Practical PFC Deadlock Prevention in Datacenter Networks

ABSTRACT

to be written.

1. Introduction

In this paper, we present a simple and practical mechanism called Tagger to prevent deadlocks in large deployments of Remote Direct Memory Access (RDMA) over Ethernet (RoCE).

Public cloud providers like Microsoft [11] and Google [8] are deploying RoCE in their data centers to provide low latency, high throughput network transfers with minimal CPU overhead [11]. RoCE uses Priority Flow Control (PFC) to prevent packet losses due to buffer overflow at the switches. PFC allows a switch to temporarily pause its immediately upstream neighbor to prevent buffer overflow. While PFC is effective, it can cause numerous problems [11], including deadlocks [4, 5, 7].

The deadlock problem is not merely theoretical – our conversations with engineers at large cloud providers confirm that they have seen the problem in practice and at least one provider has reported it publicly [4]. Deadlock is a serious problem because a deadlock is not transient – once a deadlock forms, it does not go away even after the conditions (e.g. a temporary routing loop due to link failure) that caused its formation have abated [4]

Deadlock in PFC-enabled network can occur in numerous ways [5], although in all cases, there is a circular buffer dependency (CBD) among the group of deadlocked switches. We will describe CBD more formally later in the paper. For now, the simple example shown in Figure 1 is sufficient – deadlock is formed, since switch A is blocked by switch B, which is blocked by C, which is paused by A.

A number of solutions to the deadlock problem are known, and they fall in two broad categories. The first category consists of solutions that detect the formation of the deadlock and then use various techniques to break it [9]. The problem with these solutions is that they do not address the root cause of the problem – and hence cannot guarantee that the deadlock would not immediately reappear. We do not consider these solutions any further in this paper.

The second category of solutions are designed to prevent the key necessary condition for deadlock formation – namely, the CBD. There are three ways to prevent CBD. Some solutions require centralized, SDN-style routing. These solutions are difficult to deploy in existing data centers, without wholesale infrastructure changes. The second category of solutions require major changes to the routing protocols [**?**], and thus cannot be implemented using commodity switches. Many of these solutions also require carefully controlling path of each packet – something that is simply not possible with decentralized routing in presence of link failures [10]. Finally, there are protocols that require creation of numerous priorities and buffer management according to those priorities. For example, if each of the flows in Figure 1 had its own priority and was buffered separately at each switch, there would be no deadlock. However, each priority class requires reservation of a certain amount of "headroom" buffer space, which limits the total number of priorities one can support. This headroom depends, among other factors, on the cable length – since the buffer must be large enough to absorb all packets in flight. Given the long cable lengths used in modern data centers, and the shallow buffers available on commodify switches, modern data center can typically support only two or three lossess priorities [4].

Given these shortcomings, to the best of our knowledge, no deadlock free routing solution has been deployed in production networks.

In contrast, our solution, Tagger, can be implemented on existing commodity switches, and does not require any changes to the existing routing protocol deployed in the data center. It also does not require an excessive number of priorities or queues.

The key insight behind Tagger is that CBD is caused by flows that take an "abnormal" path through the network. For example, a flow may end up on an "abnormal" path after BGP reacts to link failure. Not all abnormal paths cause CBD, however. Tagger consists of a clever tagging and buffer management scheme, such that packets that may cause CBD are automatically detected diverted to a different queues. When properly configured, this avoids CBD and hence deadlock. We show that for general topologies, TTL values can be used for tagging purpose, and we design a greedy algorithm that combines multiple tags into one lossless priority and effectively reduce the number of required queues. While the greedy heuristic is not optimal, for popular data center topologies such as as FatTree, we show that we can reduce the number of priorities even further by exploiting special structure of the topology.

We have implemented and tested Tagger on commodity Arista 7050 Switches that use the popular Broadcom chipsets.

Our key contributions are as follows: first, using traces from a large cloud provider's data center, we demonstrate the practical challenges in making RoCE deadlock-free. We

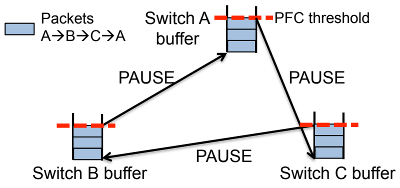


Figure 1: PFC-induced deadlock: simple illustration

also analyze Tagger using simulations and a testbed implementation. Second, we develop Tagger for general topologies and formally prove that it prevents deadlocks. Third, we optimize Tagger for popular data center topologies such as FatTree to achieve optimal results and efficiently support multiple application classes.

2. Related Work

RoCE and the need for PFC. Here we emphasize that (at least current) RoCE cannot work well without PFC. The results in DCQCN and RDMA at scale (SIGCOMM'16) paper show packet drops are very harmful for RoCE performance.

We may add a note saying that PFC has its fundamental advantage (e.g., no retransmission), if we could make it work well in practice, using PFC would be the right approach for RoCE. Our paper is trying to make this happen.

Circuit switching-based approaches. Those solutions from HPC and InfiniBand work by preemption. This does not work in Ethernet and in practice.

Brute-force buffer layering solutions. They use a lot of lossless queues. No one has discussed how they can be implemented in reality.

Summary: our differences. We believe that deciding the priority of packets along the path is better than changing routing configuratoins.

3. Challenges

3.1 Packet reroutes do happen

In a Clos based network [2, 3], if packets always follow the UP-DOWN routing, then deadlock cannot happen as CBD is not possible. In up-down routing, a packet first goes UP from the source server to one of the common ancestor switches of the source and destination servers, then it goes DOWN from the common ancestor to the destination server. In UP-DOWN routing, the following property holds: when the packet is on its way UP, it should not go DOWN; when it is on its way DOWN, it should not go UP.

Packet reroutes may happen due to several reasons including link flapping and routing protocol flapping [6]. As have shown in [9], when the UP-DOWN property is broken and packets may reroute multiple times between two layers of switches, and deadlocks may form as a result.

In this paper, we show our measurement results from a large cloud computing provider that UP-DOWN routing property does break in reality and packets can be rerouted with a non negligible probability.

Our measurement works as follows. We instrument the servers to send out IP-in-IP packets to the high-layer switches. The outer source and destination IP addresses are set to the sending server and one of the high layer switches, and the inner source and destination IP addresses are set to the switch and the sending server, respectively. The high-layer switches are configured to decapsulate those IP-in-IP packets that are targeting themselves in hardware.

Date	Total No.	Rerouted No.	Reroute probability
11/01/2016	11381533570	148416	1.3e-5
11/02/2016	11056408780	130815	1.2e-5
11/03/2016	10316034165	104472	1.0e-5
11/04/2016	10273000622	92555	0.9e-5
11/05/2016	10230003382	102872	1.0e-5
11/06/2016	10491233987	106266	1.0e-5
11/07/2016	9608289622	100916	1.1e-5

Table 1: Packet reroute measurements in the data centers of a large cloud computing service provider.

After decapsulation, the outer IP header is discarded, and the packet is then routed using its inner header. We set a TTL value, 64 in this paper, in the inner IP header. As the packet is forwarded back to the server, the TTL is decremented per hop. For a three-layer Clos network, there are three hops from the highest layer switches to the server. Hence normally the TTL value of the received packets should be 61.

If, however, the TTL value of a received packet is smaller than 61, say 59, we know the received packet was not taking the shortest path, and the packet must have taken a reroute path.

In this paper, for every measurement, a server sends out $n = 100$ IP-in-IP probing packets, if the received TTL values are not equal, we know packet reroute happened for this measurement. We then calculate the reroute probability of the measurements as $\frac{M}{N}$, where M is the number of measurements that experienced packet reroute, and N is the total number of measurements. We carried out the measurements for one week in more than 20 data centers. The measurement results are shown in Table 1.

The most important conclusion we can draw from Table 1 is that packet reroute does happen in data center networks. The reroute probability is around 10^{-5}. Though 10^{-5} is not a big number, given the large traffic volume and the large scale data center networks, the deadlocks due to packet reroute as discussed in [4, 5, 9] do not just exist in paper designs. They are real!

Therefore how to address potential deadlocks due to packet reroute becomes a pressing challenge.

3.2 Adapting to Existing Routing

We do not want to force the network operator to change any routing configuration. Existing routing configuration has been designed in a way that best fit the needs. But common routing may lead to deadlock.

Show our bouncing data from production here.

3.3 Limited Number of Lossless Queues

The number of lossless queues that can implemented on each switch is limited by two factors. First, the popular ASICs usually support up to eight different classes, and a few of which must be reserved for lossy traffic. Second, each lossless queue requires certain amount of buffer to operate. This further limits the number of lossless queues.

Below is the analysis of how many lossless queues we can have based on PFC headroom and buffer space.

A PAUSE message sent from a receiver to an upstream sender needs some time to arrive and take effect. To avoid packet drops, the receiver (i.e., the sender of the PAUSE message) must reserve enough buffer to accommodate any packets it may receive during this time. In this part, we calculate the amout of buffer needed as the PFC headroom.

Per port per priority PFC headroom: At first, we calculate the time for a PFC message to arrive and take effect at its destination port. It mainly consists of six parts.

1. **The time to send a PAUSE message at the receiver side (denoted as t_{snd}):** The transmission of the PAUSE frame can pass ahead of any other packet queued in the receiver, but cannot preempt another frame currently being transmitted in the same direction. Hence in the worst case, the receiver generates a PAUSE frame right when the first bit of a maximum-size packet (i.e., the size is equal to Maximum transmission unit (MTU)) has started engaging the transmission logic. So we have $t_{snd} = (s_{MTU} + s_{PFC})/r_l$, where s_{MTU} is the value of MTU, s_{PFC} is the size of PFC message and r_l is the line rate of the network link.

2. **The time for the PAUSE message to propagate from the receiver to the sender over the network link (denoted as t_{wire}):** The value of t_{wire} is related to the materia and the length of the network links in use.

3. **The time to receive a PAUSE message at the sender side (denoted as t_{rev}):** It is easy to know $t_{rev} = s_{PFC}/r_l$.

4. **The time to process a PAUSE message at the sender side (denoted as t_{pro}):** After a PAUSE message has been received by the sender, it will take an implementation-dependent amount of time to process the message and stop packet transmission.

5. **The time to stop packet transmission at the sender side (denoted as t_{stop}):** After the sender finally decides to stop packet transmission, it can stop only at packet boundaries, to avoid packet corruption. In the worst case, the sender will have completed the process of the PAUSE message just when the first bit of a maximum-size packet has started engaging the packet transmission. So we have $t_{snd} = s_{MTU}/r_l$

6. **The time for the remaining bytes of packets on the link to get drained after stopping packet transmission at the sender(also denoted as t_{wire}):** This part of time is equal to the time for the PAUSE message to propagate from the receiver to the sender.

In summary, the per port per priority PFC headroom can be expressed using the following equation:

$$\begin{aligned} b_{hr} &= r_l * (t_{snd} + t_{wire} + t_{rev} + t_{pro} + t_{stop} + t_{wire}) \\ &= 2(s_{MTU} + s_{PFC} + r_l * t_{wire}) + r_l * t_{pro} \qquad (1) \end{aligned}$$

For typical TCP/IP based RDMA DCNs, we have $s_{MTU} =$ 1500bytes, $s_{PFC} =$ 64bytes. The length of links used in a single DCN is usually no larger than 300 meters [4]. As every 100 meters of link delays the reception of a packet by about 500ns for copper cables [1], we have $t_{wire} \leq 1.5us$.

The value of t_{pro} is implementation related. Let a quanta be the time needed to transmit 512 bits at the current network speed. The PFC definition caps this time to 60 quanta for any implementation [1]. Hence we have $r_l * t_{pro} = 512 * 60 = 30,720$ bits.

According to the above parameters, when $r_l = 40Gbps$, the per port per priority PFC headroom $b_{hr} \leq 21968$ bytes $\approx$ 22 KB.

PFC headroom of a switch: For a n port switch which supports k priority classes, PFC PAUSE is possible to be triggered at all ports and priority classes simultaneously. So we should at least reserve $n*k*b_{hr}$ buffers as PFC headroom at every switch.

For commodity switches like Arista 7050QX32 which has 32 full duplex 40Gbps ports, when supporting 8 priority classes, it requires about $32 * 8 * 22 = 5632$KB buffer as the PFC headroom.

Reducing the PFC headroom: For tree-based topologies like Fat-tree, packets of consecutive priority classes will not enter the same switch. So the number of priority classes a switch needs to support is halved. This reduces the needed PFC headroom to 2816KB, which is about 22.9% of the total 12MB switch buffer.

4. Solution: Tag-based System

We explore the practical solution for preventing PFC deadlock without any specific assumptions on the topology and routing that operator chooses.

4.1 Key Ideas

1. Our first insight is to change the lossless queue of a packet right before it may cause cyclic buffer dependency if it does not change priority.

2. Each switch must make the decision for each packet locally and distributedly, based on the packet's past path. This operation should be purely done in the data plane.

3. The way the switch gets a packet's past path is by the tag the packet carries. We encode the information of past path into the tag, and let switches change the tag along the path.

The overhead of this design is that we will need *multiple* switch lossless queues to support one lossless application class. This overhead is inevitable given that routing is not changed. As the first step, we focus on supporting *just one* lossless application class. In Section 5, we will revisit this issue and show how we may support multiple application classes efficiently.

To use multiple switch lossless queues for one application class , we divide the buffer of network nodes into k partitions, and let the j-th partition associated with priority class j. If a packet is classified into priority class j, it will be buffered in the j-th buffer partition. Packets queued in queues $q_{in}^{i,j}$ and $q_{out}^{i,j}$ $(1 \leq i \leq n)$ are the packets currently buffered in the j-th buffer partition.

4.2 Tagging System

The tag of the packet must carry the information of each packet's past path. Thus, we divide the path of each packet into multiple stages, and use the tag to represent the stage each packet is on. We can later prove that the network is deadlock-free if the tagging system satisfies the two requirements below:

1. A packet's tag is unchanged or changed at each hop of its path. When the tag is changed, it must be changed monotonically along any packetâĂŹs path (e.g., always increasing).
2. All packets that have the same tag cannot have cyclic buffer dependency.

On each switch, packets are classified into different priority classes purely based on the tags. Then we can prove that our network is deadlock-free.

Proof of deadlock-free Need to rewrite the following.

Claim: our TTL-based solution is deadlock-free regardless of the packet scheduling algorithms.

Proof: To prove our TTL-based solution is deadlock-free, we prove by contradiction that no legal buffer state can be deadlocked buffer state under our TTL-based solution.

Assuming there exists a legal buffer state $BS_N(t)$ which is also a deadlocked buffer state under our TTL-based solution. If no new packets are injected into the network since t, according to Equation (**??**), $BS_N(t)$ will converge into a fixed non-empty buffer state $BS_N(t_0)$ at some finite time $t_0 > t$.

Case 1: All the VEQs in $BS_N(t_0)$ is empty. As $BS_N(t_0) \neq BS_N^0$, $BS_N(t_0)$ has at least one non-empty VIQ. As $BS_N(t)$ is a legal buffer state, accroding to Equation (**??**), there are finite number of packets queued in non-empty VIQs of $BS_N(t_0)$. Then according to Equations (**??**), (**??**) and (**??**), any unscheduled packet remaining in any VIQ will be forwarded to some VEQ within finite time at some finite time $t_2 > t_0$. This means that $BS_N(t_0)$ will transition to some other buffer state, which violates the fact that $\forall t_1 > t_0, BS_N(t_1) \equiv BS_N(t_0)$.

Case 2: There exist some non-empty VEQs (at least one) in $BS_N(t_0)$. Let $q_{out}^{i,m}$ be the queue of highest priority class among all the non-empty VEQ queues $(m \leq k)$. Under our TTL-based solution, packets in $q_{out}^{i,m}$ will not be paused by PFC PAUSE messages as packets can only be paused by packets of higher priority class. According to Equations (**??**) and (**??**), packets in $q_{out}^{i,m}$ will be transmitted to next hop within finite time. This means that $BS_N(t_0)$ will transition to some other buffer state at some finite time $t_2 > t_0$, which violates the fact that $\forall t_1 > t_0, BS_N(t_1) \equiv BS_N(t_0)$.

Based on the above discussion, the assumption we made will cause contradiction in both cases. Hence $BS_N(t)$ is not a deadlocked buffer state under our TTL-based solution. So our TTL-based solution is deadlock-free regardless of the packet scheduling algorithms.

TTL-based tagging system. In a generic topology and routing, TTL is a natural tagging system that satisfies our requirements.

TTL-based packet buffering:

1. Initially, we set the TTL values of all packets to $ttl_0 = d$ at all the source servers. TTL value of every packet will be decreased by 1 per hop. At all the source servers, packets are buffered in a buffer of priority class 0.
2. Let ttl_i be the TTL value of a packet p at its i-th hop $(i \geq 0)$. If $ttl_i = 0$, packet p will be dropped by the receiving switch or server. At every hop, the priority class of any incoming packet p is calculated as $\lambda_p = ttl_0 - ttl_i$. Packet p will be buffered and queued according to the calculated priority class λ_p at every hop.

The drawback of TTL-based tagging is that it requires a lot of lossless queues.

4.3 Greedy Algorithm for Minimizing the Number of Lossless Queues

Here is the generic algorithm in ppt.

5. Achieving Optimal Solution for Structured Topology

5.1 Caveats of the Generic Greedy Solution

1. It does not always yield optimal results in terms of number of lossless queues, e.g., in Fat-tree.
2. It requires $M*N$ lossless queues, where M is the number of lossless application classes, N is the number of lossless queues required by each application class. In fact, we may reduce it to $M + N - 1$.

5.2 Optimizing Fat-Tree Topology

5.3 Optimizing BCube Topology

5.4 High-level Takeaways

0

6. Implementation

The output of our algorithm can already be implemented in the commodity switches. In this section, we describe the details of our implementation, starting by explaining how the commodity shared-buffer switch maintains the queues and handles PFC.

6.1 Switch Model

In this part, we abstract the switch model on which we implement our algorithm output. Most commodity shared-buffer switches comply with this model. As shown in Fig. 2, our switch model consists of five modules: switch port, ingress pipeline, buffer management module, egress pipeline and PFC module. The function of these modules are list as follows.

- **Switch port**: The module to receive and send packets.
- **Ingress pipeline**: The module to execute packet processing tasks before a packet enters switch buffer. These tasks include packet header parsing, L2/L3 lookup, ingress ACL processing, packet tunneling, packet mirroring, etc. This module has multiple match-action stages to fulfill its tasks. Our (tag, port)-to-priority mapping function is implemented as one match-action stage here.
- **Buffer management module**: The module to manage the switch buffer. This module has m ingress queues and m egress queues. Assuming the switch has n ports and supports k priority classes. Each port has k corresponding ingress queues and k corresponding egress queues to buffer packets of k priority classes. It is easy to know m = n*k. In the module, there is also a scheduler to forward packets to egress queues based on the packet metadata calculated at ingress pipeline.

  This module maintains separate counters to track the length of each ingress queue and each egress queue (denoted as ingress counter and egress counter). The ingress counter increases when a packet enters the ingress queue, and decreases when a packet leaves the switch buffer. Egress counter works similarly. Note that when a packet is forwarded to an egress queue from some ingress queue, the corresponding ingress counter will not decrease at this stage as this packet still remains in the switch buffer.
- **Egress pipeline**: The module to execute packet processing tasks after a packet leaves switch buffer. This module also has multiple match-action stages to do tasks such as egress ACL processing, packet tunneling and packet mirroring. Our (tag, port)-to-newtag mapping function is implemented as one match-action stage here.
- **PFC engine**: The module to perform PFC function. If the switch receives a PFC PAUSE frame at some output port on some specific priority class, this module will pause the corresponding egress queue immediately. If the occupancy of some ingress queue exceeds the PFC threshold, this module will trigger the corresponding input port to send a PFC PAUSE frame to pause the upstream device on the corresponding priority.

6.2 ACL Rules for Tag Handling

To ensure packets enter the correct lossless space at each hop, our solution requires the switch to classify packets into different priority classes based on (port, tag) information. This can be done by installing some ACL rules at the ingress pipeline. As shown in Fig. 2, these rules will match the port # and value of tag, and assign packets to different priority calsses accordingly.

To optimize the number of priorities needed for Clos and BCube topologies, we need to support flexible manipulation of tag value. This can be done by installing some ACL rules at the egress pipeline. As shown in Fig. 2, these rules will calculate a new tag value for any outgoing packet based on the port # and old tag value, so the vaue of tag can be updated at each hop.

6.3 Transition of Lossless Queue

Current PFC mechanism cannot prevent packet loss in some cases when packets change their priority classes along the path. An example can be found in Fig. 3. More description about this example will be added later.

As shown in Fig. 3(c), if we want a switch to pause its upstream device correctly, for packets of the same flow, the egress queue at last hop and the ingress queue at current hop must be of the same priority class. This means that we need to decouple ingress priority from egress priority in our implementation.

Then how to do this? We notice that Broadcom chipset based commodity switches, such as Arista 7060 switch, have alternative variables that can be used to decide the priority class of a packet inside the buffer management module. In normal case, only 1 variable will be used, and ingress priority is always the same as egress priority.

We configure the switch to use different variables, denoted as ING_PRIO and EG_PRIO, to decide ingress priority and egress priority separately. To set correct value for both NG_PRIO and EG_PRIO, we install some ACL rules at the ingress pipeline to associate ING_PRIO and EG_PRIO with (port, tag) information. These rules establish separate mappings from (port, tag) to ingress priority and egress priority, respectively.

7. Evaluation

We evaluate Tagger using a combination of testbed experiments and numerical experiments. Our evaluation centers around four key questions:

- **Can Tagger prevent deadlock when deadlock-prone misconfiguration or network failure happens?**
- **Is Tagger scalable for large scale datacenters?**

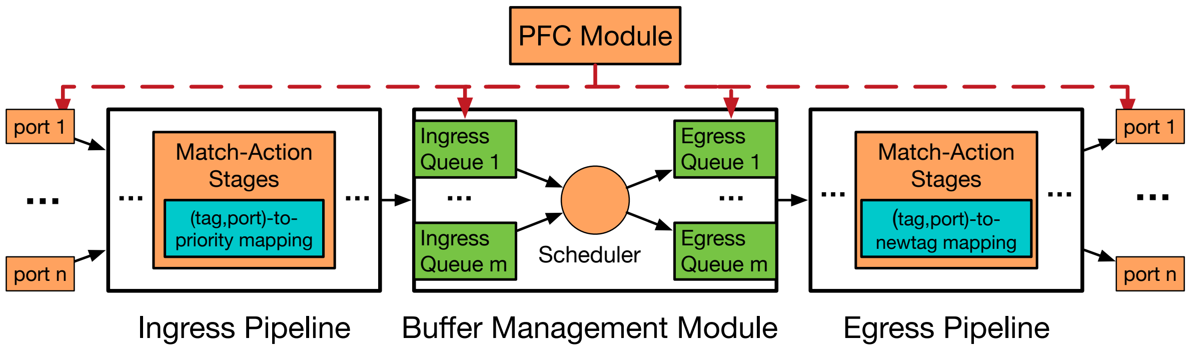
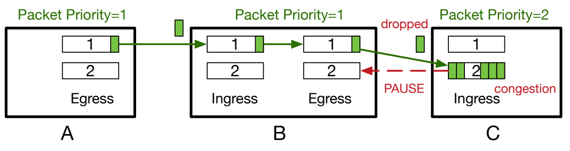
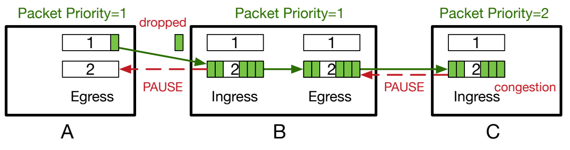


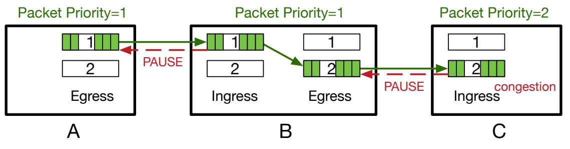
Figure 2: Switch model.



(a) Ingress priority = 1, egress priority = 1 → packet drop.

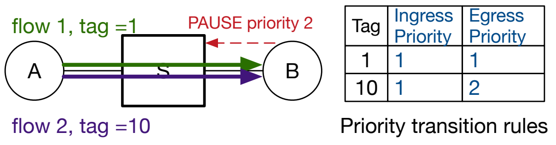


(b) Ingress priority = 2, egress priority = 2 → packet drop.



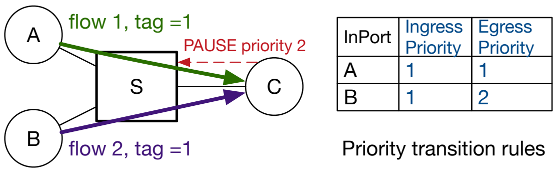
(c) Ingress priority = 1, egress priority = 2 → no drop.

Figure 3: Decoupling ingress priority from egress priority is necessary for ensuring a lossless network.



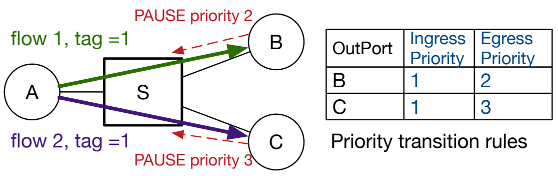
Tag	Ingress Priority	Egress Priority
1	1	1
10	1	2

(a)



InPort	Ingress Priority	Egress Priority
A	1	1
B	1	2

(b)



OutPort	Ingress Priority	Egress Priority
B	1	2
C	1	3

(c)

Figure 4: Experiment scenarios to demonstrate lossless transition between priority classes.

- **Can Tagger work at full line rate with little performance overhead?**
- **Does the mechanism of priority reusing cause any inefficiency or unfairness problem?**

7.1 Lossless transition between priority classes

Purpose: demonstrate our implementation ensures lossless transition between priority classes.

Scenario-1: As shown in Fig. 4(a), by pausing priority 2 at server B, flow 2 is paused correctly without any packet loss, while flow 1 is not affected.

Scenario-2: As shown in Fig. 4(b), by pausing priority 2 at server C, flow 2 is paused correctly without any packet loss, while flow 1 is not affected.

Scenario-3: The scenario is shown in Fig. 4(c). By pausing priority 2 at server B, flow 1 is paused correctly without any packet loss. By pausing priority 3 at server C, flow 2 is paused correctly without any packet loss.

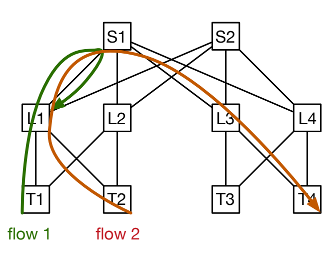
7.2 Validation

Purpose: demonstrate our solution can prevent deadlock.

Scenario-1: As shown in Fig. 5(a), In a Clos network, we generate 2 flows across different ToRs. Flow 1 enters a routing loop due to misconfiguration, and forms a deadlock. Without Tagger, flow 2 will also get paused due to propagation of PFC PAUSE. With Tagger, there is no deadlock and flow 2 is not affected by the routing loop.

Scenario-2: As shown in Fig. 5(b), we inject 3 flows into three paths that contain a CBD. Without Tagger, deadlock occurs and three flows are all paused. With Tagger, there is no deadlock and three flows achieve good throughput.

7.3 Overhead



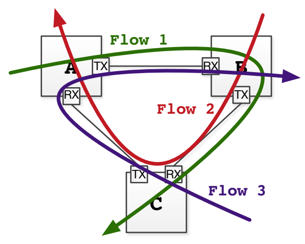


Figure 5: Scenarios of validation experiment.

7.3.1 Resource Consumption

Purpose: demonstrate our solution only consumes a small number of priorities and ACL entries.

Scenario: for Fattree, BCube and Jellyfish, we calculate the following metrics under different network scales to show Tagger is scalable in terms of resource consumption.

1. # of lossless priorities;
2. # of total ACL rules;
3. # of ACL rules at the bottleneck switch.

7.3.2 Performance overhead

Purpose: demonstrate performance overhead of Tagger is small.

Scenario-1: We let a flow traverse m hops in the network, and perform tagging and priority manipulation at every hop. We can evaluate the performance overhead of Tagger by measuring the throughput and latency when using different values of m.

Scenario-2: we increase the # of ACL rules installed in a switch, and observe the performance overhead of the flows passing over this switch.

7.4 Impact of priority reusing

Purpose: demonstrate the mechanism of priority reusing will not downgrade link utilization and throughput of flows (but may introduce unfairness problem).

Scenario: As shown in Fig. 6(a), a Clos network serves two users belonging to different application classes. For user 1, priority 1 is used for traffic following shortest paths, and priority 2 is used for traffic following 1-bounce paths. For user 2, priority 2 is used for traffic following shortest paths, and priority 1 is used for traffic following 1-bounce paths.

Assuming WRR policy among different egress queues. Assuming traffic of these two users shares a bottleneck link. We create a link failure to let flow 1 of user 1 enter the second lossless space. Flow 1 then competes link bandwidth in the same egress queue with flow 3 of user 2 at the bottleneck link L3-T4.

By measuring the link utilization and throughput of all flows, we can demonstrate that link utilization of the bottleneck link and overall throughput of all flows is not downgraded, but priority reusing does cause some unfairness problem among different suers.

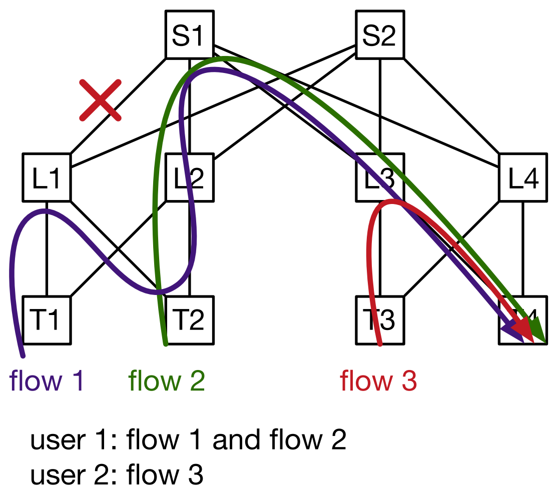


Figure 6: Scenario of priority reusing experiment.

compared scheme: no priority reusing among users (4 priorities are used)

7.5 Hierarchical lossless space improves application performance

This experiment will be moved to Section 3 as an motivation experiment.

Purpose: demonstrate that for Clos network, applications can achieve better performance when both shortest paths and 1 bounce paths are lossless.

Scenario: In a Clos network, we run both throughput-intensive and latency-sensitive applications. We randomly generate k failures in the network to have some of the flows follow 1-bounce paths.

compared schemes:

1. both shortest paths and 1 bounce paths are lossless.
2. shortest paths are lossless. 1 bounce paths are lossy.
3. Only shortest paths are allowed.

8. Conclusion

9. References

[1] Priority flow control: Build reliable layer 2 infrastructure. `http://www.cisco.com/c/en/us/products/collateral/switches/nexus-7000-series-switches/white_paper_c11-542809.pdf`.

[2] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM*, 2008.

[3] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri,

David A. Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: A scalable and flexible data center network. In *SIGCOMM*, 2009.

[4] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitendra Padhye, and Marina Lipshteyn. Rdma over commodity ethernet at scale. In *SIGCOMM '16*.

[5] Shuihai Hu, Yibo Zhu, Peng Cheng, Chuanxiong Guo, Kun Tan, Jitendra Padhye, and Kai Chen. Deadlocks in datacenter networks: Why do they form, and how to avoid them. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, pages 92–98. ACM, 2016.

[6] Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, and Thomas Anderson. F10:A Fault-Tolerant Engineered Network. In *NSDI*, 2013.

[7] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. Brent, stephens and alan, l. cox and ankit, singla and john, carter and colin, dixon and wesley, felter. In *INFOCOM '14*.

[8] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. Timely: Rtt-based congestion control for the datacenter. In *SIGCOMM '15*.

[9] Alex Shpiner, Eitan Zahavi, Vladimir Zdornov, Tal Anker, and Matty Kadosh. Unlocking credit loop deadlocks. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, pages 85–91. ACM, 2016.

[10] Xin Wu, Daniel Turner, Chao-Chih Chen, David A. Maltz, Xiaowei Yang, Lihua Yuan, and Ming Zhang. Netpilot: Automating datacenter network failure mitigation. In *SIGCOMM '12*.

[11] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale rdma deployments. In *SIGCOMM '15*.