Kuijia: Traffic Rescaling in Data Center WANs

Che Zhang*, Hong Xu*, Libin Liu*, Zhixiong Niu*, Peng Wang*, Yongqiang Tian*, Chengchen Hu†
*NetX Lab, City University of Hong Kong, Hong Kong
†Department of Computer Science and Technology, Xi'an Jiaotong University, China

***Abstract*—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 pre-installs a lot of backup paths in the switches to ensure fast reroute, or proactively pre-reserve bandwidth to achieve fault-resiliency. Our idea agilely reacts to failures in data plane while eliminating pre-installation of backup paths. We propose Kuijia, a robust traffic engineering system for data center WANs which relies on a novel failover mechanism in data plane called rate rescaling. The affected flows on failed tunnels are rescaled to the remaining tunnels, and enter low priority queues to avoid performance impairment of abnormal flows on remaining tunnels. Real system experiments show that Kuijia is effective in handling network faults and significantly outperforms conventional rescaling method.**

I. Introduction

Increasingly, traffic engineering (TE) is implemented using software defined networking (SDN), especially in inter-data center WANs. Examples include Google's B4 and Microsoft's SWAN [6, 7]. 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 [6, 7, 9].

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 I 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 [6, 7], 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 [6, 7].

TABLE I
Link failure frequencies in Microsoft data center WAN [3].

Number of link failure	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%

Controller intervention offers the best failure recovery performance given its global network view. However, re-computing a new TE plan and updating the forwarding rules across the entire network take at least minutes and are error-prone [5, 9, 10]. 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 re-direct traffic among the remaining tunnels [9]. 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 [9].

Some solutions have emerged to solve this practically important issue. Suchara et al. [14] propose to pre-compute 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. [9] 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 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,[1] a robust TE system for data center WANs that answers this question in the positive. 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 by ECMP to the remaining tunnels, but they now enter a low 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 of controller intervention.

Kuijia with rate rescaling offers an advantage over simple rescaling in ECMP. 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

[1]The word "Kuijia" means armour in Chinese. Kui is for protecting the head and neck, and Jia is for protecting the torso.

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.

II. Motivation

We motivate our idea using a simple example. Fig. 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 8Gbps traffic. When link s2–s4 in T1 fails, s1 rescales the traffic to the remaining two tunnels, resulting in traffic distribution as shown in Fig. 1(b). Since the traffic is still sent at 24Gbps, the remaining tunnels T2 and T3 need to carry 12Gbps each and are heavily congested.

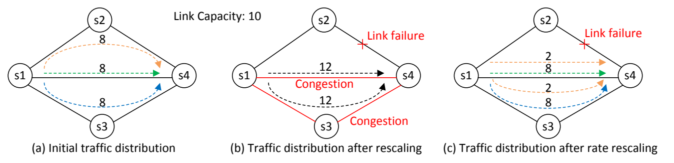

Link Capacity: 10
s1
s2
s3
s4
8
8
8
Link failure
12
Congestion
12
Congestion
2
8
2
8
(a) Initial traffic distribution
(b) Traffic distribution after rescaling
(c) Traffic distribution after rate rescaling

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

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 Fig. 1(c). The victim traffic (shown in yellow) uses the remaining capacity of T2 and T3 and sends at 2Gbps 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 [1]. We connect 4 Emulab servers running OpenvSwitch (OVS) [11] to form the same topology as in Fig. 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 egress 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.

We run two experiments with different extent of congestion to demonstrate the effectiveness of Kuijia. Table II shows

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

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

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

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

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

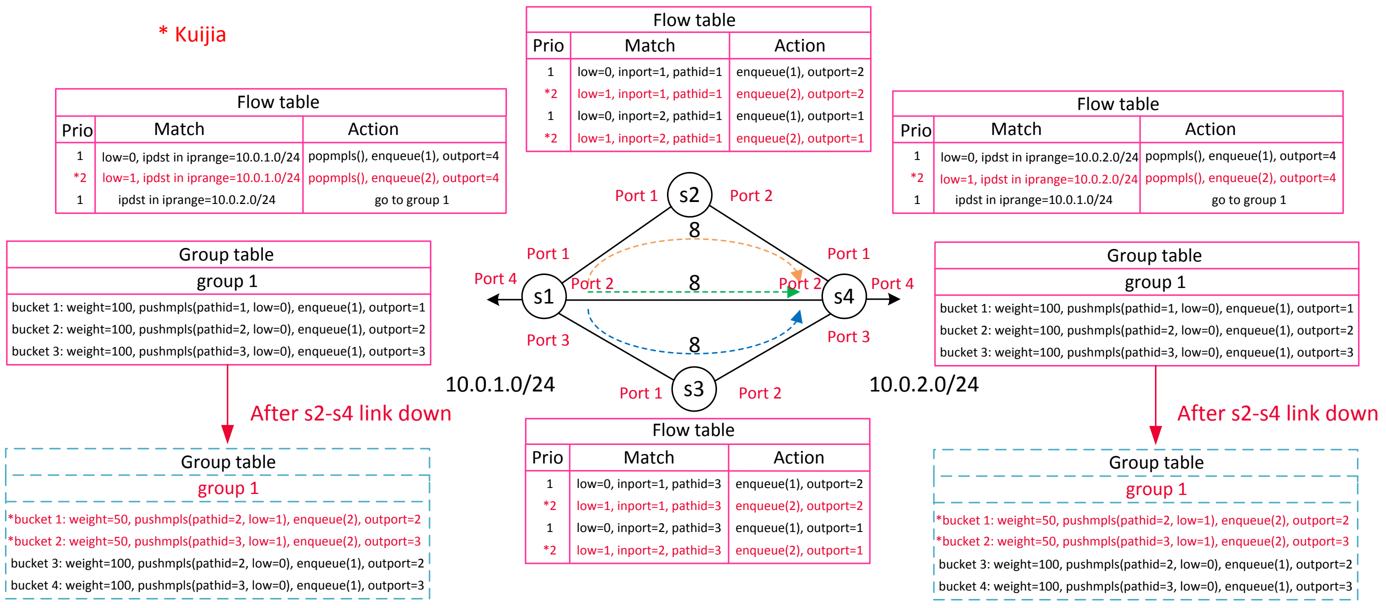
Table III shows the result when f3 and f4 send at 1Gbps, and f1 and f2 at 500Mbps each before failure. This represents the case when the remaining tunnels do not have any capacity for the victim traffic. Rescaling again causes severe congestion to aboriginal traffic on the remaining tunnels, and after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

III. Design

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

A. 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 [7, 8]. Label-based forwarding is usually used to reduce forwarding complexity [6]. 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, Fig. 2 shows the group tables



Flow table

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

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

Group table

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

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

Flow table

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

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

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

Group table

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

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

and flow tables of four switches for the network used in Fig. 1. The forwarding label can be MPLS, VLAN tags, etc.

Flows are hashed to different tunnels consistently (and applied different labels) when they arrive at the ingress switch for simplicity. Thus splitting weights are configured as ranges of the hashed values. For example in Fig. 3(a), the weights are 0.5, 0.3 and 0.2 for tunnels T1, T2 and T3. For simple rescaling, its implementation is as follows. Suppose the 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 Fig. 2.[2] 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 re-routing the victim traffic [9].

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

Fig. 3. The change of hash range after failure.

[2]Entries with * only exist in Kuijia, not in rescaling.

B. *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 I. 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 source-destination 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 Fig. 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 influence the original flows.

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 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 Fig. 4. This is simple to implement in the data plane and can handle any link failures quickly.

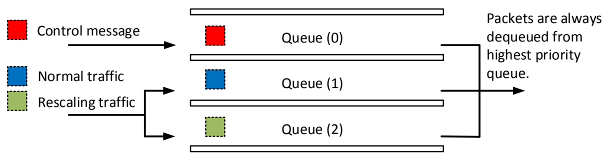


Fig. 4. The switch queues in Kuijia.

For example, in Fig. 2, for normal flows 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 re-routed flows due to link failure, e.g. when the s2-s4 link down. In the ingress switch s1, the group table applies low priority tags to the victim flows (entries with *), and direct the packets to the outport which is connected to the next-hop switch.

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

Note that as TCP connection needs two-way communication, the flow entries and group entries are also issued for two-way. 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 normal flow and 1 means re-route flow).

IV. Evaluation

We conduct comprehensive testbed experiments on Emulab to assess the effectiveness of Kuijia in this section.

A. Setup

Testbed Topology. We adopt a small scale WAN topology for Google's inter-data center network reported in [7], which we refer to as the Gscale topology. There are 12 switches and 19 links as illustrated in Fig. 5. 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`.

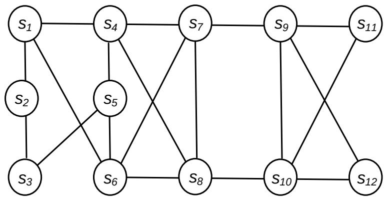


Fig. 5. The Gscale topology.

TE Implementation. Similar to prior work [6, 9], 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 [2] at each switch. Rate limiting is done by the Linux `tc`.

Instead of generating a large number of individual flows between an ingress-egress switch pair, we simply launch 2 `iperf` flows on each TE tunnel and rescaling will re-route them to the two remaining tunnels separately after a single link failure. In total there are 6 `iperf` flows for an ingress-egress switch pair. We determine the bandwidth of each `iperf` flow according to the weights of the tunnels. For example, if the TE result shows the bandwidth allocated to a switch pair is 300Mbps, and weights for each tunnel are 0.5, 0.3 and 0.2, the bandwidth of the two `iperf` flow on the first tunnel is 300*0.5*0.3/(0.3+0.2)=90Mbps and 60Mbps, respectively. 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 are already used, we can use other fields in IP options or MPLS for these purpose.

Now since we do not have many flows, rescaling is implemented by 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 send 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 0.8Gbps to 1.6Gbps in order to see Kuijia's performance with different extent of congestion. For each demand we repeat the experiment three times and report the average.

B. Benefit of Kuijia

We first look at the benefit of Kuijia compared to rescaling. Three types of flows are affected by link failures and rescaling. 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. Here we focus on the latter two types of flows. The results of victim flows are discussed in the next subsection.

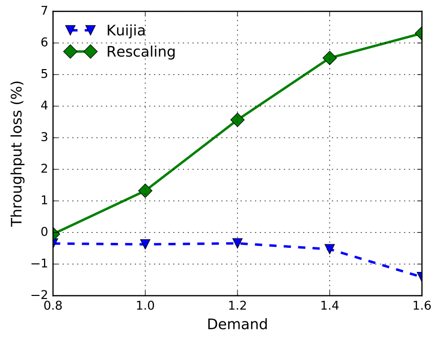


Fig. 6. Throughput loss of directly affected TCP flows.

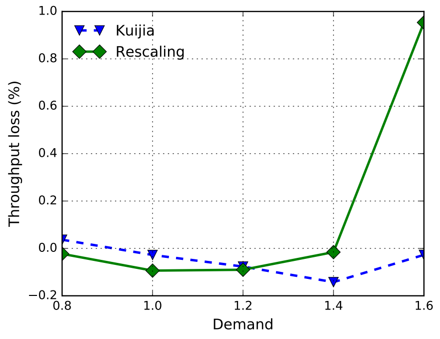


Fig. 7. Throughput loss of indirectly affected TCP flows.

For TCP flows, we evaluate the throughput loss after the failure for both the directly and indirectly affected flows are shown in Fig. 6 and Fig. 7. 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 re-route 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 1.6Gbps for each ingress-egress switch pair, the average throughput loss of directly and indirectly affected flows is little.

We also look at the convergence time of TCP after the link failure, which measure how long it takes for all flows to achieve stable throughput. Again due to the cascading effect 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 IV. Now with Kuijia, only victim flows need to back off, and thus the convergence time is greatly reduced to less than 1 seconds 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 Fig. 8 and Fig. 9. For directly affected

TABLE IV
COMPARISON OF AVERAGE TCP CONVERGENCE TIME (S)

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

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. For indirectly affected flows, the variation of packet loss after link failure is almost zero for both Kuijia and original rescaling.

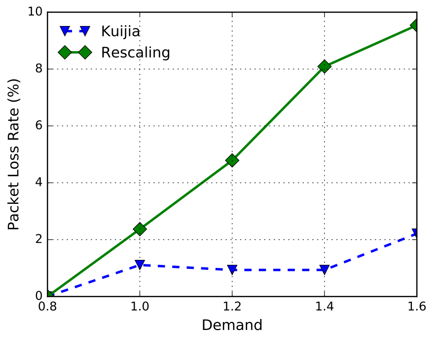


Fig. 8. Packet loss rate of directly affected UDP flows.

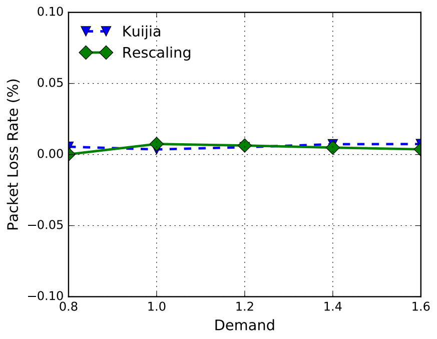


Fig. 9. Packet loss rate of indirectly affected UDP flows.

C. 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

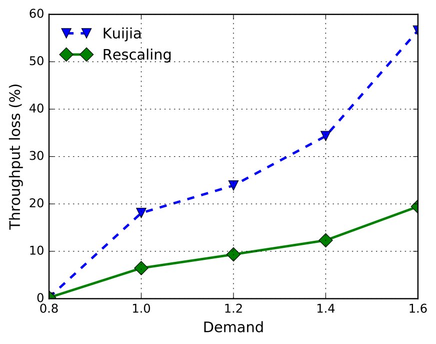


Fig. 10. The overhead of TCP victim flows.

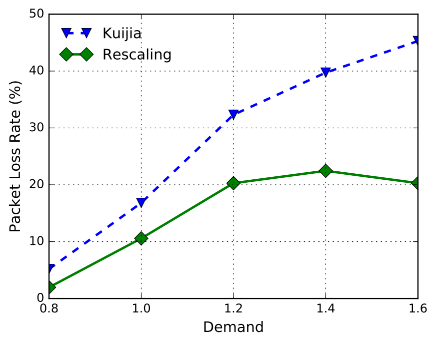


Fig. 11. The overhead of UDP victim flows.

for both TCP and UDP traffic is shown in Fig. 10 and Fig. 11. When demand of each ingress-egress switch pair increases, the average throughput loss of TCP victim flows and average packet loss rate of UDP flows also increases. We believe this is a reasonable tradeoff 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 suffering from congestion, which should be avoided.

V. Related Work

Failures in SDN. There is much work to deal with failures in SDN. [12] and [15] provide new abstractions 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. [13] propose SmartSouth to provide a new data plane for Openflow switches that can implement fault-tolerant mechanisms. Borokhovich et al. [4] develop algorithms to compute failover tables. Kuijia is different in that it focus on remedying the congestion due to rescaling.

Failures in data center WANs. [14] modifies the rescaling behaviour of ingress switch by pre-computing and configuring forwarding rules based on the likelihood of different failure cases to prevent rescaling-induced congestion after a data plane fault. SWAN [7] develops a new technique that leverages a small amount of scratch capacity on links to apply updates in a provably congestion-free manner. FFC [9] is proposed to proactively protect a network from congestion and packet loss due to data and control plane faults. Our method is different as we use priority queueing in the data plane, which is simple to implement in practice.

VI. Conclusion

We develop Kuijia, a robust TE system for data center WANs based on rate rescaling method to reduce the influenced flows due to data plane faults by re-routing the flows from failure tunnels to other healthy tunnels with low priority. This protects the original 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 we implemented, the results show Kuijia works well for both TCP and UDP traffic.

VII. Acknowledgements

We thank the anonymous reviewers for their valuable comments on this paper. The work is supported in part by the Hong Kong RGC ECS-21201714, GRF-11202315, and CRF-C7036-15G.

References

[1] Emulab. http://www.emulab.net/.

[2] RYU. https://github.com/osrg/ryu.

[3] Private conversation with Ming Zhang and Harry H. Liu, Microsoft Research, March 2015.

[4] M. Borokhovich, L. Schiff, and S. Schmid. Provable Data Plane Connectivity with Local Fast Failover: Introducing Openflow Graph Algorithms. In *Proc. ACM HotSDN*, 2014.

[5] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. Devoflow: Scaling flow management for high-performance networks. In *Proc. ACM SIGCOMM*, 2011.

[6] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving high utilization with software-driven WAN. In *Proc. ACM SIGCOMM*, 2013.

[7] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a globally-deployed software defined WAN. In *Proc. ACM SIGCOMM*, 2013.

[8] A. Kumar, S. Jain, U. Naik, A. Raghuraman, N. Kasinadhuni, E. C. Zermeno, C. S. Gunn, J. Ai, B. Carlin, M. Amarandei-Stavila, M. Robin, A. Siganporia, S. Stuart, and A. Vahdat. Bwe: Flexible, hierarchical bandwidth allocation for wan distributed computing. In *Proc. ACM SIGCOMM*, 2015.

[9] H. H. Liu, S. Kandula, R. Mahajan, M. Zhang, and D. Gelernter. Traffic engineering with forward fault correction. In *Proc. ACM SIGCOMM*, 2014.

[10] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. Maltz. zUpdate: Updating data center networks with zero loss. In *Proc. ACM SIGCOMM*, 2013.

[11] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado. The design and implementation of open vswitch. In *Proc. USENIX NSDI*, 2015.

[12] M. Reitblatt, M. Canini, A. Guha, and N. Foster. FatTire: Declarative Fault Tolerance for Software-defined Networks. In *Proc. ACM HotSDN*, 2013.

[13] L. Schiff, M. Borokhovich, and S. Schmid. Reclaiming the Brain: Useful OpenFlow Functions in the Data Plane. In *Proc. ACM HotNets*, 2014.

[14] M. Suchara, D. Xu, R. Doverspike, D. Johnson, and J. Rexford. Network architecture for joint failure recovery and traffic engineering. In *Proc. ACM Sigmetrics*, 2011.

[15] S. H. Yeganeh and Y. Ganjali. Beehive: Towards a Simple Abstraction for Scalable Software-Defined Networking. In *Proc. ACM HotNets*, 2014.