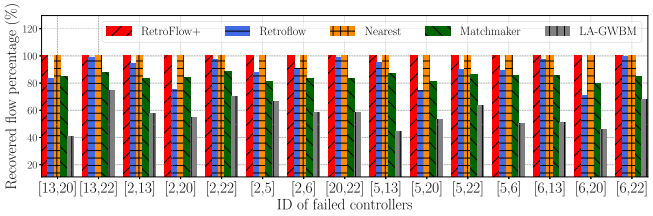
Fig. 5. Results of one controller failure under AT&T topology.

does not take minimizing the communication overhead into consideration. The queueing delay setup follows the existing work [17]. Compared with Nearest, RetroFlow+ and RetroFlow can reduce the communication overhead up to 52.6%. Fig. 5(b) shows the processing load of controllers. In this figure, Nearest experiences controller overload at all six cases.

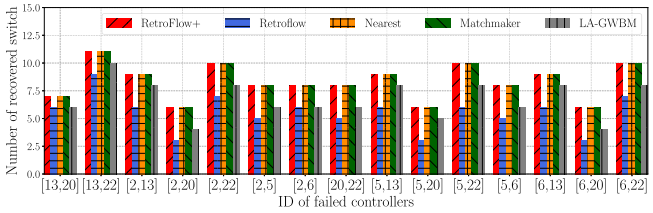
Two controller failures: Fig. 6 shows the results of five algorithms when two of the six controllers fail. There are 15 cases of two controller failures. Fig. 6(a) shows the percentage of recovered flows. In this figure, RetroFlow+ recovers all offline flows, but RetroFlow only recovers flows in the range of 71% to 99%. Matchmaker's performance is similar to RetroFlow and recovers 79% to 87% offline flows. LA-GWBM performs worst since it is a static solution and only recovers offline flows in the range of 40% to 75%. Nearest recovers 100% flows at the cost of high communication overhead, as shown in Fig. 6(c). In Fig. 6(b), RetroFlow+, Matchmaker, and Nearest recover all offline switches.

To review the benefit of RetroFlow+, we analyze the failure case of controllers C_6 and C_{20}. In this case, we have six offline switches s_0, s_1, s_6, s_7, s_{19}, and s_{20} with control cost of 81, 49, 89, 97, 49, and 63, respectively. The rest control resource of controllers C_2, C_5, C_{13}, and C_{22} are only 124, 194, 27, and 28. In Fig. 6(b), RetroFlow recovers 71% flows by recovering three switches. It maps switch s_6 to controller C_2 and maps switches s_0 and s_7 to controller C_5. After the mapping, the rest control resources of controllers C_2, C_5, C_{13}, and C_{22} are 35, 16, 27, and 28. Thus, none of the four controllers can recover switches s_1, s_{19}, or s_{20}. In contrast, RetroFlow+ recovers 29% more flows by recovering three more switches than RetroFlow does. Specifically, RetroFlow+ increases C_2 and C_{22}'s control resources to 179 and 126, respectively. As a result,C_2 recovers s_1 and s_6, C_5 recovers s_0 and s_7, and C_{22} recovers s_{19} and s_{20}. Nearest recovers 100% offline flows at the cost of high communication overhead and controller overloading. In Fig. 6(c), Nearest requires 69% more communication overhead than RetroFlow+ does due to the queueing delay of controller overloading, as shown in Fig. 6(d). RetroFlow+'s overhead is a little higher than RetroFlow and LA-GWBM for some cases since RetroFlow+ recovers more flows.

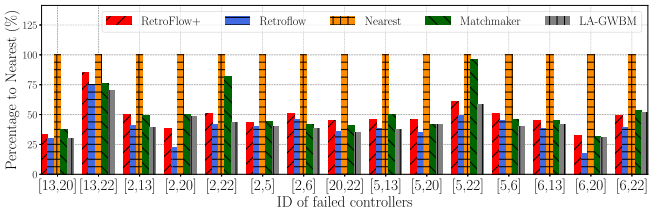
Three controller failures: Fig. 7 shows the results of five algorithms when three of the six controllers fail. There are



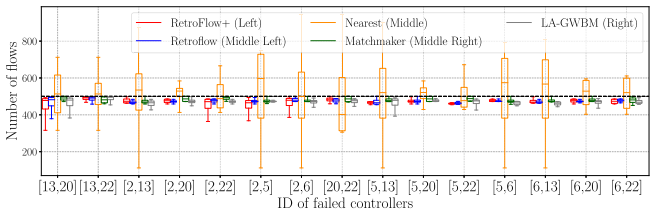
(a) Percentage of recovered flows. The higher, the better.



(b) Number of recovered offline switches.



(c) Communication overhead. The lower, the better.

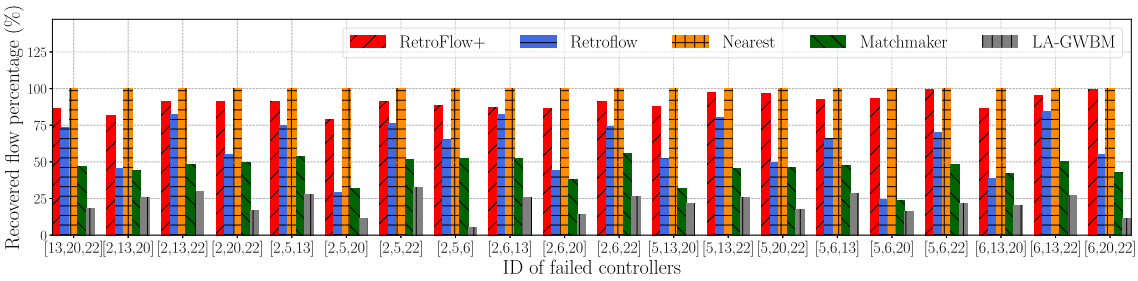


(d) Processing load of active controllers. The black dash line indicates the controller's control resource upper bound.

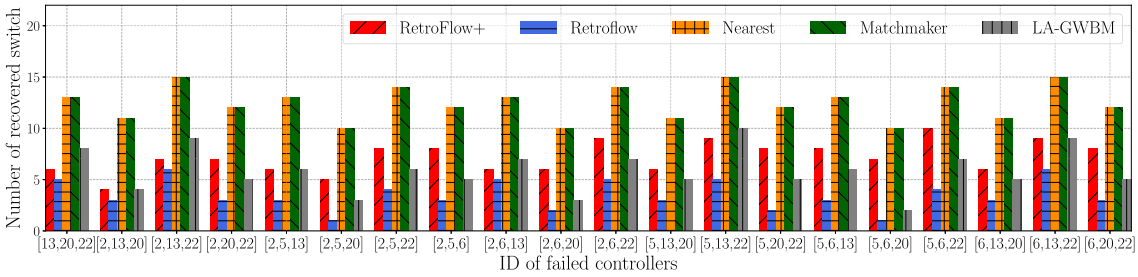
Fig. 6. Results of two controller failures under AT&T topology.

20 cases of three controller failures. In Fig. 7(a), RetroFlow+ recovers offline flows in the range of 79% to 99%, but RetroFlow only recovers flows in the range of 30% to 86%. Matchmaker and LA-GWBM perform even worse, from a range of 26% to 58% and 5% to 32%, respectively. Nearest recovers 100% flows at the cost of high communication overhead, as shown in Fig. 7(c). In Fig. 7(b), RetroFlow+ cannot recover all offline switches as Nearest does but always recovers 1-6 more switches than RetroFlow does. By rationally releasing a few control resources from controllers, RetroFlow+ provides more opportunities to recover offline flows.

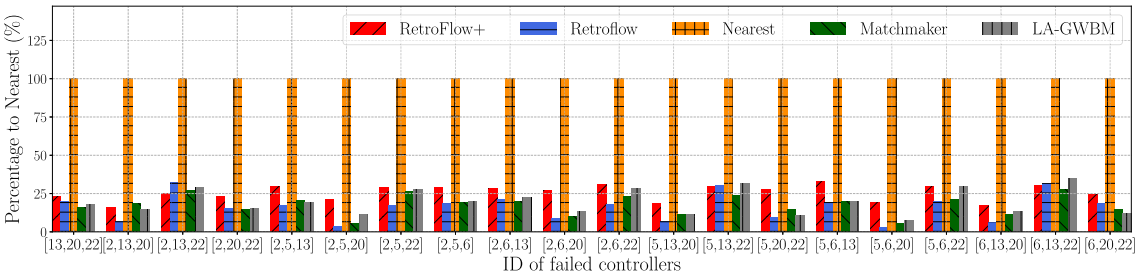
Let us look at the failure case of controllers C_5, C_6, and C_{20}. RetroFlow recovers 22% offline flows, but RetroFlow+ increases 70% flow recovery and recovers 92% offline flows. In this case, we have ten offline switches s_0, s_1, s_4, s_5,s_6, s_7, s_8, s_{14}, s_{19}, and s_{20} with control cost of 81, 49, 49, 143, 89, 97, 53, 61, 49, and 63, respectively. The rest control resource of controllers C_2, C_{13}, and C_{22} are only 124, 27, and 28 flows. In Fig. 7(b), RetroFlow only recovers switch s_7 by mapping it to controller C_2. After the mapping, the rest control



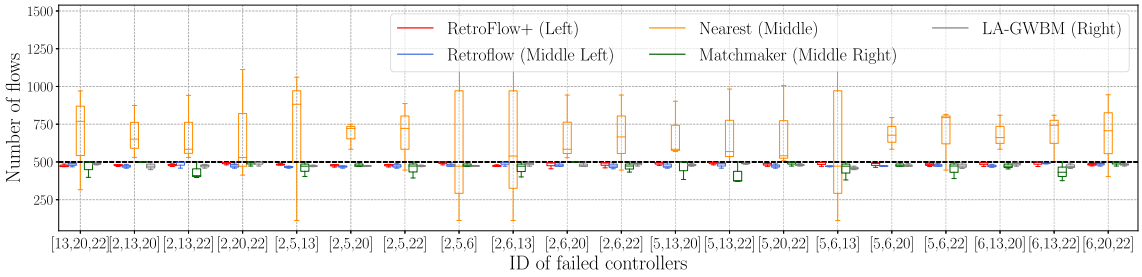
(a) Percentage of recovered flows. The higher, the better.



(b) Number of recovered offline switches.



(c) Communication overhead. The lower, the better.

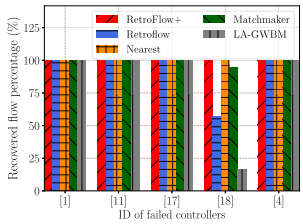


(d) Processing load of active controllers. The black dash line indicates the controller's control resource upper bound.

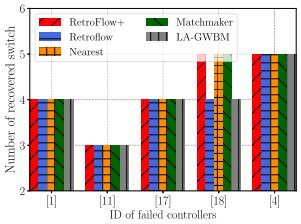
Fig. 7. Results of three controller failures under AT&T topology.

resource of controllers C_2, C_5, C_{13}, and C_{22} are 27, 27, and 28. Thus, none of the three controllers can recover the rest nine switches. In contrast, RetroFlow+ recovers seven switches and 92% flows. Specifically, RetroFlow+ increases C_2, C_{13}, and C_{22}'s control resources to 267, 240, and 126. Consequently, RetroFlow+ totally recovers seven switches: C_2 recovers s_0, s_1, and s_6, C_{13} recovers s_5 and s_{14}, and C_{22} recovers s_8 and s_{20}. After the flow recovery, the rest control resource of C_2, C_{13}, and C_{22} are 0, 36, and 10 flows, and they are not enough to recover s_1, s_4, and s_{19}, each of which requires 49. Similarly, Nearest recovers 100% offline flows at the cost of high communication overhead and controller overloading. In Fig. 7(c), Nearest requires 80% more communication overhead than RetroFlow+ does due to the queueing delay of controller overloading, as shown in Fig. 7(d). RetroFlow+'s overhead is a little higher than RetroFlow, LA-GWBM, and Matchmaker for some cases because RetroFlow+ recovers more flows.

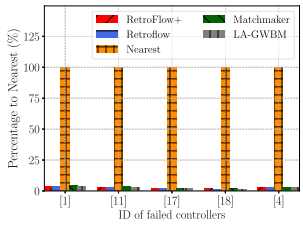
2) Under Belnet Topology: ***One controller failure:*** Fig. 8 shows the results of five algorithms when one of the five controllers fail. There are 5 combinations of one controller failure under Belnet topology. Fig. 8(a) shows the percentage of programmable flows from offline switches. All five algorithms recover 100% flows from offline switches in majority



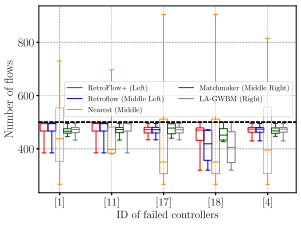
(a) Percentage of recovered flows. The higher, the better.



(b) Number of recovered offline switches.



(c) Communication overhead. The lower, the better.

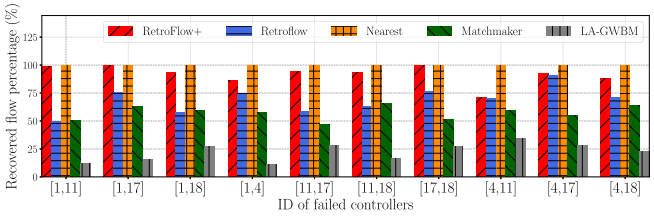


(d) Processing load of active controllers. The black dash line indicates the controller's control resource upper bound.

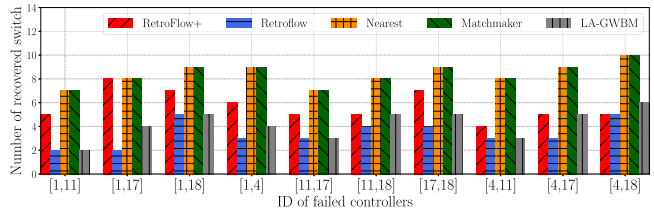
Fig. 8. Results of one controller failure under Belnet topology.

of cases except from the special failure case 18, and they remap the same number of offline switches to active controllers in these four cases as shown in Fig. 8(b). For the special failure case 18, the control cost of switch s_{18} is 243, but the available control resource of active controllers C_1, C_4, C_{11}, and C_{17} are 180, 119, 233, and 88, respectively. Thus, neither RetroFlow nor LA-GWBM can recover switch s_{18}. Matchmaker outperforms RetroFlow and LA-GWBM by dynamically altering the control cost of switch s_{18} based on the control resource of active controllers but cannot realize 100% recovery of offline flows as shown in Fig. 8(a). However, RetroFlow+ increases C_{11}'s control resource to 346 and realize 100% recovery. Nearest recovers 100% offline flows at the cost of high communication overhead and controller overloading. RetroFlow and LA-GWBM cannot recover all offline switches, as shown in Fig. 8(b). In Fig. 8(c), Nearest requires 69% more communication overhead than RetroFlow+ does due to the queueing delay of controller overloading, as shown in Fig. 8(d). RetroFlow+, RetroFlow, Matchmaker, and LA-GWBM reduce more than 95% of communication overhead compared with Nearest, and Nearest performs worse under the Belnet topology than under the AT&T topology. Recall that the total communication overhead comes from queueing delay caused by controller overload [17] and propagation delay from distance variation. In the two topologies, queueing delay contributes more than propagation delay in the overhead. However, in Belnet topology, the propagation delay of the four algorithms reduces since the Belnet topology is much smaller than the AT&T topology. RetroFlow+, RetroFlow, Matchmaker, and LA-GWBM do not introduce queueing delay, but Nearest's queueing delay increases as controllers experience more seriously overloaded in the Belnet topology, as shown in Fig. 8(d).

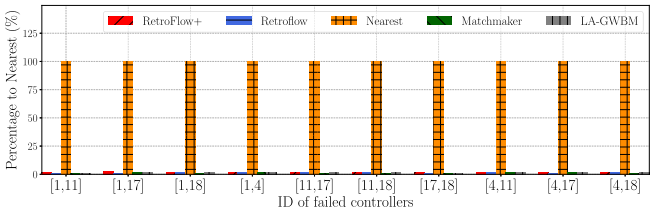
Two controller failures: Fig. 9 shows the results of five algorithms when two of the five controllers fail. There



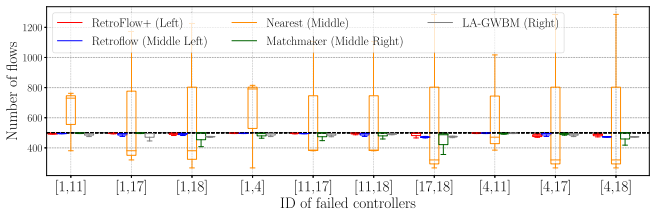
(a) Percentage of recovered flows. The higher, the better.



(b) Number of recovered offline switches.



(c) Communication overhead. The lower, the better.



(d) Processing load of active controllers. The black dash line indicates the controller's control resource upper bound.

Fig. 9. Results of two controller failures under Belnet topology.

are 10 cases of two controller failures under Belnet topology. Fig. 9(a) shows the percentage of recovered flows. RetroFlow+ recovers offline flows in the range of 70% to 100%. On the contrary, RetroFlow only recovers flows in the range of 50% to 91%. Matchmaker performs slightly worse than RetroFlow, which recovers 48% to 65% flows. Only about 12% to 34% of flows are recovered by LA-GWBM. Nearest recovers 100% flows at the cost of high communication overhead, which is around 98% higher than the other four algorithms, as shown in Fig. 9(c). In Fig. 9(b), RetroFlow+'s performance is worse than Nearest's and Matchmaker's but better than RetroFlow's and LA-GWBM's in the majority of cases. Nearest brings controller overloading in every case, as shown in Fig. 9(d).

VII. Related Work

A. Handling Controller Failure

Pareto-based optimal controller-placement [7] minimizes different objectives (e.g., the latency between switches and controllers, latency between controllers) under controller failures. Works in [8], [38] try to find the best trade-off between

the performance and cost during the controller failure under several constraints (e.g., load balancing and QoS). The solution in [39] proposes a controller placement model that ensures resiliency against the controller failure by minimizing the distance from a switch to its i-th closest controller. Capacitated Next Controller Placement [9] proposes a controller placement problem that not only considers the capacity and reliability of master controllers but also plans ahead for the master controller failure by considering a backup controller for each master controller. Tanha *et al.* [34] consider both the switch-controller/inter-controller latency requirements and the capacity of the controllers to select the resilient placement of controllers. Alenazi and Çetinkaya [40] introduce a new metric, which measures a node's importance in terms of its diverse connectivity to other nodes, and consider this metric to determine the locations of the controllers against network failure. Yang and Yeung [41] propose to protect all single link failures in a given network by updating a smallest subset of IP routers to SDN switches. Different from all the aforementioned solutions, RetroFlow [1] reduces the impact of controller failures by leveraging the features of hybrid SDN switches to maintain the advantage of SDN (i.e., flow programmability) and low communication overhead without overloading the rest active controllers. He *et al.* [35] formulate a master and slave controller assignment model to prevent multiple controller failures under the constraints of the propagation latency between switches and controllers. In [36], they take the maximum utilization of controllers into consideration. They also introduce a policy-based approach to rationally specify the master controllers in each failure scenario in their latest work [37]. Matchmaker [33] smartly changes the paths of some offline flows to adjust the control cost of offline switches for better recovery performance. Guillen *et al.* [42] consider multi-controller failures and propose an efficient mechanism to deal with it. The scheme in [43] realizes a blockchain-based solution for the recovery of an SDN controller to get back to a previously known state.

B. Handling Switch/Link Failure

OpenFlow specification 1.5 [44] proposes to connect a switch with its controller via multiple connections, including a main connection and several auxiliary connections. Vizarreta *et al.* [45] utilize multiple connections on disjoint paths between switches and controllers to tolerate node or link failures. The scheme in [46] guarantees that the probability of having more than one operational path between each switch and its controller is at least a given value. Zhong *et al.* [47] propose a min-cover based controller placement approach to meet both reliability of the control network and propagation delay when a single link failure happens. Beheshti and Zhang [48] propose algorithms for placing controllers to achieve the short switch-to-controller path length and high control resiliency. Zhang *et al.* [49] design a min-cut-like graph partitioning algorithm that maximizes the resiliency between controllers and switches. Yang *et al.* [50] propose a heuristic algorithm to solve the SDN controller placement problem for single-link and multi-link failures. Das and Gurusamy [51] propose to optimize both synchronization cost and resilience performance against single-link failure.

VIII. Conclusion

In this paper, we propose RetroFlow+ to jointly achieve resilient network control and flow programmability during controller failures. RetroFlow+ maintains the programmability of flows from offline switches while reducing the controllers' load from the offline switches by taking advantage of commercial hybrid SDN switches that support legacy mode without controllers. By jointly considering the propagation delay and controllers' states in real time, RetroFlow+ also achieves a low communication overhead between offline switches and active controllers. We hope that our work can inspire researchers to creatively utilize existing features in commercial SDN switches to better solve existing problems.

Acknowledgment

The research was partially conducted when Zehua Guo was a Post-doctoral Research Associate at the University of Minnesota, and Sen Liu and Wendi Feng were visiting Ph.D. students at the University of Minnesota.

References

[1] Z. Guo, W. Feng, S. Liu, W. Jiang, Y. Xu, and Z.-L. Zhang, "RetroFlow: Maintaining control resiliency and flow programmability for software-defined WANs," in *Proc. IEEE/ACM IWQoS*, Jun. 2019, pp. 1–10.

[2] (Nov. 2021). *Software-Defined Infrastructure at Uber*. [Online]. Available: https://www.linuxfoundation.org/blog/software-defined-infrastructure-at-uber/

[3] S. Jain *et al.*, "B4: Experience with a globally-deployed software defined WAN," in *Proc. ACM SIGCOMM*, Aug. 2013, pp. 3–14.

[4] (Nov. 2021). *First in the U.S. to Mobile 5G—What's Next? Defining AT&T's Network Path in 2019 and Beyond*. [Online]. Available: https://about.att.com/story/2019/2019_and_beyond.html

[5] T. Hu, Z. Guo, P. Yi, T. Baker, and J. Lan, "Multi-controller based software-defined networking: A survey," *IEEE Access*, vol. 6, pp. 15980–15996, 2018.

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

[7] D. Hock, M. Hartmann, S. Gebert, M. Jarschel, T. Zinner, and P. Tran-Gia, "Pareto-optimal resilient controller placement in SDN-based core networks," in *Proc. 25th Int. Teletraffic Congr. (ITC)*, Sep. 2013, pp. 1–9.

[8] M. Tanha, D. Sajjadi, and J. Pan, "Enduring node failures through resilient controller placement for software defined networks," in *Proc. IEEE Global Commun. Conf. (GLOBECOM)*, Dec. 2016, pp. 1–7.

[9] B. P. R. Killi and S. V. Rao, "Capacitated next controller placement in software defined networks," *IEEE Trans. Netw. Service Manage.*, vol. 14, no. 3, pp. 514–527, Sep. 2017.

[10] T. Hu, Z. Guo, J. Zhang, and J. Lan, "Adaptive slave controller assignment for fault-tolerant control plane in software-defined networking," in *Proc. IEEE Int. Conf. Commun. (ICC)*, May 2018, pp. 1–6.

[11] G. Yao, J. Bi, and L. Guo, "On the cascading failures of multi-controllers in software defined networks," in *Proc. 21st IEEE Int. Conf. Netw. Protocols (ICNP)*, Oct. 2013, pp. 1–2.

[12] T. Hu, P. Yi, Z. Guo, J. Lan, and Y. Hu, "Dynamic slave controller assignment for enhancing control plane robustness in software-defined networks," *Future Gener. Comput. Syst.*, vol. 95, pp. 681–693, Jun. 2019.

[13] (Nov. 2021). *Brocade MLX-8 Pe*. [Online]. Available: https://www.dataswitchworks.com/datasheets/MLX_Series_DS.pdf

[14] N. L. Van Adrichem, C. Doerr, and F. A. Kuipers, "OpenNetMon: Network monitoring in OpenFlow software-defined networks," in *Proc. IEEE NOMS*, May 2014, pp. 1–8.

[15] A. Tootoonchian, M. Ghobadi, and Y. Ganjali, "OpenTM: Traffic matrix estimator for OpenFlow networks," in *Proc. PAM*. Berlin, Germany: Springer, 2010, pp. 201–210.

[16] (Nov. 2021). *OpenFlow Switch Specification 1.3.0*. [Online]. Available: https://opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.3.0.pdf

[17] J. Xie, D. Guo, X. Li, Y. Shen, and X. Jiang, "Cutting long-tail latency of routing response in software defined networks," *IEEE J. Sel. Areas Commun.*, vol. 36, no. 3, pp. 384–396, Mar. 2018.

[18] B. Heller, R. Sherwood, and N. McKeown, "The controller placement problem," in *Proc. 1st Workshop Hot Topics Softw. Defined Netw. (HotSDN)*, 2012, pp. 7–12.

[19] G. Yao, J. Bi, Y. Li, and L. Guo, "On the capacitated controller placement problem in software defined networks," *IEEE Commun. Lett.*, vol. 18, no. 8, pp. 1339–1342, Aug. 2014.

[20] S. Agarwal, M. Kodialam, and T. V. Lakshman, "Traffic engineering in software defined networks," in *Proc. IEEE INFOCOM*, Apr. 2013, pp. 2211–2219.

[21] K. Poularakis, G. Iosifidis, G. Smaragdakis, and L. Tassiulas, "One step at a time: Optimizing SDN upgrades in ISP networks," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, May 2017, pp. 1–9.

[22] Z. Guo, W. Chen, Y.-F. Liu, Y. Xu, and Z.-L. Zhang, "Joint switch upgrade and controller deployment in hybrid software-defined networks," *IEEE J. Sel. Areas Commun.*, vol. 37, no. 5, pp. 1012–1028, May 2019.

[23] T. Wang, F. Liu, J. Guo, and H. Xu, "Dynamic SDN controller assignment in data center networks: Stable matching with transfers," in *Proc. 35th Annu. IEEE Int. Conf. Comput. Commun. (INFOCOM)*, Apr. 2016, pp. 1–9.

[24] H. Xu, Z. Yu, C. Qian, X.-Y. Li, and Z. Liu, "Minimizing flow statistics collection cost of SDN using wildcard requests," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, May 2017, pp. 1–9.

[25] M. R. Gary and D. S. Johnson, *Computers and Intractability*, vol. 174. San Francisco, CA, USA: Freeman, 1979.

[26] S. Knight, H. X. Nguyen, N. Falkner, R. Bowden, and M. Roughan, "The internet topology zoo," *IEEE J. Sel. Areas Commun.*, vol. 29, no. 9, pp. 1765–1775, Oct. 2011.

[27] C. C. Robusto, "The cosine-haversine formula," *Amer. Math. Monthly*, vol. 64, no. 1, pp. 38–40, 1957.

[28] (Nov. 2021). *Speed, Rates, Times, Delays: Data Link Parameters for CSE 461*. [Online]. Available: https://courses.cs.washington.edu/courses/cse461/99wi/issues/definitions.html

[29] C. Sieber, R. Durner, and W. Kellerer, "How fast can you reconfigure your partially deployed SDN network?" in *Proc. IFIP Netw. Conf. (IFIP Networking) Workshops*, Jun. 2017, pp. 1–9.

[30] P. Thorat, R. Challa, S. M. Raza, D. S. Kim, and H. Choo, "Proactive failure recovery scheme for data traffic in software defined networks," in *Proc. IEEE NetSoft Conf. Workshops (NetSoft)*, Jun. 2016, pp. 219–225.

[31] D. M. F. Mattos, O. C. M. B. Duarte, and G. Pujolle, "A resilient distributed controller for software defined networking," in *Proc. IEEE Int. Conf. Commun. (ICC)*, May 2016, pp. 1–6.

[32] M. Shojaee, M. Neves, and I. Haque, "SafeGuard: Congestion and memory-aware failure recovery in SD-WAN," in *Proc. 16th Int. Conf. Netw. Service Manage. (CNSM)*, Nov. 2020, pp. 1–7.

[33] S. Dou, G. Miao, Z. Guo, C. Yao, W. Wu, and Y. Xia, "Matchmaker: Maintaining network programmability for software-defined WANs under multiple controller failures," *Comput. Netw.*, vol. 192, Jun. 2021, Art. no. 108045.

[34] M. Tanha, D. Sajjadi, R. Ruby, and J. Pan, "Capacity-aware and delay-guaranteed resilient controller placement for software-defined WANs," *IEEE Trans. Netw. Service Manage.*, vol. 15, no. 3, pp. 991–1005, Sep. 2018.

[35] F. He, T. Sato, and E. Oki, "Master and slave controller assignment model against multiple failures in software defined network," in *Proc. IEEE Int. Conf. Commun. (ICC)*, May 2019, pp. 1–6.

[36] F. He and E. Oki, "Load balancing model against multiple controller failures in software defined networks," in *Proc. IEEE Int. Conf. Commun. (ICC)*, Jun. 2020, p. 1–6.

[37] F. He and E. Oki, "Master and slave controller assignment with optimal priority policy against multiple failures," *IEEE Trans. Netw. Service Manage.*, early access, Mar. 8, 2021, doi: 10.1109/TNSM.2021.3064646.

[38] N. Perrot and T. Reynaud, "Optimal placement of controllers in a resilient SDN architecture," in *Proc. 12th Int. Conf. Design Reliable Commun. Netw. (DRCN)*, Mar. 2016, pp. 145–151.

[39] A. Alshamrani, S. Guha, S. Pisharody, A. Chowdhary, and D. Huang, "Fault tolerant controller placement in distributed SDN environments," in *Proc. IEEE Int. Conf. Commun. (ICC)*, May 2018, pp. 1–7.

[40] M. J. F. Alenazi and E. K. Çetinkaya, "Resilient placement of SDN controllers exploiting disjoint paths," *Trans. Emerg. Telecommun. Technol.*, vol. 31, no. 2, p. e3725, Feb. 2020.

[41] Z. Yang and K. L. Yeung, "SDN candidate selection in hybrid IP/SDN networks for single link failure protection," *IEEE/ACM Trans. Netw.*, vol. 28, no. 1, pp. 312–321, Feb. 2020.

[42] L. Guillen, H. Takahira, S. Izumi, T. Abe, and T. Suganuma, "On designing a resilient SDN C/M-plane for multi-controller failure in disaster situations," *IEEE Access*, vol. 8, pp. 141719–141732, 2020.

[43] S. Misra, K. Sarkar, and N. Ahmed, "Blockchain-based controller recovery in SDN," in *Proc. IEEE Conf. Comput. Commun. Workshops (INFOCOM WKSHPS)*, Jul. 2020, pp. 1063–1068.

[44] *OpenFlow Specification 1.5*. [Online]. Available: https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf

[45] P. Vizarreta, C. M. Machuca, and W. Kellerer, "Controller placement strategies for a resilient SDN control plane," in *Proc. 8th Int. Workshop Resilient Netw. Design Modeling (RNDM)*, Sep. 2016, pp. 253–259.

[46] F. J. Ros and P. M. Ruiz, "On reliable controller placements in software-defined networks," *Comput. Commun.*, vol. 77, pp. 41–51, Mar. 2016.

[47] Q. Zhong, Y. Wang, W. Li, and X. Qiu, "A min-cover based controller placement approach to build reliable control network in SDN," in *Proc. IEEE/IFIP Netw. Oper. Manage. Symp. (NOMS)*, Apr. 2016, pp. 481–487.

[48] N. Beheshti and Y. Zhang, "Fast failover for control traffic in software-defined networks," in *Proc. IEEE Global Commun. Conf. (GLOBECOM)*, Dec. 2012, pp. 2665–2670.

[49] Y. Zhang, N. Beheshti, and M. Tatipamula, "On resilience of split-architecture networks," in *Proc. IEEE Global Telecommun. Conf. (GLOBECOM)*, Dec. 2011, pp. 1–6.

[50] S. Yang, L. Cui, Z. Chen, and W. Xiao, "An efficient approach to robust SDN controller placement for security," *IEEE Trans. Netw. Service Manage.*, vol. 17, no. 3, pp. 1669–1682, Sep. 2020.

[51] T. Das and M. Gurusamy, "Controller placement for resilient network state synchronization in multi-controller SDN," *IEEE Commun. Lett.*, vol. 24, no. 6, pp. 1299–1303, Jun. 2020.

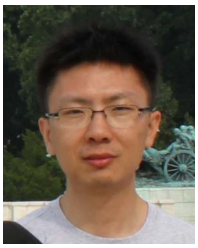
Zehua Guo (Senior Member, IEEE) received the B.S. degree from Northwestern Polytechnical University, Xi'an, China, the M.S. degree from Xidian University, Xi'an, China, and the Ph.D. degree from Northwestern Polytechnical University. He was a Research Fellow with the Department of Electrical and Computer Engineering, New York University Tandon School of Engineering, New York, NY, USA; and a Research Associate with the Department of Computer Science and Engineering, University of Minnesota Twin Cities, Minneapolis, MN, USA. His research interests include programmable networks (such as software-defined networking and network function virtualization), machine learning, and network security. He is a member of ACM. He is currently an Associate Editor of IEEE SYSTEMS JOURNAL 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 *Journal of Parallel and Distributed Computing*. He was the Session Chair of the IEEE ICC 2018 and the IEEE/ACM IWQoS 2021, and serves as the TPC for *Computer Communications* and prestigious conferences, such as AAAI, IWQoS, ICC, ICCCN, and ICA3PP.

Songshi Dou (Graduate Student Member, IEEE) received the B.S. degree from North China Electric Power University, Beijing, China, in 2019. He is currently pursuing the M.S. degree with the Beijing Institute of Technology, Beijing. His research interests cover software-defined networking, network function virtualization, and data center networks. He is a Graduate Student Member of ACM.

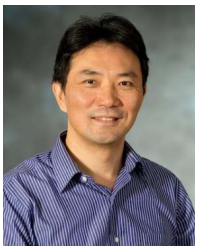
Sen Liu (Member, IEEE) received the B.S. degree from Northeastern University, the M.S. degree from the South China University of Technology, and the Ph.D. degree from Central South University. He worked as a Visiting Scholar with the Department of Computer Science and Engineering, University of Minnesota, 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.

Wendi Feng is currently pursuing the Ph.D. degree with the State Key Laboratory of Network and Switching Technology, Beijing University of Posts and Telecommunications, with Prof. Junliang Chen. He was also advised by Prof. Zhi-Li Zhang at the University of Minnesota Twin Cities. His research interests include mobile computing, cloud computing, computer networking, software-defined network, and network function virtualization.

Wenchao Jiang received the B.S. and M.S. degrees from Shanghai Jiao Tong University, China, and the Ph.D. degree from the Department of Computer Science and Engineering, University of Minnesota, Twin Cities. He is currently an Assistant Professor with the Pillar of Information Systems Technology and Design, Singapore University of Technology and Design, Singapore. His research interests include the Internet of Things, wireless networks, sensor networks, and mobile systems.



Yang Xu (Member, IEEE) received the Bachelor of Engineering 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. He has published about 90 journal and conference papers and holds more than ten U.S. and international granted patents on various aspects of networking and computing. His research interests include software-defined networks, data center networks, distributed machine learning, edge computing, network function virtualization, and network security. He served as a TPC member for many international conferences, an Editor for the *Journal of Network and Computer Applications* (Elsevier), and a Guest Editor for the IEEE JOURNAL ON SELECTED AREAS IN COMMUNICATIONS—Special Series on Network Softwarization & Enablers and *Security and Communication Networks* journal—Special Issue on Network Security and Management in SDN (Wiley).



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 a 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 is a member of ACM. He was a co-recipient of several best paper awards and has received several other awards. He has co-chaired several conferences/workshops (as the General Chair or the TCP Chair), including the IEEE INFOCOM, IEEE ICNP, ACM CONEXT, and IFIP Networking, and has served on the TPC for numerous conferences/workshops.