Unlocking Credit Loop Deadlocks

Alexander Shpiner, Eitan Zahavi, Vladimir Zdornov, Tal Anker and Matty Kadosh
Mellanox Technologies, Inc
Yokneam, Israel
{alexshp,eitan,vladimirz,ankertal,mattyk}@mellanox.com

Abstract

The recently emerging Converged Enhanced Ethernet (CEE) data center networks rely on layer-2 flow control in order to support packet loss sensitive transport protocols, such as RDMA and FCoE. Although lossless networks were proven to improve end-to-end network performance, without careful design and operation, they might suffer from in-network deadlocks, caused by cyclic buffer dependencies. These dependencies are also known as *credit loops*. Although existing credit loops rarely deadlock, when they do they can block large parts of the network. Naive solutions recover from credit loop deadlock by draining buffers and dropping packets. Previous works suggested credit-loop avoidance by central routing algorithms, but these assume specific topologies and are slow to react to failures.

In this paper, we present distributed algorithm to detect, assure traffic progress and recover from credit loop deadlock for arbitrary network topologies and routing protocols. The algorithm can be implemented in commodity switch hardware, requires negligible additional control bandwidth, and avoids packet loss after the deadlock occurs. We introduce two flavors of the algorithm and discuss their trade-off. We define a simple scenario that assures credit loop deadlock to occur and use it to test and analyze the algorithm. In addition, we provide simulation results over a 3-level fat-tree network. Last, we describe our prototype implementation in commodity data center switch.

1. INTRODUCTION

The recent trend in data center networking is lossless networks. Lossless networks provide multiple performance advantages to data center applications. First, several data center transport protocols such as RDMA and FC are sensitive to packet drops, since they were designed for lossless layer-2 protocols. Second, even for reliable transport protocols, such as TCP, dropping packets results in bandwidth decrease, significant completion time increase and statistical tail, especially with short-lived flows [17].

Nevertheless, the losslessness property has also some disadvantages that harm its adoption. One of these drawbacks is the well-known credit loop deadlock[1] phenomena [14]. Given a topology and routing rules, credit loop is defined as a cyclic sequence of switch buffers, such that every switch in the sequence sends traffic to the next switch in the sequence. This sequence is dependent on the existence of at least one routing rule[2] that forwards packets incoming to the switch from a link in the sequence to the next link in the sequence. The credit loop deadlock occurs when all the switch buffers on the sequence become full and no traffic can be propagated. Credit loops are a silent killer, they may exist for a long time, but they will deadlock only with specific traffic pattern. Under deadlock, congestion may spread to significant parts of the fabric.

A simple example for a credit loop over fat-tree topology is presented on Figure 1. The network consists of four racks A, B, C and D, two core switches $11, 12$ and four leaf switches $21, 22, 23, 24$. Assume the links between switches $12-21$, and $11-14$ failed so traffic cannot be forwarded

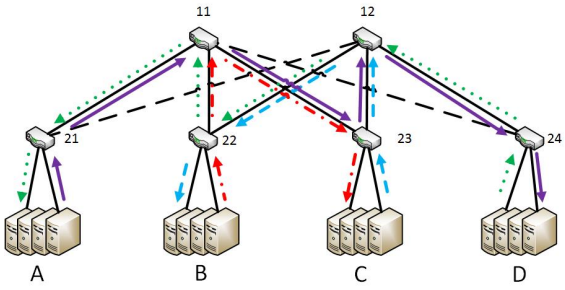


Figure 1: Example of a simple 2-level fat tree topology with the credit loop formed by failure of links 12-21 and 11-23.

[1]The name of the phenomena "credit loop deadlock" comes from ATM and Infiniband networks, which use credits in order to indicate buffer space in the next switch.

[2]Combination of forwarding rules and traffic class mapping.

HotNets-XV, November 09-10, 2016, Atlanta, GA, USA

DOI: http://dx.doi.org/10.1145/3005745.3005768

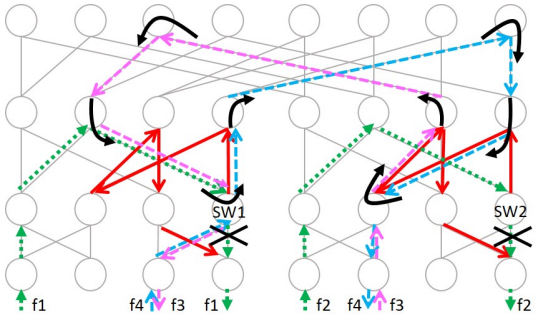


Figure 2: F10 topology, on which local reroutes by $SW1$ and $SW2$ recover flows $f1$ and $f2$ after the failure of the crossed links. However, the new routes create a credit loop with flows $f3$ and $f4$, denoted by the black curved arrows.

through them. The failure of these links breaks the shortest route between racks A and D, and force the traffic between A and D to flow via the two core switches. The example includes four flows between racks as follows: from A to D, from D to A, from B to C and from C to B. When the routes are forwarded as shown on Figure 1, a credit loop is created on the switches sequence $11 \rightarrow 23 \rightarrow 12 \rightarrow 22 \rightarrow 11$.

Other reasons for credit loops to form, even momentarily, may arise from the complexity of SDN controllers network updates [6]. Credit loop can also occur when the topology is connected incorrectly due to human mistake. We will demonstrate such a case in Section 4.3.

In hyperscale data center networks, even when central controller is used, the routing algorithm suffers a significant delay in response to link failures. To overcome this slow response and avoid packet drops, local rerouting algorithms, such as $F10$ [8], were introduced. However, these algorithms have the tendency to form credit loops. An example of a credit loop caused by $F10$ is shown on Figure 2. Assume the crossed links below $SW1$ and $SW2$ failed. $F10$ topology is built to allow the switches to avoid packet drops, by forwarding the incoming packets of flows $f1$ and $f2$ along the solid lines. Albeit the fast recovery, the re-routed $f1$ and $f2$ flows close a credit loop with the participating traffic from $f3$ and $f4$. This credit loop is depicted by the solid arrows on top the participating switches.

Several methods were proposed to deal with credit loop deadlocks. Some of these methods consider the loops when defining the forwarding rules. The most common of which is the Up*/Down* algorithm that works well with any tree-like topology [13]. According to this algorithm, credit loops are prevented if the forwarding from downstream link to upstream link is prohibited. Another set of solutions rely on multiple sets of buffers, like Ethernet traffic classes or priorities, and break the loops by changing the buffer pools used by the packets as they overlap with the start of the cycle [3]. However, these methods are all based on a central controller to calculate the forwarding rules that avoid credit loops and thus are slow to respond to failures and work only in SDN like environments.

As a measure of last resort for dealing with credit loop deadlock, Infiniband specification [1] defines Head-of-queue Lifetime Limit (HLL) mechanism. This mechanism allows switches to drop packets which are waiting in the head of the switch queue for a significant amount of time.

The contributions of this paper are as follows. We propose a distributed method for detecting and characterizing credit loop deadlock in arbitrary topology networks. The detection algorithm makes no assumptions about the routing algorithm in use. Next, we present two algorithms to resolve the credit loop deadlock. One of them, that is called *Loop Breaker*, solves the credit loop by rerouting flows. It assumes the existence of multiple paths between hosts in the network. The second algorithm, that is called *Deadlock Breaker*, given a detected loop, guarantees traffic forward progress even without multi-paths routes in the network. In order to analyze the algorithms, we define a basic scenario of network topology and a traffic pattern that assures credit loop deadlock. We describe an implementation of the algorithm over commodity switch software and analyze the algorithms using simulations in the basic scenario and in the large network scenario.

In Section 2 we survey the related work on credit loops deadlock avoidance and recovery. Next, in Section 3 we present the overview of the proposed algorithms. In Section 4 we describe our implementation of the algorithm over a real switch hardware and show simulation results over larger networks. Finally, we conclude in Section 5.

2. RELATED WORK

Networks' deadlocks is a well-studied topic in the areas of computing and manufacturing systems [18]. The existing approaches to dealing with credit loop deadlock in lossless interconnection networks can be classified into two types: avoidance and recovery. A significant research effort was invested in avoidance techniques. The most common approach is to design a central routing algorithm that considers all possible paths and avoids the formation of credit loops.

For topologies that can be described as a set of trees, and thus a clear Up direction can be defined on every link, the Up*/Down* routing [13] avoid credit loops by restricting packets routing to be first upwards and then downwards. This concept was further refined to find the minimal set of illegal turns [7]. For other topologies, like Hypercubes and Meshes it was shown that dimension ordered routing is credit loop free [3] and more generic methodologies for credit loop-free routing was presented by [11]. A totally different approach suggests virtual channels, which are mapped to isolated buffering resources within the switches, to enforce channel ordering [3, 15, 4]. A methodology named LASH [10] treats buffer pools as different layers and is able to handle arbitrary topologies.

Since using separate buffer pools can be expensive, and avoiding loops by limiting the turn space was shown to create bottlenecks in the network, *escape path* and *bubbles routing* strategies for *deadlock avoidance* schemes were pro-

posed. The *escape path* strategy [5] allows routing packet on any shortest path link, however in presence of full buffers, routing is allowed only on strictly ordered subset of paths that are proven to be credit loop-free. In torus topology, it was suggested to avoid using buffers for breaking the loop on any one-dimensional ring by means of *bubbles routing* [12]. This proposal avoids most credit loops by using dimension-order routing, all remaining loops are pre-recognized and the network switches pre-programed to allow a packet to enter the ring only if it will leave at least a single free MTU in the destination buffer. As long as at least one bubble exists on the ring, a deadlock cannot arise. We utilize a similar approach in a dynamic, topology agnostic and distributed manner in our Deadlock Breaker algorithm.

Deadlock recovery was researched to a lesser extent. Most of the work in this field was done in the context of wormhole-switched networks. The techniques mainly included finding a local heuristic that can be employed by a single switch for discovering deadlock and causing message drop as a result [9]. The concept of a set-aside buffer for resolving credit loops was described by [2] but lacks the description of detection, inclusion and eventually release of this buffer.

3. ALGORITHM DESCRIPTION

3.1 Overview

Different solutions that resolve credit loop deadlocks may need to consider the following trade-off. On the one side of the spectrum of solutions, we could design an algorithm that changes the load-balancing hash functions on the entire network when credit loop deadlock is detected. That would minimize the chance of locking on the detected credit loop again. However, that could create new credit loops on other parts of the network and cause reordering of many flows. On the other side of the spectrum, we could have an algorithm that does not change the routes at all but only temporarily reschedules the traffic on the detected credit loop. That would not prevent lock on the same credit loop again.

We present two algorithms from opposite sides of this spectrum. Both algorithms rely on detecting the credit loop deadlock as the first step, however, they differ by the method of solving the deadlock. The Loop Breaker algorithm re-routes the flows to break the loop. Hence, it assumes that multi-path routes exist between the hosts. After resolving the credit loop, the network will not lock on the same loop again. However, it cannot guarantee completely resolving the credit loop under any condition. On the other side of the spectrum, the Deadlock Breaker guarantees forward progress on the credit loop, even after the start of a deadlock occurs. But it does not break the loop itself, hence, does not prevent the same loop from reaching the same state again in the future.

The Loop Breaker algorithm is preferred for the persistent flows traffic, while the Deadlock Breaker algorithm is preferred for the short flows traffic.

For both algorithms we assume that all the network switches support the algorithm and are implementing rudimentary monitoring that is used for detecting deadlock, as detailed in the Section 4.1.

3.2 Loop Breaker

The Loop Breaker algorithm changes routing rules in the switches in order to break the loop. It is combined from following steps: (1) deadlock detection, (2) rerouting, and (3) traffic release.

The algorithm answers two major challenges. First, the switch is required to determine which pair of its {ingress,egress} ports are part of the credit loop sequence. Determining the egress port of the loop is trivial since that is the port that is blocked due to deadlock. However, determining which one of the ingress ports connects the credit loop is more complex.

Second, the credit loop deadlock blocks the traffic on all the switches on the loop almost simultaneously. The desired solution will perform minimum required routing changes. Hence, the second major challenge is distributed determination of a single switch, amongst the switches on the loop, that resolves the deadlock.

The solutions to the above challenges are presented below.

3.2.1 Deadlock detection and characterization

All switches periodically sample their ports for a deadlock. Deadlock is determined when two conditions hold on an egress port for a significant amount of time[3]: (1) zero throughput and (2) non-zero queue length. Upon these two conditions, the port is marked as *deadlock-suspected port*.

While a port is determined as deadlock-suspected, the switch periodically sends through it *Credit Loop Control Message* (CLCM) shown in 3a. These CLCMs carry a *{Switch,Port}-Unique IDentifier* (SPUID). The SPUID is randomized from a large range in order to minimize the probability of identifiers collision[4]. If several ports on the switch are deadlock-suspected, the switch sends CLCMs with different unique SPUID on each of them (unless the SPUID was replaced, as explained later). CLCM is sent with different traffic priority, hence it is not blocked by the flow control on the data priorities. Moreover, since these are one-hop control packets, their consumption does not depend on next hop buffers, and there is no problem of congestion or dropping of these packets, assuming that the switch CPU can handle them fast enough.

When CLCM packets are received by the next-hop switch their SPUIDs are compared to those sent by the switch. Next cases and reactions are considered:

- The switch does not send CLCM by itself - The switch ignores the received CLCM. That indicates failed suspicion of the deadlock port in the previous-hop switch.

[3]The specific amount of time depends on a network latency and usually can be configured to hundreds of milliseconds. The chosen value provides a trade-off between reaction time to a deadlock with false determination of a deadlock.

[4]32-bit long SPUID is enough for unique assignment in a 10,000 64-port switches network.

- Received SPUID is greater than any of the SPUIDs sent by the switch - The switch ignores the received CLCM and continues sending CLCM as previously.

- Received SPUID is lower than at least one of the SPUIDs sent by the switch - The switch replaces the SPUID on the egress ports with CLCMs with a greater value than the received SPUID, to the value of the received SPUID.

- Received SPUID is equal to SPUID that is generated by the switch - Received SPUID identifies a specific egress port in the case where several deadlock-suspected ports exist. The switch is chosen as a *master* to resolve the deadlock.

This behavior answer the two presented above challenges: The SPUID comparison provides master election process for selecting the switch that resolves the deadlock. Arriving port of the CLCM determines the ingress port of the credit loop.

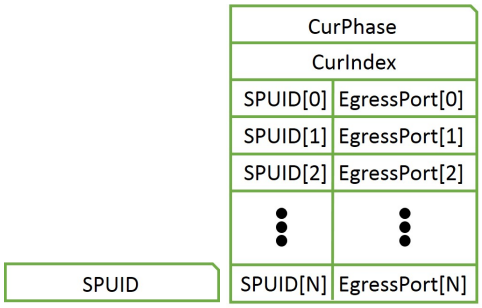
3.2.2 *Rerouting*

The selected credit loop *master* switch resolves the deadlock by rerouting flows. The rerouted flows are the flows that arrive and are forwarded on the credit loop {ingress,egress} ports. Several methods can be used in order to identify the flows that are forwarded on specific ports. One possible method is to continuously monitor the recent flows and keep a database of recent {ingress,egress} port pairs.

We assume that all the flows that are required to be rerouted have an alternative path to the destination. For each of the flows another min-distance port is chosen. However, in some topologies, specific ports on some switches do not have parallel routes to a destination. For example, it is common for fat-trees, that there is a single path to the destination for traffic flowing through any switch down towards the leaves. Hence, for some switches on such topologies the credit loop would not be resolved if these switches are chosen as a *master*. A possible solution to this case is to not initiate CLCM packet by the switch if no multiple paths exist to the destination. When any CLCM is received, it is simply forwarded to the next switch. As a result, this switch will not be selected to resolve the loop, and the loop will be resolved by another switch that has multiple paths. An alternative behavior is to reroute flows that are forwarded to the blocked port but are not arriving from the port on the loop. That will reduce the offered load on the blocked port on the loop, decrease the headroom buffer occupancy below the pause threshold, and free the deadlock.

3.2.3 *Traffic Release*

In common switch hardware architectures, packets that finished processing and enqueued to the corresponding egress queue, cannot be re-queued to another egress queue. Therefore, changing the routing rules is not sufficient to resolve the credit loop, since the ports on the credit loop are stalled and traffic cannot flow. Therefore, an additional step of releasing the traffic is required.



(a) Loop Breaker CLCM. (b) Deadlock Breaker CLCM.
Figure 3: The Creadit Loop Control Messages.

After changing the routing rules, we require the switch to transmit some amount of traffic from the port on the credit loop. That is despite the fact that the port is blocked from transmission by the flow control and these packets may be dropped by the next switch. The transmitted packets release some space on the switch buffer and allows to release the incoming ports from flow control block. The incoming traffic releases buffer space in the previous switch on the loop and so on. Eventually, the credit loop is resolved.

It is tricky to determine how much traffic the resolving switch should initially release. On the one hand, the released traffic during flow control blocking might be dropped by the next switch due to lack of sufficient buffer space. But on the other hand, at least a packet worth of traffic must be released to remove the deadlock. In our implementation, the traffic was released gradually.

In the next section, we propose an extension to the CLCM protocol that traverses the discovered loops to allocate buffers and avoid any packet drop.

3.3 Deadlock Breaker

Credit loop deadlocks are considered harmful mostly because they require packet drop to recover. Deadlock Breaker guarantees forward progress on the credit loop, even after deadlock occurs, without trying to break the loop. It assumes that loop will be broken by itself when the traffic pattern will be changed. The algorithm releases the deadlock without loss of packets and without any change to routing. It is combined from following steps: (1) deadlock detection and (2) deadlock release.

Similarly to the Loop Breaker, the Deadlock Breaker uses CLCM packets that are sent periodically through the output port of switches that are marked as *deadlock-suspected*. However, the Deadlock Breaker adds a second algorithmic phase to the previously described deadlock *detection* phase. During the new *release* phase, the CLCM packet needs to traverse again the detected loop to signal the need for additional buffers. To enable the second traversal of the exact same loop, four fields were added to the CLCM as shown in shown in 3b: The *CurPhase* field, two arrays named *SPUIDs[]* and

EgressPorts[] and an index into these arrays named *CurIndex*. When a new CLCM is create its *CurPhase* is set to *detection* and *CurIndex* is set to 0.

During the *detection* phase the following algorithm is performed when a CLCM is received by a switch: A deadlock is found when a switch finds its own SPUID in the *SPUIDs[J]* at index J. When this happens the *CurPhase* changes to *release* and the index J is written into the *CurIndex*. If the ingress port has traffic towards the *EgressPorts[CurIndex]* and that egress port is *deadlock-suspected* forward the CLCM to that port. Otherwise, drop the CLCM. If the SPUID is not included in the CLCM *SPUIDs[]* and the ingress port receiving CLCM has traffic towards any egress port that is *deadlock-suspected*, randomly select one of these egress ports and store the current switch SPUID and egress port into the *SPUIDs[CurIndex]* and *EgressPorts[CurIndex]*. Then advance the CurIndex and send the CLCM through the egress port. If no egress port found drop the CLCM.

During the *release* phase the CLCM is sent along the path defined by the series of ports from *EgressPorts* starting at *CurIndex* which is increased at every hop. When a CLCM packet with a phase set to *release* is received, the switch is able to determine the egress port by inspecting the content of *EgressPorts[CurIndex]*. It then performs two changes to its operation that are active for a configurable time period. First, it installs an arbitration rule on the egress port that only allows traffic that arrives from that ingress port to be transmitted, so no external traffic can enter the loop. Second, it increases the available buffer for the ingress port by at least a full MTU packet size, so traffic from the loop can now make progress. The available buffer of MTU on the next switch assures that packets are not dropped. The switch then advances the *CurIndex* and forward the CLCM to the next switch in the loop, until the last entry in the array is reached, and the CLCM is dropped. When the configured time period expires the arbitration rules are deleted and traffic can enter the loop again.

As stated, this algorithm does not remove the credit loop and thus it may deadlock again, but as shown in Section 4.3 progress is made and no packets are dropped. Congestion control mechanisms should sense the slow progress and compensate by lowering the bandwidth of the participating flows. Handling of multiple concurrent credit loops is left for future work.

4. EVALUATION

In this section, we describe the implementation of Loop Breaker and Deadlock Breaker algorithms. Due to the lack of space, we explain the implementation of the Loop Breaker only, and we provide large network simulation evaluation of the Deadlock Breaker only. We present a simple scenario that assures credit loop deadlock to occur and show how it is resolved by the algorithms.

4.1 Loop Breaker Implementation

We implemented the Loop Breaker algorithm using a Software Development Kit (SDK) of the Mellanox Spectrum switch. The SDK allows access to egress queue length and the outgoing bandwidth samples. We monitor these values to determine if the egress port is suspected to be blocked.

In parallel, we implemented the flow monitoring process that stores the {ingress,egress} ports for recent flows that are transmitted through the switch. Each flow is identified by the 5-tuple (source and destination IP addresses, source and destination layer ports and layer 4 protocol) since these are the fields that are used by ECMP hashing function. A preferable alternative implementation is to rely on elephant flows detection mechanism if it is available by the switch hardware.

As described in Section 3.2 when a new deadlock-suspected port is detected, new CLCM packet with random SPUID is created and sent by the switch CPU. The switch is also configured to trap on arrival of incoming CLCM packets to process in CPU, which are processed as described above.

Re-routing flows for breaking the credit loop is implemented by adding a specific rule for each flow that was monitored with the {ingress,egress} ports pair on the credit loop. The new rule for a flow excludes the deadlocked egress port from the ECMP group port list.

An alternative method to implement flow re-routing relies on fine-grain ECMP rebalancing feature available on our switch. Switch hardware implements ECMP table of 4K entries that is attached to the routing entry in the forwarding database. Each one of the entries defines a specific egress port. For a given flow the routing entry is defined by the destination IP address in the packet. Next, the header fields of the packet are mapped to one of the 4K entries using a known hash function. In our implementation of the algorithm, we looked for an ECMP entry of each of the monitored flows on the credit loop and specifically changed the egress port to randomly selected other port in the ECMP group.

Releasing the traffic on the blocked port is done by temporarily disabling flow control on a port. We use two consecutive SDK API calls to disable and then to enable back the flow control.

4.2 Loop Breaker Evaluation

We define a simple scenario with necessary conditions for credit loop deadlock to occur as presented on Figure 4. That is a simplified topology of a fat-tree network example on Figure 1. One of the switches has two hosts that are connected and injecting traffic. Each of the hosts sends a persistent non-congested controlled flow to a host on the opposite side, such that a flow is traversed over three switches on its minimal distance route. All the traffic on the ring is routed in the same direction (clockwise, in our example). The additional host on the last switch is required to break the symmetry of the scenario. Without it, the buffers would not be filled up, since in every switch there would be two incoming full rate ports which would be forwarded to two other distinct ports. The addition of an incoming port to one of the switches, makes it

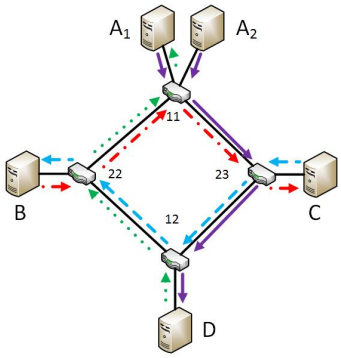
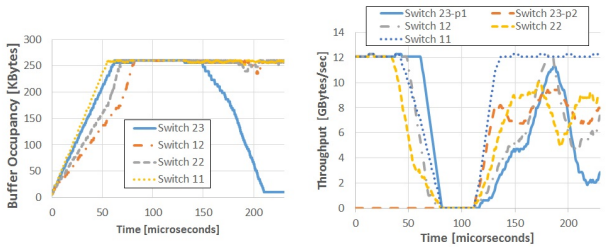


Figure 4: Simple credit loop scenario set-up. That is simplified topology of a fat-tree network example on Figure 1. The same notations for switches and hosts are used, and the flows are marked with the same patterns and colors.



(a) Buffer occupancy over time. (b) Throughput over time.

Figure 5: Simulation of a credit loop scenario of Figure 4

congested since now there are three incoming full rate ports that are forwarded to two egress ports only. The congestion is spread to the previous switch on a ring, and then to all the switches on the ring.

This scenario was tested in the lab and indeed the credit loop deadlock was resolved by the new mechanism, the measured bandwidth dropped to zero without Loop Breaker and recovered once the algorithm was run. In order to provide finer grain waveforms, we simulated the above scenario using OMNeT++ based infrastructure [16]. Figure 5 shows the buffer occupancy and the throughput of ports on the ring over time, respectively. At a time of 80 μsec the loop locks and the algorithm starts to operate on a switch that was chosen as a master and the new incoming data of flows is rerouted to another link. Switch 23 is selected as a master and reroutes the flows between the ports at the time of 120 μsec. The loop is broken and traffic released from a deadlock. The buffer occupancy of switch 23 decreases, since now it has two outgoing links used on the ring. The time span between the deadlock to occur until it resolution depends on the propagation times of the links on the ring and on the processing latency of CLCMs in switch CPU.

4.3 Deadlock Breaker Evaluation

The Deadlock Breaker algorithm was implemented in the network simulator only. We tested the algorithm on a 3-level fat-tree presented in Figure 6. Wiring mistakes, drawn by dashed lines, makes this fat-tree prone to credit loops even

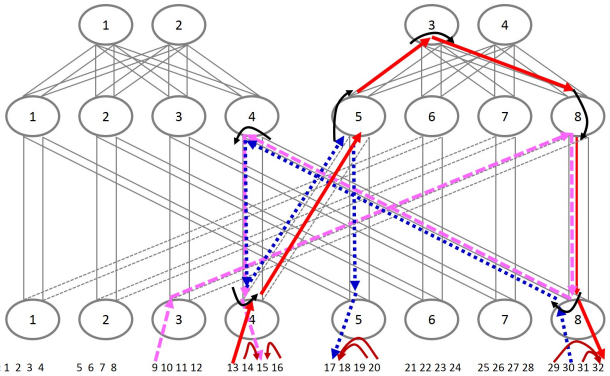
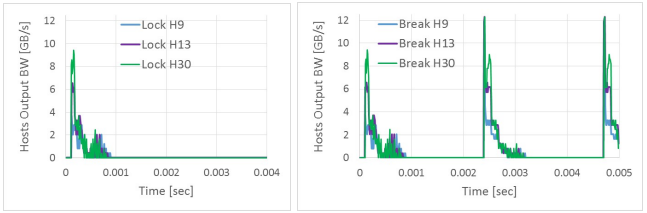


Figure 6: Credit loops created by minimal hop forwarding on 3 level fat-tree due to cable connection irregularity.



(a) The credit loop deadlock. (b) Deadlock Breaker allows progress.

Figure 7: Deadlock Breaker run on Figure 6 fat tree.

when routed with minimal hop routing algorithm. To cause a deadlock we introduce nine flows: $9 \rightarrow 15, 13 \rightarrow 32, 30 \rightarrow 17$ are forming the credit loop and $14 \rightarrow 15, 16 \rightarrow 15, 18 \rightarrow 17, 20 \rightarrow 17, 29 \rightarrow 32, 31 \rightarrow 32$ are creating the back-pressure required to fill up the loop buffers.

We simulated the network with and without our deadlock breaking algorithm and measure the output bandwidth of the flows on the loop. As presented in Figure 7a the bandwidth of the flows on the loop drops fast and stay at zero since the loop deadlocks. By enabling the Loop Breaker algorithm the loop flows are making temporary progress shown in Figure 7b.

5. SUMMARY

In this paper we addressed a credit loop deadlock phenomena that occurs in lossless networks. We presented two distributed algorithms to solve the deadlock and discussed their trade-offs. The algorithms can be implemented in Ethernet networks without central controller. We presented our implementation using switch software. We defined simple scenario for credit loop deadlock to occur and evaluated our solution on it using real implementation and simulations. In addition, we extended the simulation analysis over the large network. Our future work includes analytical analysis of the credit loop phenomena to determine the required conditions and the probability of dead lock to occur. In addition, in future work we aim to find how credit loop deadlock can be handled using congestion control and adaptive routing algorithms.

6. REFERENCES

[1] InfiniBand Trade Association et al., InfiniBand architecture specification, Volume 1, Release 1.0, Oct. 2000.

[2] K. Anjan and T. M. Pinkston. An efficient, fully adaptive deadlock recovery scheme: Disha. In *ACM SIGARCH Computer Architecture News*, volume 23, pages 201–210. ACM, 1995.

[3] W. J. Dally and C. L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Transactions on computers*, 100(5):547–553.

[4] J. Domke, T. Hoefler, and W. E. Nagel. Deadlock-free oblivious routing for arbitrary topologies. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 616–627. IEEE, 2011.

[5] J. Duato. A new theory of deadlock-free adaptive routing in wormhole networks. *IEEE transactions on parallel and distributed systems*, 4(12):1320–1331.

[6] K.-T. Förster, R. Mahajan, and R. Wattenhofer. Consistent updates in software defined networks: On dependencies, loop freedom, and blackholes. *Proc. 15th IFIP Networking*, 2016.

[7] C. J. Glass and L. M. Ni. The turn model for adaptive routing. *ACM SIGARCH Computer Architecture News*, 20(2):278–287.

[8] V. Liu, D. Halperin, A. Krishnamurthy, and T. Anderson. F10: A fault-tolerant engineered network. In *NSDI 2013*, pages 399–412, 2013.

[9] P. Lopez, J. M. Martinez, and J. Duato. A very efficient distributed deadlock detection mechanism for wormhole networks. In *High-Performance Computer Architecture, 1998*, pages 57–66.

[10] O. Lysne, T. Skeie, S.-A. Reinemo, and I. Theiss. Layered routing in irregular networks. *IEEE Transactions on Parallel and Distributed Systems*, 17(1):51–65, 2006.

[11] H. Park and D. P. Agrawal. Generic methodologies for deadlock-free routing. In *Parallel Processing Symposium, 1996., Proceedings of IPPS'96, The 10th International*, pages 638–643. IEEE, 1996.

[12] V. Puente, R. Beivide, J. A. Gregorio, J. M. Prellezo, J. Duato, and C. Izu. Adaptive bubble router: a design to improve performance in torus networks. In *Parallel Processing, 1999.*, pages 58–67. IEEE.

[13] M. D. Schroeder, A. D. Birrell, M. Burrows, H. Murray, R. M. Needham, T. L. Rodeheffer, E. H. Satterthwaite, and C. P. Thacker. *Autonet: a high-speed, self-configuring local area network using point-to-point links*. Digital Equipment Corporation Systems Research Center.

[14] S. Toueg and K. Steiglitz. Some complexity results in the design of deadlock-free packet switching networks. *SIAM Journal on Computing*, 10(4):702–712.

[15] K. D. Underwood and E. Borch. A unified algorithm for both randomized deterministic and adaptive routing in torus networks. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 723–732. IEEE, 2011.

[16] A. Varga. Omnet++ (http://www.omnetpp.org/), 2004.

[17] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz. Detail: reducing the flow completion time tail in datacenter networks. *ACM SIGCOMM Computer Communication Review*, 42(4):139–150, 2012.

[18] M. Zhou and M. P. Fanti. *Deadlock resolution in computer-integrated systems*. CRC Press, 2004.