Hindawi
Security and Communication Networks
Volume 2018, Article ID 6361901, 12 pages
https://doi.org/10.1155/2018/6361901

Research Article

Kuijia: Traffic Rescaling in Software-Defined Data Center WANs

Che Zhang, Hong Xu, Libin Liu, Zhixiong Niu, and Peng Wang

NetX Lab, City University of Hong Kong, Kowloon Tong, Hong Kong

Correspondence should be addressed to Che Zhang; czhang226-c@my.cityu.edu.hk

Received 27 September 2017; Accepted 6 November 2017; Published 15 January 2018

Academic Editor: Yang Xu

Network faults like link or switch failures can cause heavy congestion and packet loss. Traffic engineering systems need a lot of time to detect and react to such faults, which results in significant recovery times. Recent work either preinstalls a lot of backup paths in the switches to ensure fast rerouting or proactively prereserves bandwidth to achieve fault resiliency. Our idea agilely reacts to failures in the data plane while eliminating the preinstallation of backup paths. We propose Kuijia, a robust traffic engineering system for data center WANs, which relies on a novel failover mechanism in the data plane called rate rescaling. The victim flows on failed tunnels are rescaled to the remaining tunnels and enter lower priority queues to avoid performance impairment of aboriginal flows. Real system experiments show that Kuijia is effective in handling network faults and significantly outperforms the conventional rescaling method.

1. Introduction

Traffic engineering (TE) is increasingly implemented using software-defined networking (SDN), especially in inter-data center WANs. Examples include Google's B4 and Microsoft's SWAN [1, 2]. Usually, some tunnel protocol is used: the controller establishes multiple tunnels (i.e., network paths) between an ingress-egress switch pair and configures splitting weights at the ingress switch. The ingress switch then uses hashing based multipath forwarding such as ECMP to send flows.

An important issue about TE that is commonly overlooked in the literature is robustness against failures. In reality, failures are the norm rather than exception, especially for large networks. Table 1 shows failure statistics data from Microsoft's data center WAN [3]. The probability of having at least one link failure within five minutes, which corresponds to the TE frequency [1, 2], is more than 20%. Even with a single link failure, the impact can be severe as a data center WAN operates near capacity for maximum efficiency [1, 2].

Controller intervention offers the best failure recovery performance given its global network view. However, recomputing a new TE plan and updating the forwarding rules across the entire network take at least minutes and are error-prone [4–6]. When the controller is being attacked by Distributed Denial of Service, or others, the reaction time of the controller can be even longer [7–9]. Therefore, we need to have a mechanism to protect the data plane from congestion after failures without the intervention of a controller. For responsiveness, a simple data plane reactive method called *rescaling* is deployed in practice. Upon detecting the failure, the ingress switch normalizes splitting weights to redirect traffic among the remaining tunnels [6]. Rescaling quickly restores connectivity without involving the controller at all. However, since traffic is still sending at the original rates, local rescaling more than often leaves the network in a congested state [6].

Some solutions have emerged to solve this practically important issue. Suchara et al. [10] propose to precompute the splitting weights for arbitrary faults to reduce transient congestion. This approach may not work well for large production networks due to the exponentially many failure cases. Liu et al. [6] propose forward fault correction (FFC). FFC proactively considers failures when formulating the TE problem. As a result, the TE solution can guarantee no congestion happens for arbitrary k faults with rescaling. Intuitively, such strong guarantees come with a price: in FFC, a portion (about 5%–10% depending on k) of the network capacity has to be always left vacant in order to handle traffic from rescaling. This means hundreds of Gbps bandwidth is

Table 1: Link failure frequencies in Microsoft data center WAN [3].

Number of link failures	Time intervals		
	2 min	5 min	10 min
1	10.6%	21.5%	31.2%
2	0.14%	1.1%	4.2%
3	0.14%	0.7%	1.4%

wasted most of the time. Arguably, the cost outweighs the benefits of eliminating transient congestion.

Thus, the following question remains largely open: can we design a robust TE system that is (1) responsive in quickly restoring connectivity, (2) effective in reducing congestion without excessive bandwidth overhead, and (3) practical and simple enough to be deployed in existing switches?

Our main contribution is the design and evaluation of Kuijia (the word "Kuijia" means armor in Chinese; Kui is for protecting the head and neck, and Jia is for protecting the torso), a robust TE system for data center WANs that answers the above question affirmatively. We argue to isolate the affected flows from the aboriginal ones to avoid the propagation of failure impact. This is particularly useful when there are many latency-sensitive flows like video, online shopping, and search whose traffic is rapidly growing due to the development of mobile devices and high-speed cellular networking.

Kuijia relies on a novel failover mechanism in the data plane called *rate rescaling* that rescales the traffic sending rates in addition to splitting weights, by using priority queueing at switches. The victim flows are still rescaled to the remaining tunnels, but they now enter a lower priority queue at the switches and do not compete with aboriginal flows on the remaining tunnels. Effectively, their sending rates are automatically throttled to only using the available bandwidth of the remaining tunnels without the need for controller intervention.

Kuijia with rate rescaling offers an advantage over simple rescaling. Rescaling only ensures the failed link is avoided. Yet, flows are still sending at their original rates to the remaining tunnels. Clearly, with the loss of capacity, many packets will be dropped after rescaling, and every TCP flow on the remaining tunnels will back off and suffer from throughput loss. Rate rescaling ensures there is no congestion even with the victim flows, and the aboriginal flows are not affected. It maintains the responsiveness of rescaling, is simple to implement as priority queueing, is widely supported by commercial switches, and is effective in utilizing the available bandwidth due to the work-conserving nature.

This paper is an extended version of work published in [11]. We extend our previous work to handle traffic with multiple priorities. Specifically, we propose a new flow table decomposition method to produce multiple tables in order to reduce the number of flow entries. For experiment, we add a large-scale simulation to demonstrate our design of using original weights for rate rescaling is more simple and effective compared with storing and using the precomputed weights. In addition, we add testbed experiments to evaluate performance and flow entries' memory usage of Kuijia for multipriority traffic. We also add a new section to discuss several issues related to the use of Kuijia in a production data center WAN and we explain them in three aspects, which include traffic characteristics, traffic priorities, and impact of flow size to hashing.

2. Related Work

Failures in SDN. There is much work to deal with failures in SDN. New abstractions are proposed in [12, 13] to enable developers to write fault-tolerant SDN applications. Some other work relies on the local fast failover mechanism introduced in OpenFlow to design new functions. Schiff et al. [14] propose SmartSouth to provide a new data plane for OpenFlow switches that can implement fault-tolerant mechanisms. Borokhovich et al. [15] develop algorithms to compute failover tables. Chang et al. [16] develop an optimization-theoretic framework to validate network designs under uncertain demands and failures. Kuijia is different in that it focuses on remedying the congestion due to rescaling.

Failures in Data Center WANs. The most widely used approach to deal with network failures, including link or switch failures, is to recompute a new TE solution based on the changed topology and reprogram the switches [1, 2]. However, such a reactive approach is not fast and efficient enough as discussed in Section 1.

Several proactive approaches have been proposed to solve this important problem. Suchara et al. [10] modify the ingress switches' rescaling behavior. Rather than simple proportional rescaling, tunnel splitting weights are based on the set of residual tunnels after the failure. These weights are precomputed and preconfigured at switches. Although it achieves near-optimal load balancing, this approach can handle only a limited number of potential failure cases as there are exponentially many of them to consider, and switches have limited space for flow rules.

Liu et al. [6] propose forward fault correction (FFC) to handle failures proactively. FFC ensures that each time the operator computes a new TE, it is congestion-free not only without any failures, but also with any link failures that could happen in the following TE interval. This is in sharp contrast to recomputing TE after failures as it requires no update to TE in response to failures. Although FFC spreads network traffic such that congestion-free property is guaranteed under arbitrary combinations of up to k failures, the price is very high. About 5%–10% of the network capacity depending on k has to be always left vacant to handle traffic from rescaling. SWAN [1] develops a new technique that also leverages a small amount of scratch capacity on links to apply updates in a provably congestion-free manner.

Instead of waiting for rescaling in the ingress switches, Zheng et al. [17] use backup tunnels that start from the failing switch and end at the egress switches to redirect the affected traffic. It can be faster and more effective in reducing congestion but the cost is still high as there are exponentially many failure cases to consider. The large number of flow entries required for backup tunnels is too expensive for the limited hardware tables [18].

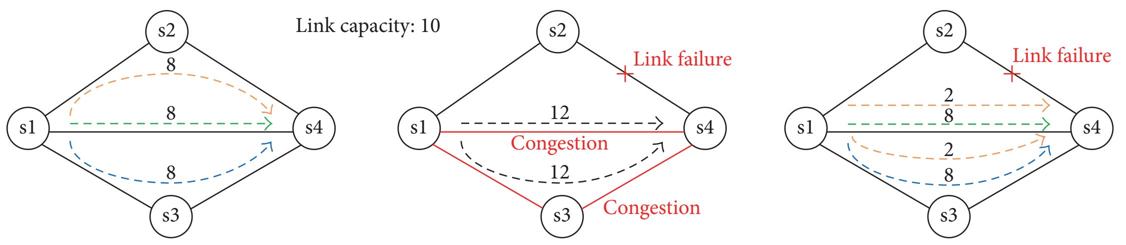

Link capacity: 10
s1
s2
s3
s4
8
8
8
(a) Initial traffic distribution
Link failure
12
Congestion
12
Congestion
(b) Traffic distribution after rescaling
Link failure
2
8
2
8
(c) Traffic distribution after rate rescaling

FIGURE 1: Comparison of rescaling and rate rescaling in handling a single link failure.

Our method, Kuijia, is different from the existing work as we use priority queueing in switches, which is simple to implement in practice and has no overhead of excessive bandwidth or a large number of flow entries for backup tunnels.

3. Motivation

We motivate our idea using a simple example. Figure 1(a) shows a small network with traffic sent from s1 to s4. The traffic is routed over three tunnels: s1 → s2 → s4 (T1), s1 → s4 (T2), and s1 → s3 → s4 (T3). Each tunnel is configured with the same weight and carries 8 Gbps traffic. When link s2–s4 in T1 fails, s1 rescales the traffic to the remaining two tunnels, resulting in traffic distribution as shown in Figure 1(b). Since the traffic is still sent at 24 Gbps, the remaining tunnels T2 and T3 need to carry 12 Gbps each and are heavily congested.

The difference between Kuijia and conventional rescaling is that Kuijia differentiates aboriginal traffic on remaining tunnels from victim traffic rescaled to them. Kuijia places the victim traffic into a low priority queue of the remaining tunnels, while the aboriginal traffic enters a higher priority queue. With Kuijia, traffic is distributed as shown in Figure 1(c). The victim traffic (shown in yellow) uses the remaining capacity of T2 and T3 and sends at 2 Gbps in each tunnel. This does not cause any congestion or packet loss for the aboriginal flows and fully utilizes the link capacity.

We experimentally verify the effectiveness of Kuijia using a testbed on Emulab [19]. We connect 4 Emulab servers running OpenvSwitch (OVS) [20] to form the same topology as in Figure 1. A dedicated server runs the controller to manage the network. In Kuijia, 3 strict priority queues are configured on the egress ports of each switch. Control messages enter the highest priority queue with priority 0. Normal application traffic has priority 1 but is demoted to priority 2 once it is rescaled to other tunnels after failures. For simplicity, both rescaling and Kuijia are implemented in the control plane: a switch informs the controller of a link failure. The controller then adjusts the flow splitting weights and priority numbers at the corresponding ingress switches of the victim flows.

Switch s1 starts `iperf` TCP connections to s4 over three tunnels. Since in our example rescaling splits the victim flow on T1 to T2 and T3, we configure s1 to send two `iperf` TCP flows f1 and f2 over T1. Flow f1 is rescaled to T2 and f2 to T3. s1 sends another two flows f3 and f4 over T2 and T3, respectively.

TABLE 2: Testbed experiment for the motivation example, where the remaining tunnels have a vacant capacity for victim traffic.

Flows	f1	f2	f3	f4
Rescaling				
Before failure	380 Mbps	381 Mbps	762 Mbps	762 Mbps
After failure	379 Mbps	378 Mbps	584 Mbps	586 Mbps
Kuijia				
Before failure	380 Mbps	381 Mbps	762 Mbps	762 Mbps
After failure	177 Mbps	177 Mbps	762 Mbps	762 Mbps

TABLE 3: Testbed experiment for the motivation example, where the remaining tunnels do not have a vacant capacity.

Flows	f1	f2	f3	f4
Rescaling				
Before failure	475 Mbps	468 Mbps	943 Mbps	941 Mbps
After failure	472 Mbps	474 Mbps	465 Mbps	470 Mbps
Kuijia				
Before failure	475 Mbps	468 Mbps	943 Mbps	941 Mbps
After failure	0.074 Mbps	0.011 Mbps	943 Mbps	941 Mbps

We run two experiments with different extents of congestion to demonstrate the effectiveness of Kuijia. Table 2 shows the result when flows f3 and f4 send at 800 Mbps and f1 and f2 send at 400 Mbps each before failure. This represents the case when the remaining tunnels (T2 and T3) have vacant capacity. We observe that, with simple rescaling, the throughput of all flows degrades after failures, since the aggregate demand of victim and aboriginal flows (1.2 Gbps) exceeds 1 Gbps. Now with Kuijia, aboriginal flows f3 and f4 are not affected at all as shown in Table 2, and the victim flows use the vacant capacity of 200 Mbps without causing any congestion or packet loss.

Table 3 shows the result when f3 and f4 send at 1 Gbps and f1 and f2 send at 500 Mbps each before failure. This represents the case when the remaining tunnels do not have any capacity for the victim traffic. Rescaling again causes severe congestion to aboriginal traffic on the remaining tunnels, and after TCP convergence f3 and f4 achieve throughput of ~470 Mbps. With Kuijia, the victim traffic (f1 and f2) does not obtain any throughput and the aboriginal flows are not impacted at all.

4. Design

In this section, we first introduce the background of TE and rescaling implementation in production data center WANs, and then we explain the design of Kuijia and its difference from rescaling.

4.1. Background. In a data center WAN, after the controller computes the bandwidth allocation and weights for all the tunnels of each ingress-egress switch pair, it issues the group table entries and flow table entries in OpenFlow [1, 21]. Label-based forwarding is usually used to reduce forwarding complexity [2]. The ingress switch uses group entry in the group table to split traffic across multiple tunnels and assigns a label to traffic of a specific tunnel. The downstream switches simply read the label and forward packets based on the flow entries for that label from the flow table. As an example, Figure 2 shows the group tables and flow tables of four switches for the network used in Figure 1. The forwarding label can be MPLS, VLAN tags, and so forth.

Flows are hashed to different tunnels consistently (and different labels are applied) when they arrive at the ingress switch for simplicity. Thus, splitting weights are configured as ranges of the hashed values. For example, in Figure 3(a), the weights are 0.5, 0.3, and 0.2 for tunnels T1, T2, and T3, respectively. For simple rescaling, its implementation is as follows. Suppose tunnel T1 fails as in the motivation example. The ingress switch rescales the traffic to the remaining tunnels by removing the bucket in the group entry that corresponds to the failed tunnel as shown in Figure 2 (entries with $*$ only exist in Kuijia, not in rescaling). The entries in the blue table are issued after failures. In addition, since T1 fails, the hash value ranges for T2 and T3 also "rescale" accordingly, so that weights of T2 and T3 are now 0.6 and 0.4. As discussed already, this may cause congestion after rerouting the victim traffic [6].

4.2. Kuijia. Here, we explain the detailed design of Kuijia for SDN based data center WANs. We focus on dealing with single link failures, which are most common in production networks as shown in Table 1. Multiple link failures are rare and can be handled by controller intervention on a need basis.

We propose Kuijia with *rate rescaling* to reduce the impact of congestion after failures. Its design is simple and can be implemented in OpenFlow switches. Suppose there are k tunnels for traffic between a given ingress-egress switch pair, and one tunnel fails. Kuijia keeps the original hash range and separates the hash range of the failed tunnel into $k-1$ parts according to weights of the $k-1$ tunnels to form the new hash ranges. It also marks the hash range of the failed tunnel to low priority in order to enforce priority queueing. This way, Kuijia can differentiate the aboriginal traffic on the remaining $k-1$ tunnels from the victim traffic that is rescaled to them. For the same example in Figure 3(b), when T1 fails, its hash range is split into two parts for T2 and T3 with weights to 0.3 and 0.2, respectively. One can easily verify that the aboriginal flows on T2 and T3 are still hashed to the same ranges and routed normally. Victim traffic on T1 is now rescaled to T2 and T3 and tagged as low priority in order not to affect the aboriginal flows.

In order to verify that it is effective to use the original weights and just separate the hash range of the failed tunnels, we compare it to rerouting using precomputed weights. The precomputation works as follows. For a given link failure, we keep the weights of unaffected tunnels and sending rates of unaffected flows unchanged. We take the remaining bandwidth of each link and tunnel, as well as the victim flows that need to be rerouted, as the input of a new TE program and compute the best weights for these victim flows. Clearly, recomputing the weights yields the optimal performance to deal with the failure. We run experiments with 10 random graphs each having 100 nodes and 200 links. For each graph, we randomly select 40 switch pairs for 10 runs, and each pair has 3 tunnels. As [10] shows, even in a large ISP backbone, three or four tunnels per switch pair is sufficient. In each run, we vary the demand of each switch pair from 0.8 Gbps to 3.0 Gbps. We sequentially fail all the edges one by one and then compute the average throughput loss for each demand.

Figure 4 shows the comparison result. We observe that the performance of using original weights is highly comparable to that of precomputing new weights. As each switch pair has 3 tunnels and they may have some common links, with one link down, there is only one or two remaining tunnels for each affected switch pair. When the demand is small, which means the network is not that congested before failure, simply using original weights can meet the demand in most cases. When the demand becomes larger, which means the network is more congested, using TE to precompute weights cannot reduce throughput loss much further. Reference [10] also shows similar results. The results demonstrate that our design of using original weights and separating the hash range is simple and effective and also avoids the complexity of storing the precomputed weights.

Note that when one link fails, any intermediate switch may potentially become congested due to rescaling. Thus, it is necessary for *all* switches to perform priority queueing for the victim flows, not just the corresponding ingress switch. To do that, there are two ways. The first one is to compute which links will be congested after rescaling, and then we only need to configure the corresponding flow entries at those switches to realize priority queueing. Although this method uses fewer flow entries, it is hard to achieve in reality because the ingress switch has no information of all the traffic in the network, and the controller has to compute which links will be congested after failures actually happen, which defeats the purpose of having a data plane failover mechanism.

Thus, Kuijia uses a simple method that doubles the flow entries in all switches for each tunnel. We have a normal priority queue and a low priority queue at each port of each switch. Each queue has the same set of flow entries to route traffic. Traffic with low priority tags is sent to the low priority queue as shown in Figure 5. This is simple to implement in the data plane and can handle any link failures quickly.

For example, in Figure 2, for aboriginal flows sending to 10.0.2.0/24, they match `low` = 0, `inport` = 1, `pathid` = 3 in s3 and go to `queue(1)`. The corresponding entry, matching `low` = 1, `inport` = 1, `pathid` = 3 will go to `queue(2)` which is the low priority queue and is used when there are victim flows due to link failure, for example, when the s2–s4 link is down. In the ingress switch s1, the group table applies low priority tags to the victim flows (entries with $*$) and directs the packets to the `outport` which is connected to the next-hop switch.

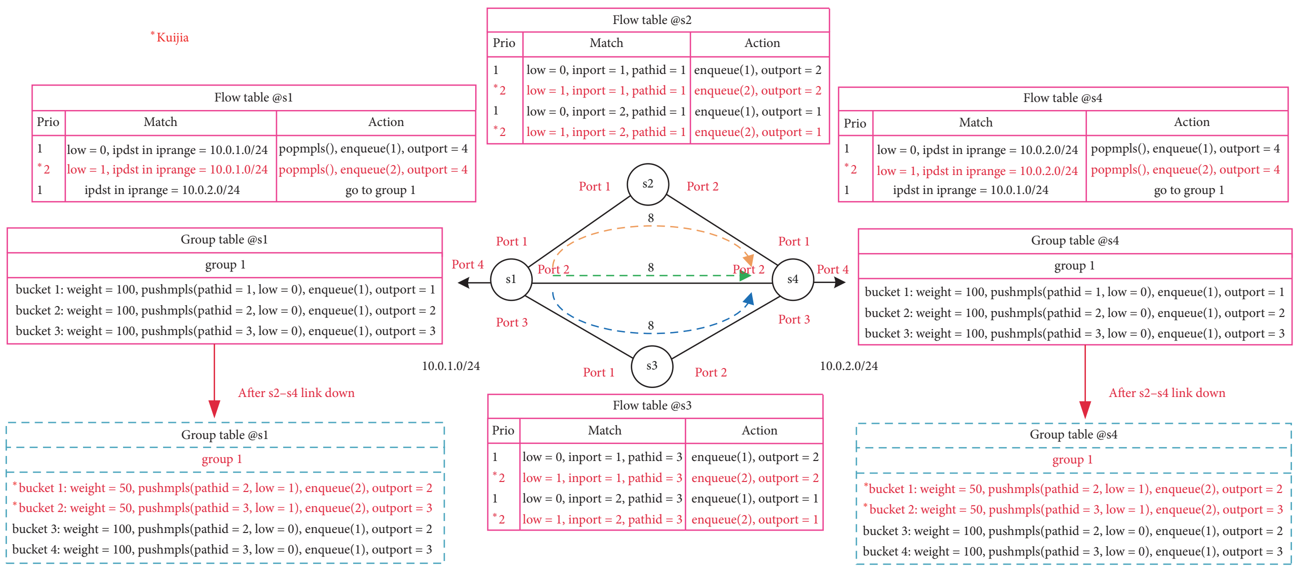

*Kuijia
Flow table @s1
Prio | Match | Action
1 | low = 0, ipdst in iprange = 10.0.1.0/24 | popmpls(), enqueue(1), outport = 4
*2 | low = 1, ipdst in iprange = 10.0.1.0/24 | popmpls(), enqueue(2), outport = 4
1 | ipdst in iprange = 10.0.2.0/24 | go to group 1
Group table @s1
group 1
bucket 1: weight = 100, pushmpls(pathid = 1, low = 0), enqueue(1), outport = 1
bucket 2: weight = 100, pushmpls(pathid = 2, low = 0), enqueue(1), outport = 2
bucket 3: weight = 100, pushmpls(pathid = 3, low = 0), enqueue(1), outport = 3
After s2–s4 link down
Group table @s1
group 1
*bucket 1: weight = 50, pushmpls(pathid = 2, low = 1), enqueue(2), outport = 2
*bucket 2: weight = 50, pushmpls(pathid = 3, low = 1), enqueue(2), outport = 3
bucket 3: weight = 100, pushmpls(pathid = 2, low = 0), enqueue(1), outport = 2
bucket 4: weight = 100, pushmpls(pathid = 3, low = 0), enqueue(1), outport = 3
Flow table @s2
Prio | Match | Action
1 | low = 0, inport = 1, pathid = 1 | enqueue(1), outport = 2
*2 | low = 1, inport = 1, pathid = 1 | enqueue(2), outport = 2
1 | low = 0, inport = 2, pathid = 1 | enqueue(1), outport = 1
*2 | low = 1, inport = 2, pathid = 1 | enqueue(2), outport = 1
Port 1
s2
Port 2
8
Port 1
Port 4
s1
Port 2
8
Port 2
Port 1
s4
Port 4
Port 3
8
Port 3
10.0.1.0/24
Port 1
s3
Port 2
10.0.2.0/24
Flow table @s3
Prio | Match | Action
1 | low = 0, inport = 1, pathid = 3 | enqueue(1), outport = 2
*2 | low = 1, inport = 1, pathid = 3 | enqueue(2), outport = 2
1 | low = 0, inport = 2, pathid = 3 | enqueue(1), outport = 1
*2 | low = 1, inport = 2, pathid = 3 | enqueue(2), outport = 1
Flow table @s4
Prio | Match | Action
1 | low = 0, ipdst in iprange = 10.0.2.0/24 | popmpls(), enqueue(1), outport = 4
*2 | low = 1, ipdst in iprange = 10.0.2.0/24 | popmpls(), enqueue(2), outport = 4
1 | ipdst in iprange = 10.0.1.0/24 | go to group 1
Group table @s4
group 1
bucket 1: weight = 100, pushmpls(pathid = 1, low = 0), enqueue(1), outport = 1
bucket 2: weight = 100, pushmpls(pathid = 2, low = 0), enqueue(1), outport = 2
bucket 3: weight = 100, pushmpls(pathid = 3, low = 0), enqueue(1), outport = 3
After s2–s4 link down
Group table @s4
group 1
*bucket 1: weight = 50, pushmpls(pathid = 2, low = 1), enqueue(2), outport = 2
*bucket 2: weight = 50, pushmpls(pathid = 3, low = 1), enqueue(2), outport = 3
bucket 3: weight = 100, pushmpls(pathid = 2, low = 0), enqueue(1), outport = 2
bucket 4: weight = 100, pushmpls(pathid = 3, low = 0), enqueue(1), outport = 3

FIGURE 2: The design of flow table and group table of each switch in the simple topology.

```
01234567012345670123456701234567    01234567012345670123456701234567
+----------+----------+----------+----------+    +----------+----------+----------+----------+
|         T1          |    T2    |    T3    |    |         T1          |    T2    |    T3    |
+----------+----------+----------+----------+    +----------+----------+----------+----------+
|              T2                |    T3    |    |  T2 low  | T3 low   |    T2    |    T3    |
+----------+----------+----------+----------+    +----------+----------+----------+----------+
                  (a)                                              (b)
```

FIGURE 3: The change of hash range after failure.

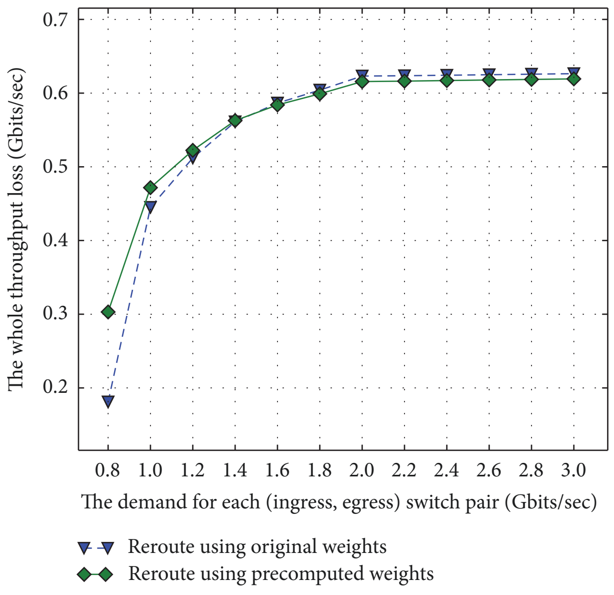


FIGURE 4: The whole throughput loss after single link failure as demand increases.

Each intermediate switch of the tunnel matches packets on priority, inport, and pathid. Victim flows are then routed to queue(2) (rerouting flow, low priority tag = 1) of outport while aboriginal flows are routed to queue(1).

Note that as TCP connection needs two-way communication, the flow entries and group entries are also issued for two-way communication. For example, if s1 sends TCP packets to s4 through s1-s3-s4, we need to issue the flow entries not only for the direction of s1 $\rightarrow$ s3 $\rightarrow$ s4, but also for the reverse direction of s4 $\rightarrow$ s3 $\rightarrow$ s1 (e.g., s3 has the flow entry: match{low = 0, inport = 2, pathid = 3}, actions{enqueue(1), outport = 1}). Hence, the TCP ACK packets could be returned to s1 by matching the flow entry of switches in the reverse direction. We use MPLS `label` field to store our path ID (each tunnel (path) has a unique path ID) and `tc` field to store our low priority tag (0 means aboriginal flow and 1 means victim flow).

4.3. Multiple Priorities. Kuijia can also be extended to handle traffic with multiple priorities. We have assumed that all traffic has the same priority before failures thus far. In practice, it is common for networks to use multiple priorities to differentiate applications with distinct performance requirements [22, 23]. Kuijia can be extended to this setting so that high priority victim flows could also obtain as much bandwidth as possible after rerouting.

To illustrate Kuijia's working for multipriority traffic, consider a simple case where switches in the network have

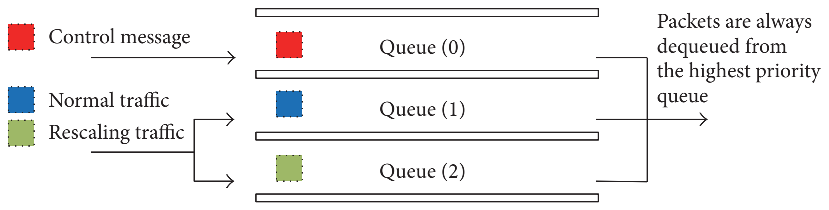
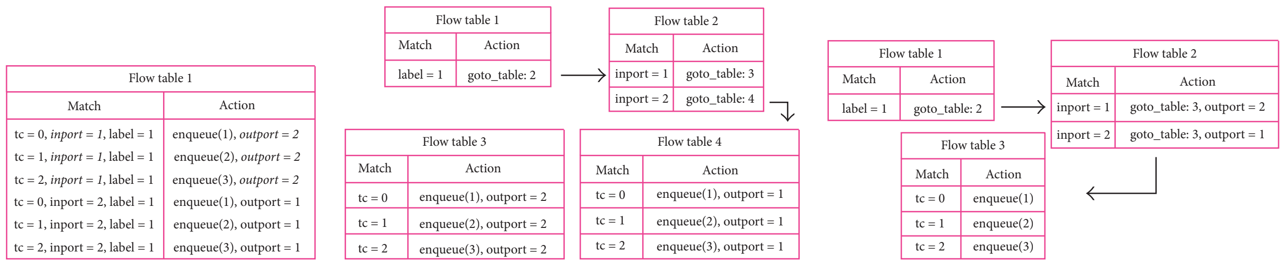


FIGURE 5: The switch queues in Kuijia.



(a)

Flow table 1	
Match	Action
tc = 0, *inport = 1*, label = 1	enqueue(1), *outport = 2*
tc = 1, *inport = 1*, label = 1	enqueue(2), *outport = 2*
tc = 2, *inport = 1*, label = 1	enqueue(3), *outport = 2*
tc = 0, inport = 2, label = 1	enqueue(1), outport = 1
tc = 1, inport = 2, label = 1	enqueue(2), outport = 1
tc = 2, inport = 2, label = 1	enqueue(3), outport = 1

(b)

Flow table 1	
Match	Action
label = 1	goto_table: 2

Flow table 2	
Match	Action
inport = 1	goto_table: 3
inport = 2	goto_table: 4

Flow table 3	
Match	Action
tc = 0	enqueue(1), outport = 2
tc = 1	enqueue(2), outport = 2
tc = 2	enqueue(3), outport = 2

Flow table 4	
Match	Action
tc = 0	enqueue(1), outport = 1
tc = 1	enqueue(2), outport = 1
tc = 2	enqueue(3), outport = 1

(c)

Flow table 1	
Match	Action
label = 1	goto_table: 2

Flow table 2	
Match	Action
inport = 1	goto_table: 3, outport = 2
inport = 2	goto_table: 3, outport = 1

Flow table 3	
Match	Action
tc = 0	enqueue(1)
tc = 1	enqueue(2)
tc = 2	enqueue(3)

FIGURE 6: The example of three kinds of design for flow entries under multiple priorities.

8 priority queues each. The highest priority 0 is for control traffic. We set priorities 1 to 4 for high priority traffic, while we set priority 6 for background low priority traffic before failures. After a link failure, the high priority victim flows are rerouted by Kuijia to the remaining tunnels through priority 5 to guarantee they can get bandwidth by occupying the bandwidth of low priority traffic. Similarly, the low priority victim flows are rerouted to the remaining tunnels through priority 7 which is the lowest priority to guarantee that the aboriginal flows are not affected.

Note two design particulars here. First, Kuijia ensures that the lowest priority traffic is always dropped first when there is not enough bandwidth. Second, the bandwidth of each high priority queue can be changed by occupying the bandwidth of lower priority.

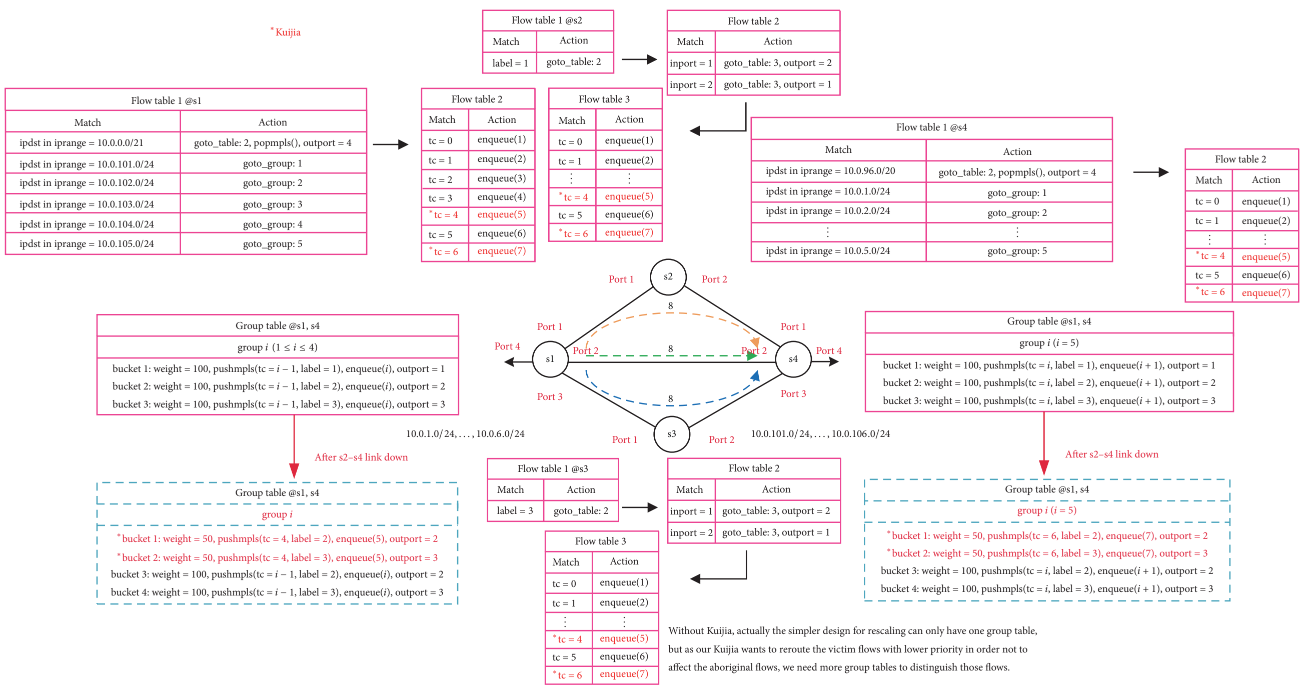
Therefore, we set the maximum bandwidth for the priority `queue`(0) to `queue`(6) to the capacity of each link and the minimum bandwidth for the lowest priority `queue`(7) to `queue`(0).

One challenge with multiple priorities is how to handle the increased number of flow entries required to implement Kuijia. Using a simple example where s1 sends traffic with 3 priorities to s2 from `inport` 1 to `outport` 2, we can show how to reduce the memory usage of flow entries of s1 step by step. Our previous design [11], as shown in Figure 6(a), implements rate rescaling by replicating the flow entries with some parameter changes. This results in high memory usage as each flow entry includes duplicated information, such as `label` = 1 in each match part. We can reduce such duplication by using a greedy heuristic proposed in [22]. This still results in duplicated information between flow table 3 and flow table 4 as in Figure 6(b). Notice that these two tables are the same except the `outport`. We now propose a new method that can further reduce the duplicated information by the following analysis.

Notice that sometimes the values in `match` and `action` have correspondence between each. For instance, in Figure 6(a), ignoring others, for the same `inports` in the `match` field, the `outports` are also the same in the `action` field (`inport` = 1 corresponding to `outport` = 2 and vice versa). Therefore, we can further reduce the number of tables by recording such correspondence and checking it in each decomposition. In this example, finally we can get Figure 6(c) which saves one table compared to Figure 6(b) by adding the corresponding `outport` in the `action` of flow table 2.

This new flow table decomposition method is useful for Kuijia with multiple priorities. Traffic with different priorities goes through the same tunnels for each switch pair. If we built the flow tables based on simple replication as shown in Figure 6(a), most flow entries are almost the same except that different priority traffic needs to be tagged differently and matched to different priority queues. This yields a lot of opportunities for our decomposition method to exploit, and we can reduce the memory utilization of flow and group tables. For example, in Figure 7, the first flow entry in s1 is for traffic from s4 to s1. As there is only one outport for s1 in the simple topology, we only need to match the dst ip 10.0.0.0/21 of each packet in flow table 1 and then go to flow table 2 to check the `tc` (traffic class) filed in MPLS to decide which queue the packet should go to. Next it goes back to flow table 1 to execute next action `popmpls()` and output the packet from `outport` 4. The detailed explanation of the packet processing pipeline for multiple tables can be found in page 19 of [24].

Figure 8 shows the memory usage comparison between replicating flow entries and our new flow table decomposition method for the example shown in Figure 7. For simplicity, we treat the memory usage of each `match` (e.g., ipdst in ip range = 10.0.0.0/21) and `action` (e.g., `enqueue`(1)) the same—32 bits. Using our new method, the slope of the curve for multiple



Flow table 1 @s1	
Match	Action
ipdst in iprange = 10.0.0.0/21	goto_table: 2, popmpls(), outport = 4
ipdst in iprange = 10.0.101.0/24	goto_group: 1
ipdst in iprange = 10.0.102.0/24	goto_group: 2
ipdst in iprange = 10.0.103.0/24	goto_group: 3
ipdst in iprange = 10.0.104.0/24	goto_group: 4
ipdst in iprange = 10.0.105.0/24	goto_group: 5

Flow table 2	
Match	Action
tc = 0	enqueue(1)
tc = 1	enqueue(2)
tc = 2	enqueue(3)
tc = 3	enqueue(4)
*tc = 4	enqueue(5)
tc = 5	enqueue(6)
*tc = 6	enqueue(7)

Flow table 1 @s2	
Match	Action
label = 1	goto_table: 2

Flow table 2	
Match	Action
inport = 1	goto_table: 3, outport = 2
inport = 2	goto_table: 3, outport = 1

Flow table 3	
Match	Action
tc = 0	enqueue(1)
tc = 1	enqueue(2)
⋮	⋮
*tc = 4	enqueue(5)
tc = 5	enqueue(6)
*tc = 6	enqueue(7)

Flow table 1 @s4	
Match	Action
ipdst in iprange = 10.0.96.0/20	goto_table: 2, popmpls(), outport = 4
ipdst in iprange = 10.0.1.0/24	goto_group: 1
ipdst in iprange = 10.0.2.0/24	goto_group: 2
⋮	⋮
ipdst in iprange = 10.0.5.0/24	goto_group: 5

Flow table 2	
Match	Action
tc = 0	enqueue(1)
tc = 1	enqueue(2)
⋮	⋮
*tc = 4	enqueue(5)
tc = 5	enqueue(6)
*tc = 6	enqueue(7)

Group table @s1, s4
group i $(1 \le i \le 4)$
bucket 1: weight = 100, pushmpls(tc = $i - 1$, label = 1), enqueue(i), outport = 1
bucket 2: weight = 100, pushmpls(tc = $i - 1$, label = 2), enqueue(i), outport = 2
bucket 3: weight = 100, pushmpls(tc = $i - 1$, label = 3), enqueue(i), outport = 3

Group table @s1, s4
group i $(i = 5)$
bucket 1: weight = 100, pushmpls(tc = i, label = 1), enqueue($i + 1$), outport = 1
bucket 2: weight = 100, pushmpls(tc = i, label = 2), enqueue($i + 1$), outport = 2
bucket 3: weight = 100, pushmpls(tc = i, label = 3), enqueue($i + 1$), outport = 3

Group table @s1, s4
group i
*bucket 1: weight = 50, pushmpls(tc = 4, label = 2), enqueue(5), outport = 2
*bucket 2: weight = 50, pushmpls(tc = 4, label = 3), enqueue(5), outport = 3
bucket 3: weight = 100, pushmpls(tc = $i - 1$, label = 2), enqueue(i), outport = 2
bucket 4: weight = 100, pushmpls(tc = $i - 1$, label = 3), enqueue(i), outport = 3

Flow table 1 @s3	
Match	Action
label = 3	goto_table: 2

Flow table 2	
Match	Action
inport = 1	goto_table: 3, outport = 2
inport = 2	goto_table: 3, outport = 1

Flow table 3	
Match	Action
tc = 0	enqueue(1)
tc = 1	enqueue(2)
⋮	⋮
*tc = 4	enqueue(5)
tc = 5	enqueue(6)
*tc = 6	enqueue(7)

Group table @s1, s4
group i $(i = 5)$
*bucket 1: weight = 50, pushmpls(tc = 6, label = 2), enqueue(7), outport = 2
*bucket 2: weight = 50, pushmpls(tc = 6, label = 3), enqueue(7), outport = 3
bucket 3: weight = 100, pushmpls(tc = i, label = 2), enqueue($i + 1$), outport = 2
bucket 4: weight = 100, pushmpls(tc = i, label = 3), enqueue($i + 1$), outport = 3

Without Kuijia, actually the simpler design for rescaling can only have one group table, but as our Kuijia wants to reroute the victim flows with lower priority in order not to affect the aboriginal flows, we need more group tables to distinguish those flows.

Figure 7: The design of flow table and group table of each switch in the simple topology under multiple priorities.

tables is about 0.36 times the slope of the curve for simple replicating method. This means our method can reduce more and more memory when the number of priorities increases (e.g., when there are 8 priorities, more than half of the memory of flow entries is saved).

5. Evaluation

We conduct comprehensive testbed experiments on Emulab to assess the effectiveness of Kuijia in this section. The evaluation details for nonpriority traffic are in the first three subsections and the fourth one shows the evaluation for multiple priority traffic.

5.1. Setup

Testbed Topology. We adopt a small-scale WAN topology for Google's inter-data center network reported in [1], which we refer to as the Gscale topology. There are 12 switches and 19 links as illustrated in Figure 9. We use a d430 node in Emulab running OVS to emulate a WAN switch in Gscale. Each link capacity is 1 Gbps. Each switch port has three queues: queue 0 is for control messages, queue 1 is for normal flows, and queue 2 is for rescaled flows. We test both TCP and UDP traffic sources using `iperf`.

TE Implementation. Similar to prior work [2, 6], we assume that there are 3 TE tunnels or paths between an ingress-egress switch pair. We use edge-disjoint paths whenever possible. The TE solution is obtained by solving a throughput maximization program using CVX. The corresponding group tables and flow tables are then configured by a RYU controller [25] at each switch. Rate limiting is done by the `Linux tc`.

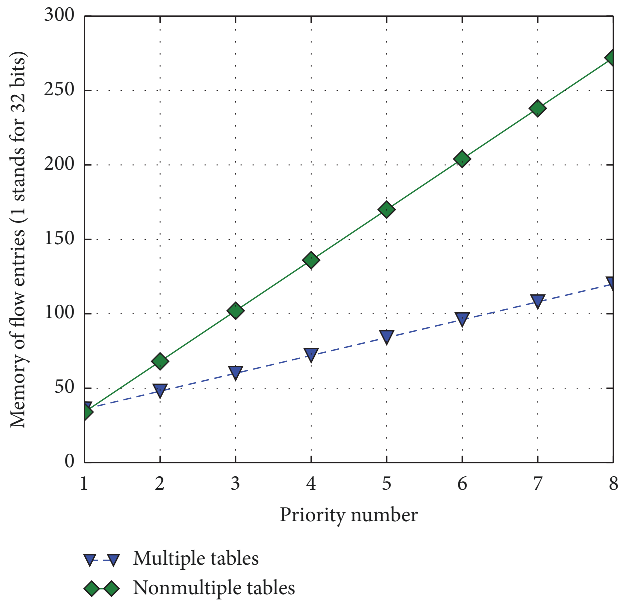


Figure 8: Memory utilization comparison of two designs for flow entries under multiple priorities.

Instead of generating a large number of individual flows between an ingress-egress switch pair, we simply launch 2 `iperf` aggregated flows on each TE tunnel and rescaling will reroute them to the two remaining tunnels separately after a single link failure. In total, there are 6 `iperf` aggregated flows for an ingress-egress switch pair. We determine the bandwidth of each `iperf` aggregated flow according to the weights of the tunnels. For example, if the TE result shows the bandwidth allocated to a switch pair is 300 Mbps and weights

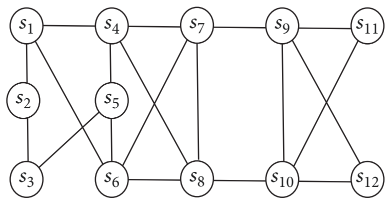


Figure 9: The Gscale topology.

for each tunnel are 0.5, 0.3, and 0.2, the bandwidth of the two `iperf` flows on the first tunnel is $300 * 0.5 * 0.3/(0.3 + 0.2) =$ 90 Mbps and 60 Mbps, respectively. Similar to BwE [21], we use the DSCP field to carry the path ID in the packet header, since Emulab uses VLAN internally to connect its machines. We use the ECN bit as the priority tag. In environments when ECN or DSCP is already used, we can use other fields in IP options or MPLS for these purposes.

Now, since we do not have many flows, rescaling is implemented by a controller changing the action of the flow entries for the victim flows, so they are routed to the remaining tunnels. For Kuijia, the controller also changes the priority tag and sends the victim flows to the low priority queue after a failure.

Traffic. We use five random ingress-egress switch pairs in each experiment. We vary the demand of each switch pair from 0.8 Gbps to 1.6 Gbps in order to see Kuijia's performance with different extents of congestion. For each demand, we repeat the experiment three times and report the average.

5.2. Benefit of Kuijia. We first look at the benefit of Kuijia compared to rescaling. Three types of flows are affected by link failures. The first is the victim flows that are routed through the failed link. The second type is the directly affected flows, which are routed through path segments that the victim flows are rescaled to. The third type is the indirectly affected flows, which pass through path segments that the directly affected flows use. As these flows are hardly affected (less than 1% for rescaling and almost no effect for Kuijia in the experiments), we do not include them in the figures. Here, we focus on the directly affected flows. The results of victim flows are discussed in the next subsection.

For TCP flows, we evaluate the throughput loss after the failure for the directly affected flows shown in Figure 10. As the demand of each ingress-egress switch pair increases, the average throughput loss in terms of percentage for directly affected flows increases with the simple rescaling. This is because as demand increases, more links in the network may be fully utilized even before failure. After rescaling, they become congested and all flows passing these links suffer throughput loss. For Kuijia, as we reroute the victim flows with low priority, they are the only flows suffering packet loss and throughput degradation after failures. Thus, even when the demand is 1.6 Gbps for each ingress-egress switch pair, the average throughput loss of directly affected flows is little.

We also look at the convergence time of TCP after the link failure, which measures how long it takes for all flows to achieve stable throughput. Again, due to the cascading effect

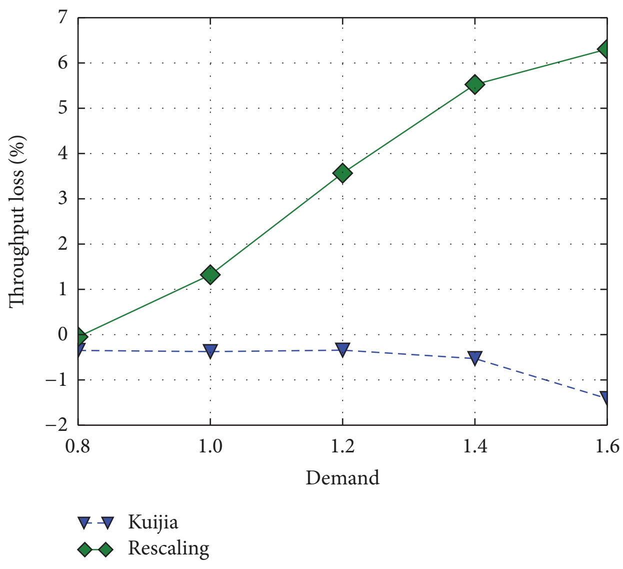


Figure 10: Throughput loss of directly affected TCP flows.

Table 4: Comparison of average TCP convergence time (s).

Link failure	Links 2-3 down	Links 7–9 down	Links 10-11 down
	Demand 0.8 Gbps		
Rescaling	1	1	1
Kuijia	<1	<1	<1
	Demand 1.0 Gbps		
Rescaling	3.75	4.50	1.75
Kuijia	<1	<1	<1
	Demand 1.2 Gbps		
Rescaling	11.00	12.00	12.75
Kuijia	<1	<1	<1
	Demand 1.4 Gbps		
Rescaling	10.50	16.00	12.25
Kuijia	<1	2.33	<1
	Demand 1.6 Gbps		
Rescaling	11.25	9.83	22.00
Kuijia	1.25	1.33	<1

of rescaling, all flows suffer from packet loss and enter the congestion avoidance phase. The convergence time is over 10 seconds when the demand exceeds link capacity as shown in Table 4. Now, with Kuijia, only victim flows need to back off, and thus the convergence time is greatly reduced to less than 1 second in almost all cases. One can also observe that the convergence time exhibits less variance with Kuijia compared to rescaling, since the congestion levels of tunnels can be vastly different with rescaling.

The benefit of Kuijia for UDP traffic is different. We use packet loss rate to measure the performance of UDP flows. The results are shown in Figure 11. For directly affected flows, packet loss rate with Kuijia is less than 2% in almost all cases, implying that the impact is negligible. Rescaling, on the other hand, results in much higher packet loss rates which are also increasing as demand increases.

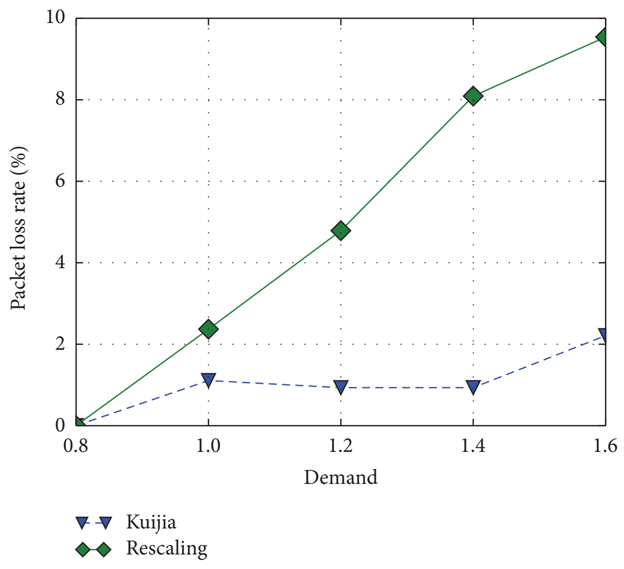


Figure 11: Packet loss rate of directly affected UDP flows.

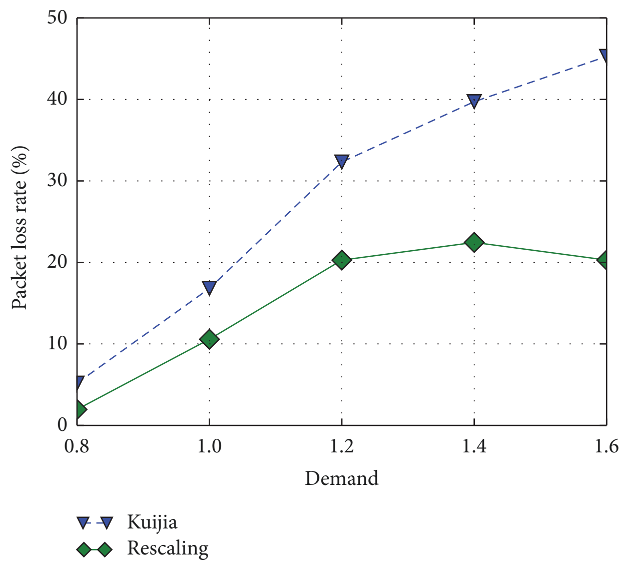


Figure 13: The overhead of UDP victim flows.

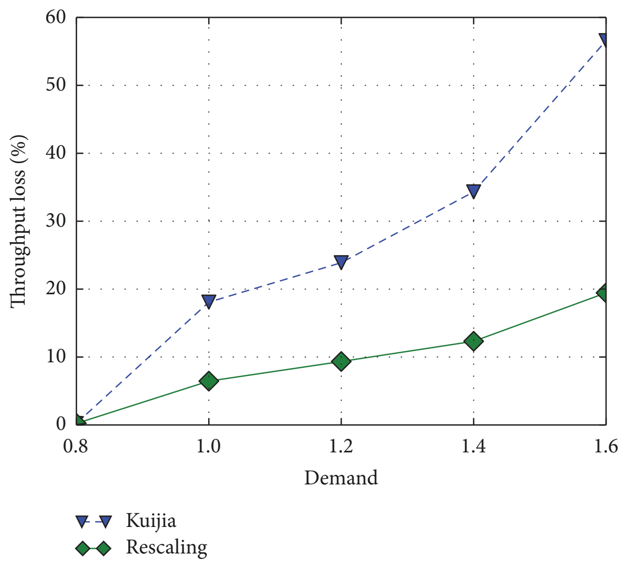


Figure 12: The overhead of TCP victim flows.

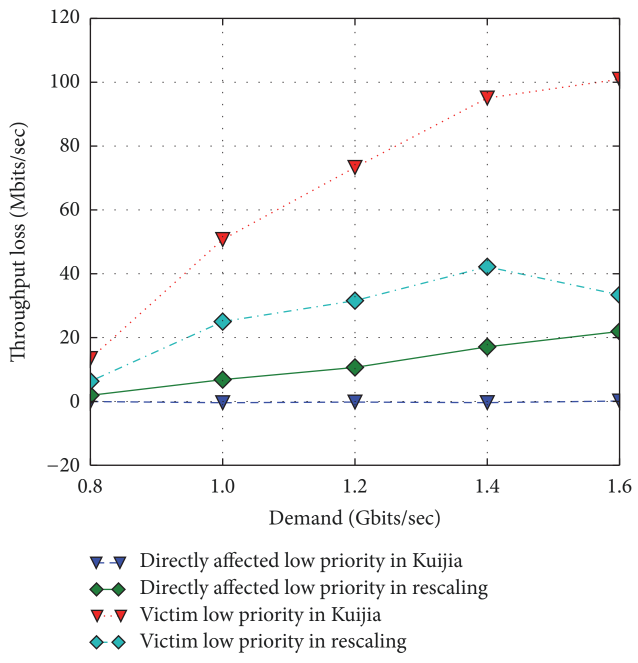


Figure 14: Throughput loss of low priority directly affected flows and victim flows under multiple priorities.

5.3. Overhead. Victim flows perform worse in Kuijia compared to rescaling, since they are the only flows that suffer throughput loss due to failures. We now look at this overhead of Kuijia. The result for both TCP and UDP traffic is shown in Figures 12 and 13. When demand of each switch pair increases, the average throughput loss of TCP victim flows and average packet loss rate of UDP flows also increase. We believe this is a reasonable trade-off to make, because in case of a link failure, traffic that traverses through this link is inevitably affected, especially when the demand exceeds link capacity in the first place. On the other hand rescaling causes too much collateral damage by making many other flows suffer from congestion, which should be avoided.

5.4. Benefit of Kuijia for Multipriority Traffic. We now use experiments to demonstrate the performance of Kuijia with multipriority traffic. Here, the setup is similar to Section 5.1. The difference is that now we have 4 queues for each port of the switches and each switch pair has 12 iperf TCP flows. Six flows are high priority traffic going through `queue(0)` before failure and `queue(1)` after failure if they are the victim flows. Correspondingly, the remaining six are for low priority traffic going through `queue(2)` before failure and `queue(3)` after failure if they are victim flows. Note that the setup is a simplification of the 8-priority design in Section 4.3. According to [1], we set the ratio of low/high priority traffic to 10.

The result is shown in Figure 14. As demand increases, the average throughput loss increases with simple rescaling

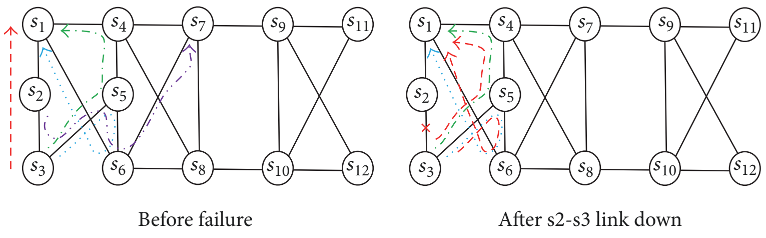


Figure 15: The partial flows' change after a link failure in one Gscale testbed experiment.

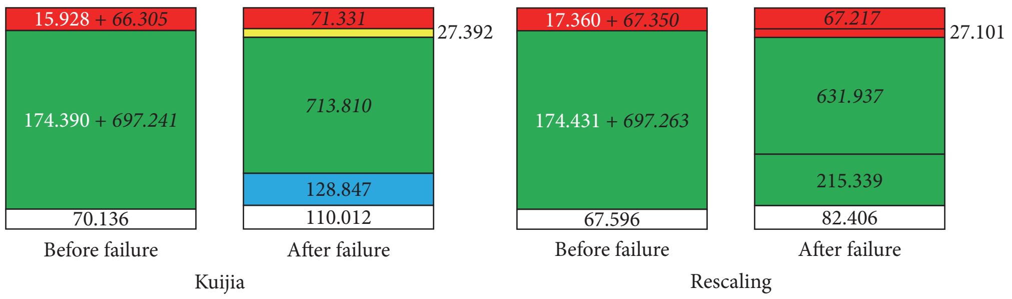


Figure 16: The traffic (Mbps) change of different priorities for flows passing s3 → s5 when demand = 1.2 Gbps.

for both victim low priority flows and directly affected low priority flows. For Kuijia, only the victim low priority flows suffer from throughput loss. The reason is similar to the explanation for Figure 10.

In order to compare the changes of traffic volume of different priorities after a link failure, we use the example as shown in Figure 15. Before failure, the purple (s2 → s3 → s5 → s6 → s7) and red (s3 → s2 → s1) flows pass through the link s2-s3 (two directions), and the green (s3 → s5 → s4 → s1) and blue (s3 → s5 → s6 → s1) flows pass through the link s3 → s5 (single direction). After the link s2-s3 link is down, the purple flows are rerouted to s7 without taking s3 → s5, so we remove it in the right topology of Figure 15. The red flows are rerouted through the remaining tunnels.

We compute the traffic volume of flows passing through the link s3 → s5 before and after failure for each priority and show the comparison result in Figure 16. The same color represents the same priority traffic, and from up to down, the priority decreases. The white area is traffic volume of the purple flows which will be rerouted away after s2-s3 link is down. As the ratio of low/high priority traffic is 10, we can observe that, for both Kuijia and rescaling, the high priority directly affected flows (the italic numbers in the top red area) and high priority victim flows (the black numbers on the right side of the yellow area for Kuijia and of the second red area for rescaling) are almost not affected by the link failure. However, for rescaling, the low priority directly affected flows suffer the throughput loss (697.263 Mbps decreases to 631.937 Mbps) as it reroutes the victim flows with unchanged priorities. When the number of priorities increases and the volume of rerouted victim flows increases, the benefit of Kuijia can be more salient.

To evaluate the benefit of our new flow table decomposition method proposed in Section 4.3, we count the number of `match` fields and `actions` of each flow entry in the experiment (e.g., a flow entry: `match` {tc = 0, inport = 1, label = 1}, `actions` {enqueue(1), outport = 2}; the number of fields is 3 + 2 = 5). We find that our new flow table decomposition method can reduce the average number of fields for all the flow entries used in the experiment at least from 2552 (without using multiple tables) to 1580. It can save about 38% memory.

6. Discussion

We now discuss several issues pertaining to the use of Kuijia in a production data center WAN.

Traffic Characteristics. The benefit of Kuijia depends on the traffic's characteristics. For elastic traffic like file transfer which is TCP friendly, no matter how large the bandwidth is, the file can be received finally. Therefore, in Google BwE [21], many of their WAN links run at 90% utilization by sending elastic traffic at a low priority. In these cases, rescaling may be good enough as it shares the bandwidth after rescaling across many users. With Kuijia, some users have to suffer significant throughput loss which may stall their transmission and hurt their experience.

However, for inelastic traffic like video conferencing, video streaming, online search, or stock trading, they all need a certain level of bandwidth to be delivered on time with quality. As a result, Kuijia is much better here as it limits the impact of failures to the victim flows. It is much worse when the cascading effect of simple rescaling causes many users to experience playback delay, whereas Kuijia limits the performance impact to only the victim flows that have to suffer from failures no matter what. With the rapid growth of mobile traffic, there will be more and more inelastic traffic, creating more use cases for Kuijia in data center WANs.

Traffic Priorities. As a data center WAN carries both elastic and inelastic traffic, it usually employs priorities to differentiate the QoS [1, 2, 21]. Inelastic traffic is given high priority while elastic traffic is given low priority [1, 2, 21]. Thus, it is important to consider multipriority when dealing with failures. In such a case, Kuijia's potential benefit can be more significant in the future, given the growth of high priority inelastic traffic such as video.

Impact of Flow Size to Hashing. In intra-data center networks, it is known that hashing based ECMP leads to suboptimal performance due to flow size imbalance. A few elephant flows may be hashed to the same path among many choices creating hotspots in the network. This does not happen in data center WANs. The WAN carries aggregated flows over more than thousands of individual TCP flows across the wide area, using a few sets of tunnels [1, 2, 6, 21]. The aggregated behavior of flows is a persistent flow with infinite data to send, which is the common abstraction used in the literature [1, 2, 6, 21]. TE calculates the splitting ratios of the aggregated flow across a few tunnels as well as the sending rate for the next interval. Hashing works well and can achieve the splitting ratios given by TE when the actual number of TCP flows is extremely large compared to the number of paths (tunnels) available. Thus, hash imbalance is not an issue.

7. Conclusion

We develop Kuijia, a robust TE system for data center WANs based on rate rescaling method, to reduce the affected flows due to data plane faults by rerouting the victim flows from failure tunnels to other healthy tunnels with lower priority. This protects the aboriginal traffic of those healthy tunnels from congestion and packet loss, as the traffic from the failure tunnels will suffer them. By evaluating our method in Emulab Gscale testbed that we implemented, the results show that Kuijia works well for both nonpriority traffic and multipriority traffic whether in pure SDN network or in a hybrid network like Emulab.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

The work is supported in part by the Hong Kong RGC ECS-21201714, GRF-11202315, and CRF-C7036-15G.

References

[1] S. Jain, A. Kumar, S. Mandal et al., "B4: Experience with a globally-deployed software defined WAN," in *Proceedings of the ACM SIGCOMM 2013 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, SIGCOMM 2013*, pp. 3–14, China, August 2013.

[2] C.-Y. Hong, S. Kandula, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving high utilization with software-driven WAN," in *Proceedings of the ACM Conference on SIGCOMM*, pp. 15–26, ACM, Hong Kong, August 2013.

[3] M. Zhang and H. H. Liu, *Private Conversation with The Authors, Microsoft Research*, March 2015.

[4] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "DevoFlow: Scaling flow management for high-performance networks," in *Proceedings of the ACM SIGCOMM*, pp. 254–265, New York, NY, USA, August 2011.

[5] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. Maltz, "zUpdate: Updating data center networks with zero loss," in *Proceedings of the ACM SIGCOMM 2013 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, SIGCOMM 2013*, pp. 411–422, China, August 2013.

[6] H. H. Liu, S. Kandula, R. Mahajan, M. Zhang, and D. Gelernter, "Traffic engineering with forward fault correction," in *Proceedings of the 2014 ACM Conference on Special Interest Group on Data Communication, SIGCOMM 2014*, pp. 527–538, USA, August 2014.

[7] S. M. Mousavi and M. St-Hilaire, "Early detection of DDoS attacks against SDN controllers," in *Proceedings of the 2015 International Conference on Computing, Networking and Communications, ICNC 2015*, pp. 77–81, USA, February 2015.

[8] S. T. Ali, V. Sivaraman, A. Radford, and S. Jha, "A survey of securing networks using software defined networking," *IEEE Transactions on Reliability*, vol. 64, no. 3, pp. 1086–1097, 2015.

[9] Q. Yan and F. R. Yu, "Distributed denial of service attacks in software-defined networking with cloud computing," *IEEE Communications Magazine*, vol. 53, no. 4, pp. 52–59, 2015.

[10] M. Suchara, D. Xu, R. Doverspike, D. Johnson, and J. Rexford, "Network architecture for joint failure recovery and traffic engineering," in *Proceedings of the the ACM SIGMETRICS joint international conference*, p. 97, San Jose, California, USA, June 2011.

[11] C. Zhang, H. Xu, L. Liu et al., "Kuijia: Traffic rescaling in data center WANs," in *Proceedings of the 37th IEEE Sarnoff Symposium, Sarnoff 2016*, pp. 142–147, USA, September 2016.

[12] M. Reitblatt, M. Canini, A. Guha, and N. Foster, "FatTire: Declarative fault tolerance for software-defined networks," in *Proceedings of the 2013 2nd ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, HotSDN 2013*, pp. 109–114, China, August 2013.

[13] S. H. Yeganeh and Y. Ganjali, "Beehive: Towards a simple abstraction for scalable software-defined networking," in *Proceedings of the 13th ACM SIGCOMM Workshop on Hot Topics in Networks, HotNets 2014*, USA, October 2014.

[14] L. Schiff, M. Borokhovich, and S. Schmid, "Reclaiming the brain: Useful OpenFlow functions in the data plane," in *Proceedings of the 13th ACM SIGCOMM Workshop on Hot Topics in Networks, HotNets 2014*, USA, October 2014.

[15] M. Borokhovich, L. Schiff, and S. Schmid, "Provable data plane connectivity with local fast failover: Introducing OpenFlow graph algorithms," in *Proceedings of the 3rd ACM SIGCOMM 2014 Workshop on Hot Topics in Software Defined Networking, HotSDN 2014*, pp. 121–126, USA, August 2014.

[16] Y. Chang, S. Rao, and M. Tawarmalani, "Robust validation of network designs under uncertain demands and failures," in *Proceedings of the USENIX NSDI*, 2017.

[17] J. Zheng, H. Xu, X. Zhu, G. Chen, and Y. Geng, "We've got you covered: Failure recovery with backup tunnels in traffic

engineering," in *Proceedings of the 24th IEEE International Conference on Network Protocols, ICNP 2016*, Singapore, November 2016.

[18] N. Katta, O. Alipourfard, J. Rexford, and D. Walker, "Rule-Caching Algorithms for Software-Defined Networks," Tech. Rep., Princeton University, 2014.

[19] The University of Utah, Emulab, 2017, http://www.emulab.net/.

[20] B. Pfaff, J. Pettit, T. Koponen et al., "The design and implementation of open vSwitch," in *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2015*, pp. 117–130, usa, May 2015.

[21] A. Kumar, S. Jain, U. Naik et al., "BwE: flexible, hierarchical bandwidth allocation for WAN distributed computing," in *Proceedings of the ACM Conference on Special Interest Group on Data Communication (SIGCOMM '15)*, pp. 1–14, London, UK, August 2015.

[22] L. Molnár, G. Pongrácz, G. Enyedi et al., "Dataplane specialization for high-performance OpenFlow software switching," in *Proceedings of the 2016 ACM Conference on Special Interest Group on Data Communication, SIGCOMM 2016*, pp. 539–552, Brazil, August 2016.

[23] J. M. Wang, Y. Wang, X. Dai, and B. Bensaou, "SDN-based multi-class QoS-guaranteed inter-data center traffic management," in *Proceedings of the 2014 3rd IEEE International Conference on Cloud Networking, CloudNet 2014*, pp. 401–406, Luxembourg, October 2014.

[24] Open Networking Foundation, OpenFlow Switch Specification 1.5.1, 2015, https://www.opennetworking.org/images//openflow-switch-v1.5.1.pdf.

[25] SDN Framework Community, RYU, 2016, https://github.com/osrg/ryu.