Fast and Cautious: Leveraging Multi-path Diversity for Transport Loss Recovery in Data Centers

Guo Chen[1,2], Yuanwei Lu[3,2], Yuan Meng[1], Bojie Li[3,2], Kun Tan[2], Dan Pei[1*], Peng Cheng[2], Layong (Larry) Luo[2], Yongqiang Xiong[2], Xiaoliang Wang[4], and Youjian Zhao[1]

[1]*Tsinghua National Laboratory for Information Science and Technology, Tsinghua University,* [2]*Microsoft Research,* [3]*University of Science & Technology of China,* [4]*Nanjing University*

Abstract

To achieve low TCP flow completion time (FCT) in data center networks (DCNs), it is critical and challenging to rapidly recover loss without adding extra congestion. Therefore, in this paper we propose a novel loss recovery approach FUSO that exploits multi-path diversity in DCN for transport loss recovery. In FUSO, when a multi-path transport sender suspects loss on one sub-flow, recovery packets are immediately sent over another sub-flow that is not or less lossy *and* has spare congestion window slots. FUSO is *fast* in that it does not need to wait for timeout on the lossy sub-flow, and it is *cautious* in that it does not violate congestion control algorithm. Testbed experiments and simulations show that FUSO decreases the latency-sensitive flows' 99^{th} percentile FCT by up to ∼82.3% in a 1Gbps testbed, and up to ∼87.9% in a 10Gpbs large-scale simulated network.

1 Introduction

In recent years, large data centers have been built at an unforeseen rate and scale worldwide. Each data center may contain 100K servers, interconnected together by a large data center network (DCN) consisting of thousands of network equipments *e.g.*, switches and links. Modern applications hosted in DCN care much about the tail flow completion time (FCT) (*e.g.*, 99^{th} percentile). For example, in response to a user request, a web application (*e.g.*, Bing, Google, Facebook) often touches computation or memory resources of hundreds of machines, generating a large number of parallel latency-sensitive flows within the DCN. The overall application performance is commonly governed by the last completed flows [1, 2]. Therefore, the application performance will be greatly impaired if the network is *lossy*, as the tail FCT of TCP flows may greatly suffer from retransmission timeouts (RTO) [3, 4] under lossy condition.

Unluckily, packet losses are not uncommon even in well-engineered modern datacenter networks (§2.1). Conventionally, most of packet losses are due to buffer overflow caused by congestion, *e.g.*, incast [5, 6]. However, with the increasing deployment of the Explicit Congestion Notification (ECN) and fine-tuned TCP congestion control algorithm (*e.g.*, [1, 7]), the network congestion has been greatly mitigated (*e.g.*, from 1% to 0.01% [6]). But it still cannot be eliminated [7, 8]. Besides congestion, packets may also get lost due to failure (*e.g.*, malfunctioning hardware [3]). While normally hardware-induced loss rate is low (∼0.001%) [3], the rate can exceed 1% when hardware does not function properly. The reason for malfunctioning hardware is complex. It can come from ASIC deficits, or simply due to aging. Although the overall instances of malfunctioning hardware are small, once it happens, it usually takes hours or days to detect and mitigate [3].

We show, both analytically and experimentally, that even a moderate rise of loss rate (*e.g.*, to 1%) can already cause more than 1% of flows to hit RTOs (§2), and therefore greatly increases the 99^{th} percentile of flow FCT. Thus, we need a more robust transport that can ensure low tail FCT even when facing this adverse situation with lossy hardware. Previously, several techniques have been proposed to reduce TCP RTOs by adding more aggressiveness in loss recovery [4]. These schemes, originally designed for the Internet, have not been well tested in a DCN environment, where congestion may be highly correlated, *i.e.*, incast. Therefore, they are facing a difficult dilemma: if being too aggressive, this additional aggressiveness may offset the effect of the fine-tuned congestion control algorithm for DCN and induce congestion losses; Otherwise, being too timid would still cause delayed tail FCT.

In this paper, we advocate to utilize *multiple parallel paths*, which are plenty in most existing DCN topologies [6, 9–12], *to perform faster loss recovery*, without adding more congestion. To this end, we present *F*ast M*u*lti-path Lo*s*s Rec*o*very (FUSO), which employs multiple distinct paths for data transmission (similar to MPTCP [13–15]). FUSO fundamentally avoids the aforementioned dilemma of single-path TCP enhancements [4]. On one hand, FUSO strictly follows TCP congestion control algorithm which is well tuned for existing DCN. That is, a packet can leave the sender only when the TCP congestion window allows. Therefore, FUSO will behave equally aggressively as TCP flows (or precisely MPTCP flows). On the other hand, FUSO sender

*Dan Pei is the corresponding author.

will proactively (immediately) recover potential packet loss in a few paths (usually the "bad" paths) using other paths (usually the "good" paths). By exploiting the diversity of these paths, FUSO can keep the tail FCT low even with malfunctioning hardware. This behavior is fundamentally different from MPTCP, where each sub-flow is normally responsible to only recover its own losses. Although MPTCP provides an excellent performance for long flows' throughput, it may actually hurt the tail FCT of small flows compared to normal TCP (more discussion in §2.4).

Particularly, FUSO conducts proactive multi-path loss recovery as follows. When a sub-flow has no more new data to send, FUSO tries to utilize this sub-flow's spare resources permitted by transport congestion control to do proactive loss recovery on another sub-flow. FUSO speculates a path status from the information already recorded in the transport stack (*e.g.*, packet retransmission). Then it proactively transmits recovery packets through those good paths, to protect those packets suspected to be lost in the bad paths. By doing this, there is no need to wait for bad paths to recover loss by themselves which may cost a rather long time (*e.g.*, rely on timeout). Note that, because FUSO adds no aggressiveness to congestion control, even when loss happens at the edge (*e.g.*, incast) where no path diversity could be utilized, FUSO can still gracefully bound the redundancy incurred by proactive loss recovery, and offer a good performance (§5.2.3). The major contributions of the paper are summarized as follows.

1) We measure the attributes of packets loss in a Microsoft's production DCN. Then, through analysis and testbed experiments, we quantify the impact of packet loss on TCP FCT in DCN for the first time. We show that even a moderate rise of loss rate (*e.g.*, to 1%) would already cause enough flows (*e.g.*, $>$1%) to timeout to affect the 99th percentile FCT.

2) We identify that the fundamental challenge for transport loss recovery in DCN is how to accelerate loss recovery under various loss conditions without causing congestion. We further show that existing loss recovery solutions differ just in their *fixed* choices of aggressiveness when dealing with the above challenge, and are not adaptive enough to deal with different loss conditions.

3) We design a novel loss transport recovery approach that exploits multi-path diversity in DCN. In our proposed solution FUSO, when loss is suspected on one sub-flow, recovery packets are immediately sent over another sub-flow that is speculated to be not or less lossy *and* has a spare congestion window. However, we show that, although conventional MPTCP [13–15] provides an excellent multi-path transport architecture and significantly improves the performance for long flows, it actually hurts the tail FCT for small flows.

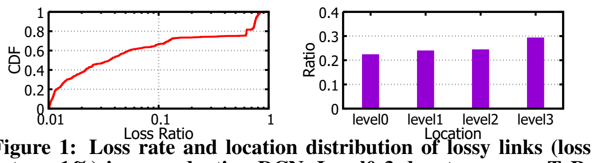


Figure 1: Loss rate and location distribution of lossy links (loss rate $>$ 1%) in a production DCN. Level0-3 denote server$\leftrightarrow$ToR, ToR$\leftrightarrow$Agg, Agg$\leftrightarrow$Spine, and Spine$\leftrightarrow$Core, respectively.

4) We implement FUSO in Linux kernel with $\sim$900 lines of code (available at `https://github.com/1989chenguo/FUSO`). Experiment results show that FUSO's dynamic speculation-based loss recovery adapts to various loss conditions well. It decreases the latency-sensitive flows' 99th percentile FCT by up to $\sim$82.3% in an 1Gbps testbed, and up to $\sim$87.9% in a 10Gpbs large-scale simulated network.

2 Fighting Against Packet Loss

2.1 Packet Loss in DCN

We first measure the attributes of packets loss in DCN, using *Netbouncer* within a Microsoft Azure's production data center. NetBouncer is a service deployed in Microsoft data centers for measuring link status. It is an end-host and switch joint solution and employs an active probing mechanism. End-hosts inject probing packets destined to network switches via IP-in-IP tunneling and switches bounce back the packets to the endhosts. It is an always-on service and the probing is done periodically. We have measured the packet loss in the data center for five days during December 1st-5th, 2015. The data center has four layers of switches, top-of-rack (ToR), Aggregation (Agg), Spine and Core from bottom to top.

Loss is not uncommon: In our operation experience, we find that although the portion of lossy links is small, they are not uncommon (also revealed in [3]). We define those links with loss rate (measured per hour) greater than 1% as *lossy links*, which may greatly impair the up-layer application performance (§2.2). Taking one day's data as an example, Fig. 1 (left part) shows the loss rate distribution among all lossy links during an hour (22:00-23:00). The mean loss rate of all the lossy links is $\sim$4%, and $\sim$63% of lossy links have the loss rate between 1% to 10%. About 22% of links even have a detected loss rate larger than 60%, where such exceptionally high loss rate maybe due to switch ASIC deficits (*e.g.*, packet blackhole [3]). We examine all the 5 days's data and find the loss rate distributions all very similar. It shows that although the portion of lossy link is small, they are the norm rather than the exception in large-scale data centers. Packet loss can be caused due to various reasons including failures and congestion.

Location of loss: Next, we analyze the location where packet loss happens. As shown in Fig. 1 (right part), among all the detected lossy links, there are only $\sim$22% of lossy links that are at the edge (server$\leftrightarrow$ToR, *i.e.*, level0), and $\sim$78% are happening in the network (above

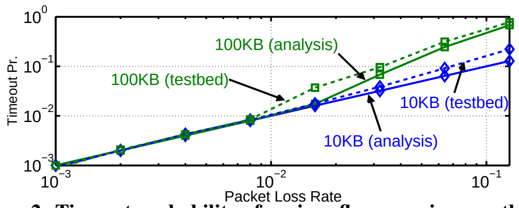


Figure 2: Timeout probability of various flows passing a path with different random packet loss rate.

ToR, *i.e.*, level1-3). About 22%, 24%, 25% and 29% of lossy links are located respectively at server↔ToR, ToR↔Agg, Agg↔Spine and Spine↔Core.

In summary, even in well-engineered modern data center networks, *packet losses are inevitable*. Although the overall loss rate is low, the packet loss rate in some areas (*e.g.*, links) can exceed several percents, when there are failures such as malfunctioning hardware or severe congestions. Moreover, *most losses happen in the network instead of the edge*.

2.2 Impact of Packet Loss

Once a packet gets lost in the network, TCP needs to recover it to provide reliable communication. There are two existing loss detection and recovery mechanisms in TCP [1]: *fast recovery* and *retransmission timeout (RTO)*. Fast recovery detects a packet loss by monitoring duplicated ACKs (or DACKs) and starts to retransmit an old packet once a certain number (*i.e.*, three) of DACKs have been received. If there are not enough DACKs, TCP has to rely on RTO and retransmits all un-ACKed packets after the timeout. To prevent premature timeouts and also limited by the kernel timer resolution, the RTO value is set rather conservatively, usually several times of the round-trip-time (RTT). Specifically, in a production data center, the minimum RTO is set to be $5ms$ [1,3] (the lowest value supported in current Linux kernel [16]), while the RTT is usually hundreds of μs [1, 3, 16]. As a consequence, for a latency-sensitive flow, which is usually small in size, encountering merely one RTO would already increase its completion time by several times and cause unacceptable performance degradation.

Therefore, the core issue in achieving low FCT for small latency-sensitive flows when facing packet losses is to avoid RTO. However, current TCP still has to rely on RTO to recover from packet loss in the following three cases [4, 17, 18]. i) The last packet or a series of consecutive packets at the tail of a TCP flow are lost (*i.e.*, *tail loss*), where the TCP sender cannot get enough DACKs to trigger fast recovery and will incur an RTO. ii) A whole window worth of packets are lost (*i.e.*, *whole window loss*). iii) A retransmitted packet also gets lost (*i.e.*, *retransmission loss*).

[1]Many production data centers also use DCTCP [1] as their network transport protocol. DCTCP has the same loss recovery scheme as TCP. Thus, for ease of presentation, we use TCP to stand for both TCP and DCTCP while discussing the loss recovery.

To understand how likely RTO may occur to a flow, we take both a simple mathematical analysis (estimated lower bound) and testbed experiments to analyze the timeout probability of a TCP flow with different flow sizes and different loss rates. We consider one-way random loss condition here for simplicity, but the impact on TCP performance and our FUSO scheme are by no means limited to this loss pattern (see §5).

Let's first assume the network path has a loss probability of p. Assuming the TCP sender needs k DACKs to trigger fast recovery, any of the last k packets getting lost will lead to an RTO. This tail loss probability is $p_{tail} = 1-(1-p)^k$. For standard TCP, $k = 3$, but recent Linux kernel which implement's *early retransmit* [19] reduces k to 1 at the end of the transaction. Therefore, if we consider early retransmit, the tail loss probability is simply p. The whole window loss probability can easily be derived as $p_{win} = p^w$, where w is the TCP window size. For retransmission loss, clearly, the probability that both the original packet and its retransmission are lost is p^2. Let x be the number of packets in a TCP flow. The probability that the flow encounters at least one retransmission loss is $p_{retx} = 1-(1-p^2)^x$. In summary, the timeout probability of the flow should be $p_{RTO} \geq max(p_{tail}, p_{win}, p_{retx})$. The solid lines in Fig. 2 show the analyzed lower bound timeout probability of a TCP flow with different flow sizes under various loss rates. Here, we consider the early retransmit ($k = 1$).

To verify our analysis, we also conduct a testbed experiment to generate TCP flows between two servers. All flows pass through a path with one-way random loss. *Netem* [20, 21] is used to generate different loss rate on the path. More details about the testbed settings can be found in §4.2 and §5. The dotted lines in Fig. 2 shows the testbed results, which verify that our analysis serves as a good lower bound of the timeout probability.

There are a few observations. Firstly, for tiny flows (*e.g.*, 10KB), the timeout probability linearly grows with the random loss rate. This is because the tail loss probability dominates. However, a tiny loss probability would affect the tail of FCT. For example, a moderate rise of the probability to 1% would cause a timeout probability larger than 1%, which means the 99th percentile of FCT would be greatly impacted. Secondly, when the flow size increases, *e.g.*, ≥100KB, the retransmission loss may dominate, especially when the random hardware loss rate is larger than 1%. We can see a clear rise in timeout probabilities for the flows with 100KB in Fig. 2. In summary, we conclude that *a small random loss rate (i.e., >1%) would already cause enough flows to timeout to affect the 99th percentile of FCT*. This can also explain why a malfunctioning switch in the Azure datacenter that drops ∼2% of the packets causes great performance degradation of all the services that traverse this switch [3].

2.3 Challenge for TCP Loss Recovery

To prevent timeout, when there are not enough returned DACKs to trigger fast recovery, prior work (*e.g.*, [4]) adds aggressiveness to congestion control to do loss recovery before RTO. However, deciding the *aggressiveness level*, *i.e.*, how long to wait before sending recovery packets, to adapt to complex network conditions in DCNs is a daunting task.

As introduced before, congestion and failure loss coexist in DCN. Congestion losses are very bursty and often lead to multiple consecutive packet losses [1,4,5,7]. For congestion loss, recovery should be delayed for enough time before being sent out after the original packets. If a recovery packet is sent too fast before congestion disappears, the recovery packet may get dropped by the overflowed buffer and also worsen the congestion. However, for some failure loss such as random drop, recovery packets should be sent as fast as possible to accelerate the recovery process. Otherwise the delay for sending recovery packets already increases the FCT of latency-sensitive flows. Facing this difficult dilemma, previous schemes choose different *aggressiveness levels* in an ad-hoc manner, from a conservative 2RTT in Tail Loss Probe (TLP) [22], modestly conservative 1/4 RTT in TCP Instant Recovery (TCP-IR) [23], to a very aggressive zero time in Proactive [4]. Unfortunately, the fixed settings of aggressiveness levels make above existing schemes incapable of adapting to complex loss conditions: different loss characteristics under either congestion loss, failure loss or both.

Essentially, we identify that the fundamental challenge for transport loss recovery in DCN is *how to accelerate loss recovery as soon as possible, under various loss conditions without causing congestion*. Single-path loss recovery is not a promising direction to address this challenge because the recovery packets have to be sent over the same path that is under various loss conditions, the exact nature (congestion-induced, failure-induced, or both) of which are often unclear to the sender. One might think that through explicitly identifying congestion loss using schemes such as CP [24], transport can distinguish congestion and failure loss with the help of switches. However, there lacks a study on such design and its reliability under hardware failure conditions still remains to be an open question in complex production DCNs.

2.4 Utilizing Multi-path

Then it is natural to raise a question: why not try another good path when loss is speculated on one "bad" path? Actually, current DCN environment offers us a good chance to design a better loss recovery scheme based on multi-path. Current DCN provides many parallel paths (*e.g.*, 64 or more) between any two nodes by dense interconnected topologies [6,9–12]. Usually, these paths have a big loss diversity due to different congestion and failure conditions. When a few paths are experiencing failure such as random loss or black-hole, the rest paths (*i.e.*, the majority) may remain in a good state without failure loss. Also, caused by uneven load balance [25], some paths may be heavily congested to drop packets while other paths are in light load.

One might think that using multi-path transport protocol such as MPTCP [13–15] is able to address the challenge above. On the contrary, although MPTCP provides excellent performance for long flows, it actually hurts the tail FCT of small latency-sensitive flows under lossy condition (see §5). It is because that, while MPTCP explores multiple paths, each of its paths normally has to recover loss by itself. Therefore, its overall completion time depends on the last completed sub-flow on the worst path. Simply exploring multiple paths actually increases the chance to hit the bad paths. Therefore, MPTCP's lack of an effective loss recovery mechanism leads to a long tail FCT especially for small flows.

To this end, we propose *F*ast M*u*lti-path Lo*s*s Rec*o*very (FUSO), which leverages multi-path diversity for transport loss recovery. FUSO fundamentally avoids the aforementioned dilemma (§2.3), by utilizing those paths in good status to proactively (or immediately) conduct loss recovery for bad paths. First, FUSO is *cautious* in that it strictly follows TCP congestion control algorithm that is tuned for existing DCN, adding no aggressiveness. Second, FUSO is *fast* in that the sender will proactively recover potential packet loss in bad paths using good paths before timeout. As shown before, most losses happen in the network (§2.1), which gives plenty of opportunities for FUSO to leverage multi-path diversity. On the other hand, sometimes packet losses may happen at the edge (*e.g.*, incast) due to congestion, where there is no path diversity that can be utilized for multi-path loss recovery. Thanks to strictly following the congestion control, FUSO can adaptively throttle its proactive loss recovery behaviour and be conservative to avoid worsening the congestion (see §5.2.3).

Note that there is a mechanism named opportunistic retransmission [14] in MPTCP, which may also trigger proactive retransmission through alternative good subflows similar to the scheme in our FUSO solution. Although sharing the same high-level idea which is utilizing path diversity, it addresses different problems from FUSO. MPTCP opportunistic retransmission is designed for wide-area network (WAN) to maintain a high throughput and minimize the memory (receive or send buffer) usage, to cope with severe reordering caused by diverse delay of multiple paths. It is triggered only when the new data cannot be sent because the receive window or the send buffer is full. It immediately retransmits the oldest un-ACKed packet through alternative good paths which have the smallest RTT. Although opportunistic re-

Algorithm 1 Proactive multi-path loss recovery.

```
1: function TRY_SEND_RECOVERIES( )
2:     while $BytesInFlight_{Total} < CWND_{Total}$ and no new data do
3:         return ← SEND_A_RECOVERY( )
4:         if return == NOT_SEND then
5:             break
1: function SEND_A_RECOVERY( )
2:     FIND_WORST_SUB-FLOW( )
3:     FIND_BEST_SUB-FLOW( )
4:     if no worst found or no best sub-flow found then
5:         return NOT_SEND
6:     recovery_packet←one un-ACKed packet of the worst sub-flow
7:     Send the recovery_packet through the best sub-flow
8:     $BytesInFlight_{Total}$ += $Size_{recovery\_packet}$
```

transmission helps to achieve a high throughput for long flows in WAN scenario, it offers little help on maintaining a low FCT under lossy condition in DCN scenario where paths often have very similar delay. More importantly, in DCN those latency-sensitive flows are often with too small sizes (*e.g.*, <100KB) to cause severe reordering, which cannot eat up the end-host's buffer. Therefore, these small flows cannot trigger the opportunistic retransmission.

3 FUSO Design

3.1 Overview

We now introduce FUSO. FUSO is built on top of the multi-path transport, in which a TCP flow is divided into multiple sub-flows. Note that FUSO focuses on multi-path loss recovery rather than multi-path congestion control. Particularly, in this paper, we build FUSO on MPTCP[2] [13–15]. ECMP [26] or SDN methods (*e.g.*, XPath [27]) can be used to implicitly or explicitly map the sub-flows[3] onto different physical paths in DCN.

The core scheme of FUSO is that, by strictly following the congestion control, if there is a spare congestion window (*cwnd*), FUSO first tries to transmit new data. If the up-layer application currently has no new data, FUSO utilizes this transmission opportunity to proactively/immediately transmit recovery packets for those suspected lost (un-ACKed[4]) packets on "bad" sub-flows, by utilizing "good" sub-flows. Note that FUSO does not affect the existing MPTCP opportunistic retransmission mechanism triggered by full receive window. These two mechanisms can be complementary to each other.

We separately discuss the FUSO sender and receiver for better clarification. In a FUSO connection, the sender and receiver refer to the end hosts sending data and the ACK respectively. Both ends are simultaneously the sender and receiver in a two-way connection.

[2]FUSO can also work on other multi-path transport protocols.

[3]We use 'sub-flow' and 'path' interchangeably in this Section.

[4]For TCP with SACK [28] enabled, un-ACKed packets refer to those un-SACKed and un-ACKed ones.

3.2 FUSO Sender

The FUSO sender's proactive multi-path loss recovery process can be summarized as Algo. 1. Specifically, we insert a function *TRY_SEND_RECOVERIES()* in the transport stack, monitoring the changes of $BytesInFlight_{Total}$, $CWND_{Total}$ and the application data. This function needs to be inserted into two positions: i) after all the data delivered from the application has been pushed into the transport send buffer and sent out, which indicates that there is currently no more new data delivered from the up-layer application; ii) after an ACK is received and the transport status (*e.g.*, $BytesInFlight_{Total}$, $CWND_{Total}$) has been changed. More implementation-related details are discussed in §4.1. Within this function, the sender calls the function *SEND_A_RECOVERY()* to send a recovery packet if the following conditions are both satisfied: i) there is spare window capacity allowed by congestion control, *and* ii) all new data has been sent out.

In the function *SEND_A_RECOVERY()*, FUSO sender first calls the function *FIND_WORST_SUB-FLOW()* and *FIND_BEST_SUB-FLOW()* to find the current worst and best sub-flows. The worst sub-flow is selected only among those who have *un-ACKed data*, and the best sub-flow is selected only among those whose *congestion window (cwnd) has spare spaces* permitted by congestion control. We defer the discussion on how to find the worst and best paths to §3.2.1.

If currently there is no worst or no best sub-flow, FUSO stops generating recovery packets for this round. Next, if the worst *and* best sub-flows are found, a recovery packet for the worst sub-flow is generated. Because FUSO conducts proactive loss recovery before a packet is detected as lost either by DACKs or RTO, we have to *guess* which packet is most likely to be the lost one. FUSO infers the packet as the oldest un-ACKed packet which has been sent out for the longest time. Thus, the sender proactively generates a recovery packet for one un-ACKed packet on the worst path in the ascending order of TCP sequence number (*i.e.*, the oldest packet in this path). To avoid adding too much unnecessary traffic to the network, an un-ACKed packet will be sent at most once by the proactive loss recovery scheme in FUSO.

After the recovery packet is generated, FUSO sender sends it through the best sub-flow. Note that the recovery packet is regarded as a *new data packet* for the best sub-flow. The recovery packet is under the best sub-flow's congestion control, and, if it gets lost in the best sub-flow, it will be retransmitted as normal packets in the best sub-flow using the standard TCP loss recovery. However, to avoid duplicate recovery, these packets will not be counted in the un-ACKed packets waiting for recovery when FUSO sender conducts fast multi-path loss recovery later. In the last step of *SEND_A_RECOVERY()*, $BytesInFlight_{Total}$ is incremented and the conditions in the while loop in *TRY_SEND_RECOVERIES()* will be checked again.

3.2.1 Path Selection

Whenever congestion control offers a chance to transmit packets, FUSO tries to proactively recover the suspected lost packet in the currently “worst” path which is most likely to encounter packet loss, utilizing the currently “best” path which is least likely to encounter packet loss. Therefore, we define a metric $C_l = \alpha \cdot \overline{lossrate} + \beta \cdot lossrate_{last}$, to describe the possibility of packet loss happening in a sub-flow. C_l is the weighted sum of the overall packet loss rate $\overline{lossrate}$ and the most recent packet loss rate $lossrate_{last}$ in this sub-flow. α and β are the respective weight of each part. Since current TCP/MPTCP retransmits a packet after detecting it as lost either by DACK or RTO, FUSO uses the ratio of total retransmitted packets *to* the total transmitted packets as the approximation of $\overline{lossrate}$. Note that recovery packets generated by FUSO are regarded as new packets instead of retransmitted packets for sub-flows. $lossrate_{last}$ is calculated as the ratio of one *to* the number of transmitted packets from (including) the last retransmission.

The worst sub-flow is picked among those which have at least one un-ACKed packet (possibly lost), and with the *largest* C_l. For sub-flows which have never encountered a retransmission yet, their C_l equals zero. If all sub-flows' C_l equals zero, FUSO picks the one with the largest measured RTT thus to optimize the overall FCT.

The best sub-flow is picked among those which have spare *cwnd*, and with the *smallest* C_l. For sub-flows never encountering a retransmission yet, their C_l equals zero and is smaller than others. If more than one sub-flows have zero C_l, FUSO picks the one with the smallest measured RTT as the best sub-flow. Note that at the initial state, some sub-flows may have never transmitted any data when FUSO starts proactive loss recovery. Then these sub-flows' C_l equal infinity and have the least priority when FUSO selects the best sub-flow. If all sub-flows' C_l equal infinity, FUSO randomly picks one as the best sub-flow. Note that the best and worst sub-flows may be the same one. Under this condition, FUSO simply transmits the recovery packets in the same sub-flow after the original packets.

3.3 FUSO Receiver

FUSO receiver is relatively simple. In multi-path transport protocol such as MPTCP, the receiver has a data-level (*i.e.*, flow-level) receive buffer and each sub-flow has a virtual receive buffer that is mapped to the data-level receive buffer. Upon receiving a FUSO recovery packet, FUSO receiver directly inserts the recovery packet into the corresponding position of the data-level receive buffer, to complete the flow transmission. The FUSO recovery packets will not affect the bad sub-flows' behaviour on the sub-flow-level, but directly recovery the lost packets on the data-level.

For the best sub-flow that transmits FUSO recovery packets, these packets are naturally regarded as normal data packets in terms of this sub-flow's congestion control and original TCP loss recovery. Although protected by them, the bad sub-flow is not aware of these recovery packets, and may unnecessarily retransmit the old data packet (if lost) itself. FUSO currently chooses such a simple approach to maintain the exact congestion control behavior and add no aggressiveness, both on individual sub-flows and the overall multi-path transport flow. It needs no further coordination besides the original ACK schemes in TCP between the sender and the receiver, but may incur some redundant retransmissions. A naive solution to eliminate the redundant retransmissions may be that the receiver proactively generates ACKs for the original packets in the bad sub-flow, upon receiving recovery packets from other good sub-flows. However, this may cause adverse interaction with congestion control. Specifically, the bad sub-flow's sender end may wrongly judge the path as in a good status and increases its sending rate, which may exacerbate the loss.

In order to maintain the congestion control behavior *and* eliminate the redundant retransmissions, it may need very complex changes to the MCTCP/TCP protocols. The sender and receiver must coordinate to decide whether/how it should change each sub-flow's congestion control behavior (*e.g.*, increase/decrease how much to the *cwnd*), to cope with various conditions, such as i) the proactive retransmission received but the original packet lost, ii) the original packet received but the proactive retransmission lost, iii) both packets lost, iv) the proactive retransmission received before the original packet, v) the original packet received before the proactive retransmission, *etc.* The feasibility of such a solution and how to design it still requires further study and is left as our future work. FUSO currently chooses to trade a little redundant retransmission (see §5.2.2 and 5.2.3) for the aforementioned simple and low-cost approach.

4 Implementation and Testbed Setup

4.1 FUSO Implementation

We implemented FUSO in Linux kernel 3.18 with 827 lines of code, building FUSO upon MPTCP's latest Linux implementation (v0.90) [29].

FUSO sender: We insert *TRY_SEND_RECOVERIES()* in Algo. 1 into the following positions of the sender's transport kernel, to check whether a FUSO recovery packet should be sent now: 1) in function *tcp_sendmsg()* after all the data delivered from the application has been pushed into the send buffer; 2) in function *tcp_v4_rcv()* after an ACK is received and the transport status (*cwnd*, bytes in flight *etc.*) has been changed.

In *TRY_SEND_RECOVERIES()*, FUSO detects that there is currently no more new data from the up-layer application, if the two conditions are both satisfied: i) the data delivered from the application has all been pushed in the send buffer; ii) the packets in the send buffer have all been sent. If a multi-path loss recovery packet is allowed to be sent, FUSO sender calls the function *SEND_A_RECOVERY()* and picks one un-ACKed packets (in ascending order of sequence number) on the worst sub-flow, then copies and transmits it on the best sub-flow. We utilize existing functions in MPTCP to reconstruct the mapping of the recovery packet's data-level (*i.e.*, flow-level) sequence number to the new sub-flow-level sequence number. Also, FUSO sender remembers this packet to ensure that each un-ACKed packet is protected for at most once. In FUSO, both the formats of data packets and FUSO recovery packets have no difference from those in the original MPTCP protocol. The data-level sequence number (DSN) in the existing Data Sequence Signal (DSS) option of MPTCP header can notify the receiver how to map this recovery packet into data-level data.

It is noteworthy that, besides the opportunistic retransmission introduced before, original MPTCP may also retransmit the data packets originally delivered to one sub-flow through other sub-flows under the following condition: if one sub-flow is judged to be dead when it encounters certain number of consecutive timeouts, all the packets once distributed to this sub-flow will be re-injected to a special flow-level sending buffer called "reinject queue". Then MPTCP will redistribute these packets to other sub-flows. This is a failover scheme to deal with the case that some of its sub-flows completely fail. However, it is too slow (after a sub-flow is dead) to provide a low FCT under lossy conditions.

FUSO receiver: The receiving process has already been implemented in MPTCP's original receiving logic, which requires no other modification. According to the DSN in the header option, the receiver will insert the multi-path loss recovery packet in the corresponding position of the data-level receive buffer, and complete the data transmission. Note that in the current MPTCP's Linux implementation, the receiver only hands over packets to the data-level receive buffer which are in-sequence in the sub-flow level, and buffers the packets which are out-of-sequence (OoS) in the sub-flow level in the sub-flow's OoS queue. This implementation reduces the reordering computation overhead, but may severely defer the completion time of the overall MPTCP flow. Since packets may be retransmitted by other sub-flows, those packets OoS in sub-flow level may be in-sequence in the data level. As such, in-sequence data-level packets may not be inserted to the data-level receive buffer even when they arrive at the receiver, because of being deferred by the former lost packets in the same sub-flow. To solve this problem, we implement a minor modification to current MPTCP's receiving end implementation, which immediately copies the sub-flow-level OoS packets directly to the MPTCP data-level receive buffer. This receiving end modification is 34 lines of code.

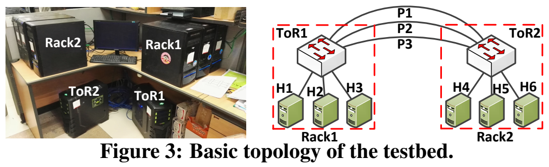


Figure 3: Basic topology of the testbed.

4.2 Testbed Setup

We build a small 1Gbps testbed as shown in Fig. 3. It consists of two 6-port ToR switches (*ToR1, ToR2*) and six hosts (*H1* ∼*H6*) located in the two racks below the ToR switches. There are three parallel paths between the two racks, emulating the multi-path DCN environment.

Each host is a desktop with an Intel E7300 Core2 Duo 2.66GHz CPU, 4GB RAM and 1Gbps NIC, and runs Ubuntu 14.04 64-bit with Linux 3.18.20 kernel. We use two servers to emulate the two ToR switches. Each server-emulated switch is a desktop with an Intel Core i7-4790 3.60GHz CPU, 32GB RAM, and 7 Intel I350 Gigabit Ethernet NICs (one reserved for the management). All server-emulated switches run Ubuntu 14.04 64-bit with Linux 4.4-RC7 kernel. Originally, current Linux kernel only support IP-address-based (<*src,dst*> pair) ECMP [26] when forwarding packets. Therefore, we made a minor modification (8 lines of code) to the switches kernel, thus to enable layer-4 port-based ECMP [26] (<*src,dst,sport,dport,protocol*> pair) which is widely supported by commodity switches and used in production DCNs [3, 6, 16].

Flows are randomly hashed to the physical paths by ECMP in our testbed. Each switch port buffer size is 128KB. The basic RTT in our testbed is ∼280μs. ECN is enabled using Linux *qdisc RED* module, with marking threshold set to be 32KB according to the guidance by [7]. We set TCP minRTO to 5*ms* [1, 3]. These settings are used in all the testbed experiments .

5 Evaluation

In this section, we use both testbed experiments and ns-2 simulations to show the following key points. 1) Our testbed experiments show ***FUSO's good performance under various lossy conditions, including failure loss, failure & congestion loss, and congestion loss.*** 2) We also use targeted testbed experiments to ***analyze the impact of sub-flow number on FUSO's performance.*** 3) Our detailed packet-level simulations confirm that ***FUSO scales to large topologies.***

5.1 Schemes Compared

We compare the following schemes with FUSO in our testbed and simulation experiments. For the simulations, we implement all the following schemes in ns-2 [30] simulator. For the testbed, we implement Proactive and Repflow [31] in Linux, and directly use the source codes of other schemes.

TCP: The standard TCP acting as the baseline. We enable the latest loss recovery schemes in IETF RFCs for TCP, including SACK [28], SACK based recovery [18], Limited Transmit [32] and Early Retransmission [19]. The rest of the compared schemes are all built on this baseline TCP.

Tail Loss Probe (TLP) [22]: The latest single-path TCP enhancement scheme using prober to accelerate loss recovery. TLP transmits one more packet after 2 RTTs when no ACK is received at the end of the transaction or when the congestion window is full. This extra packet is a prober to trigger the duplicate ACKs from the receiver before timeout.

TCP Instant Recovery (TCP-IR)[5] [23]: The latest single-path TCP enhancement scheme using both prober and redundancy. It generates a coded packet for every group of packets sent in a time bin, and waits for 1/4 RTT to send it out. This coded packet protects a single packet loss in this group providing "instant recovery", and also acts like a prober as in TLP. According to the authors' recommendation [4], we set the coding timebin to be 1/2 RTT and the maximum coding block to be 16.

Proactive [4]: A single-path TCP enhancement scheme to accelerate loss recovery by duplicating every TCP data packet. We have implemented Proactive in Linux kernel 3.18.

MPTCP [15]: The state-of-the-art multi-path transport protocol. We use the latest Linux version of MPTCP implementation (v0.90 [29]), which includes the opportunistic retransmission mechanism [14].

RepFlow [31]: A simple multi-path latency improvement scheme by proactively transmitting two duplicated flows. We have implemented RepFlow in the application layer according to [31].

For all compared schemes, the initial TCP window is set to 16 [3]. Note that for FUSO and MPTCP, the initial window of each sub-flow is set to be $\frac{16}{number\ of\ subflows}$, which forms the same 16 initial window in total for a connection. Unless specified otherwise, we configure 4 sub-flows for each FUSO and MPTCP connection in the testbed experiments, which offers the best performance for both methods in various conditions. We compare the performance of FUSO/ MPTCP using different number of sub-flows in §5.2.4. FUSO's path selection parameters α, β (§3.2.1) are both set to be 0.5.

[5]TCP-IR has published its code [33] and we successfully compiled it to our testbed hosts. However, after trying various settings, we are not able to get it running on our testbed environment due to some unknown reasons. As such, we evaluate TCP-IR only in simulation experiments.

5.2 Testbed Experiments

Benchmark Traffic: Based on the code in [34], we develop a simple client-server application. Each client sends requests to some randomly chosen servers for a certain size of data, with inter arrival time obeying the Poisson process. There are two types of requests from the client, 1) latency-sensitive queries with data sizes smaller than 100KB, and 2) background requests with sizes larger than 100KB. All the requests' sizes are sampled from two real data center workloads, web-search [1] and data-mining [10]. Each client initiates 10 long-lived transport connections (5 for latency-sensitive queries, and 5 for background requests) to each server, and round-robinly distributes the requests on each connection (of their type) to the server. We generate different loads through adjusting the requests' inter arrival time. All 6 hosts run both client and server processes. We separately enable the various compared schemes to serve the connections for latency-sensitive queries, and use standard TCP for the rest of connections for background requests. Note that before all evaluations, we generate 100KB data to warmup each FUSO/MPTCP connection and wait for an idle time to reset the initial window, thus to activate all the sub-flows. We compare the request completion time[6] of those latency-sensitive queries.

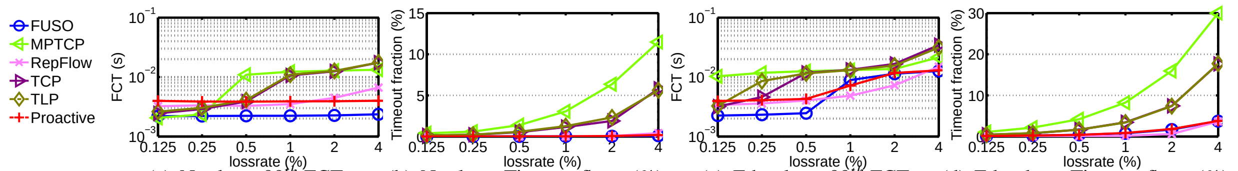
Emulating failure loss: We use *netem* [20,21] to generate failure packet loss with different loss patterns and loss rates. The network and edge loss are emulated by enabling *netem* loss module on certain network interfaces (on the switches or hosts). Two widely-used loss patterns are evaluated, random loss and bursty loss [35].

Due to space limitation, we only present the testbed results under random loss using web-search workload. We have evaluated FUSO under various settings, with different loss models (random and bursty [35]) using different workloads (web-search and data-mining), and FUSO consistently outperforms other schemes (reduce the latency-sensitive flows' 99th percentile FCT by up to ~86.2% under bursty loss and data-mining traffic). All the experiment results are from 10 runs in total, with 15K flows generated in each run.

5.2.1 Failure Loss

We first show how FUSO can gracefully handle failure loss in DCN. To avoid the interference of congestion, no background traffic is injected, and we deploy the clients on $H4$-$H6$ generating small latency-sensitive requests (data size $<$ 100KB) respectively to $H1$-$H3$ without edge contention. We only focus on the failure loss in this experiment, and later we will show how FUSO performs when failure and congestion coexist. The requests are generated in an average load of 10Mbps [1].

[6]We use 'flow' and 'request', 'request completion time' and 'FCT' interchangeably in §5.

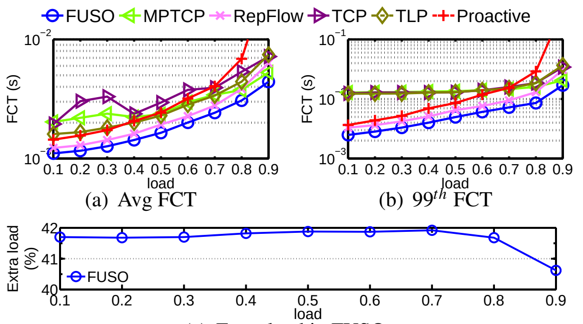


(a) Net-loss: 99th FCT (b) Net-loss: Timeout flows (%) (c) Edge-loss: 99th FCT (d) Edge-loss: Timeout flows (%)

Figure 4: Failure loss in the network *or* at the edge: 99th FCT (log scale) and timeout fraction of the latency-sensitive flows, while one network path is lossy *or* all the edge paths are lossy. Path loss rate varies from 0.125% to 4%.

Loss in the network: We first evaluate the condition when failure loss happens in the network, by deliberately generating different loss rate for the path $P1$ in Fig. 3. Note that the two directions of $P1$ both have the same loss rate. Fig. 4(a) and Fig. 4(b) present the 99th percentile FCT and the fraction of timeout ones among all the latency-sensitive flows. The results show that FUSO maintains both very low 99th percentile FCT (<2.4ms) and fraction of timeout flows (<0.096%), as the path loss rate varies from 0.125% to 4%. FUSO reduces the 99th percentile FCT by up to ∼82.3%, and the timeout fraction up to 100% (no timeout occurs in FUSO), compared to other schemes. The improvement is due to the multipath loss recovery mechanisms of FUSO, which can explore and utilize good paths that are not lossy, and also makes the FCT depend on the best path explored. Although MPTCP also explores multiple paths, each of its paths normally has to recover loss by itself (more details in §2.4). Therefore, its overall completion time depends on the last completed sub-flow on the worst path. Lacking an effective loss recovery mechanism actually lengthens the tail FCT in MPTCP, as exploring multiple paths actually increases the chance to hit the bad paths. RepFlow offers a relatively better performance than other schemes by excessively duplicating every flow. However, this way of redundancy is actually less effective than FUSO, because each flow independently transmits data with no cooperative loss recovery as in FUSO. Since there is still a big chance for ECMP to hash the two duplicated flows into the same lossy path, it makes RepFlow have an ∼32%-64.5% higher 99th percentile FCT than FUSO. Proactive also behaves inferiorly, suffering from similar problems as RepFlow. TLP performs almost the same as TCP because it sacrifices the ability to prevent timeouts in order to keep low aggressiveness.

Loss at the edge: We then evaluate the extreme condition when severe failure loss happens at the edge, by deliberately generating different loss rates for all the access links of $H4$-$H6$. We try to investigate how FUSO performs when sub-flows cannot leverage diversity among different physical paths. Fig. 4(c) and Fig. 4(d) show the results. Even with all sub-flows passing the same lossy path, FUSO still can maintain a consistent low timeout fraction to be under 0.8% when the loss rate is below 1%. However, the timeout fraction of other approaches except RepFlow and Proactive exceeds 3.3% at the same loss rate. As such, FUSO reduces the 99th percentile FCT by up to ∼80.4% compared to TCP, TLP and MPTCP. When loss rate exceeds 2%, FUSO still maintains the 99th FCT under 12.7ms. Although all sub-flows traverse the same lossy path in this scenario, the chance that all four of them simultaneously hit the loss has been decreased. FUSO can leverage the sub-flow which does not encounter loss currently to help recover lost packets in the sub-flow which hits loss at this moment. Due to the excessive redundancy, RepFlow and Proactive perform the best in this scenario when loss rate is high, but hurt the 99th FCT when the loss rate is low. Later (§5.2.3) we will show that this excessive load and the non-adaptive redundancy ratio will substantially degrade the performance of latency-sensitive flows, when the congestion is caused by themselves such as in the incast [5] scenario.



(a) Avg FCT (b) 99th FCT (c) Extra load in FUSO

Figure 5: Failure & congestion loss: Average and 99th FCT (log scale) of latency-sensitive flows, and the average extra load of all FUSO flows. Each flow's extra load is calculated by *the extra bytes incurred by FUSO* divided by *the total bytes transmitted*.

5.2.2 Failure & Congestion Loss

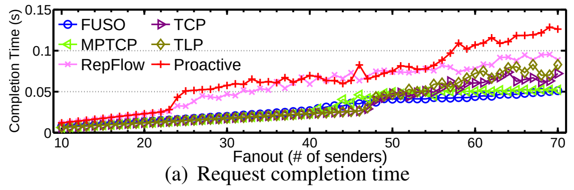
Next we evaluate that how FUSO performs with coexisting failure and congestion loss. We generate a 2% random loss rate on both directions of path $P1$, which is similar to a real Spine switch failure case in production DCN [3]. We deploy the aforementioned clients on the $H4$-$H6$ and configure them to generate small latency-sensitive queries as well as background requests, to the servers randomly chosen from $H1$-$H3$. This cross-rack traffic [13, 36] ensures that all the flows have a chance going through the lossy network path. We inject different traffic load from light (0.1) to high (0.9), to investigate how FUSO performs from failure-loss-dominated scenario to congestion-loss-dominated scenario.

Results: Fig. 5 shows the results. Under conditions where failure and congestion loss coexist, FUSO maintains both very low average and 99th percentile FCT of latency-sensitive flows, from light to high load. Compared to MPTCP, TCP and TLP, FUSO reduces the average FCT by ∼28.2%-47.1%, and the 99th percentile by ∼17.2%-80.6%. FUSO also outperforms RepFlow by ∼10%-30.3% in average and ∼20.1%-44.8% in tail, due to two reasons: 1) the chance of two replicated flows in RepFlow being hashed to the same lossy path is non-negligible, and 2) excessive redundancy RepFlow adds congestion when load is high. As for Proactive, the replicated packets always go through the same path as the original data packets, which makes them share the same loss rate and further decrease its redundancy efficiency compared to RepFlow. Moreover, the simple duplicating behaviour extremely degrades its performance under heavy load. On the contrary, FUSO's proactive multi-path loss recovery helps to recover the congestion and failure loss *fast*, meanwhile remaining *cautious* to avoid adding aggressiveness. Even at the high load of 0.9, FUSO maintains the average and tail FCT to be below 4.5ms and 17.1ms, respectively. TCP behaves inferiorly due to coexisting severe congestion and failure loss, while MPTCP performs better in this case. TLP's faster loss recovery by adding moderate aggressiveness makes it perform better than both TCP and MPTCP.

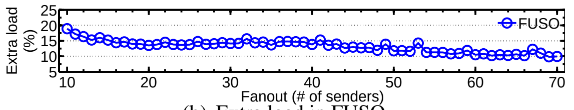
We show the average extra load of all FUSO flows in Fig. 5(c). Each flow's extra load is calculated by *the extra bytes incurred by FUSO* divided by *the total bytes transmitted*. The results show that FUSO's fast loss recovery behaviour can gracefully adapt to the network condition. Particularly, when the load is low, FUSO generates relatively more recovery packets (∼42% extra load) to proactively recover the potential loss. Such relative high redundancy rate does not affect the FUSO flows' FCT, because that FUSO only generates redundancy utilizing the opportunity when the network is not congested (detected from spare *cwnd*) and there is no more new data. As the congestion becomes severe, FUSO naturally throttles the redundancy generation (down to ∼40% in 0.9 load) by strictly following the congestion control behaviour. Later (§5.2.3) we will show that FUSO generates even lesser redundancy when the network is more congested. Note that although FUSO's redundancy helps to improve the latency flows' FCT, they do incur some extra load to the network and slightly affect the overall network throughput. Compared with MPTCP, which uses the same congestion control as FUSO but incurs zero extra load, the average throughput of all flows in FUSO is ∼7.8% lower to ∼3.4% higher at various loads.

5.2.3 Congestion Loss: Incast

Now, we focus on the congestion loss at the edge which is a very challenging scenario for FUSO, to investigate whether FUSO is cautious enough to avoid adding congestion when there is no spare capacity in the bottleneck link. We deploy a receiver application on $H1$ to simultaneously generate requests to a number of sender applications deployed on $H2$-$H6$. After receiving the request, each sender immediately responds with a 64 KB data using the maximum sending rate. This traffic pattern, which is called incast [5], is very common in MapReduce-like [37] applications. We use all physical sending hosts in our testbed to emulate multiple senders [38]. We measure the completion time when all the responses have been successfully received. In this case we do not inject failure loss.



(a) Request completion time

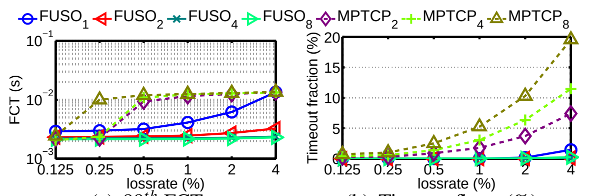


(b) Extra load in FUSO

Figure 6: Incast: Request completion time and the average extra load of all FUSO flows.

Results: Fig. 6(a) shows the request completion time as the number of senders (*i.e.*, fanout) grows. When the fanout is below 44, FUSO, MPTCP, TCP and TLP behave similarly. As studied before [24, 39], a small minRTO and appropriate ECN setting can offer a fairly good performance for standard TCP in the incast scenario. Because the total response size equals 64KB×fanout, the completion time linearly grows as the fanout increases. When fanout grows above 44, timeout occurs in MPTCP, which leads to a sudden rise of completion time. It is due to the relatively high burstiness caused by multiple sub-flows. However, FUSO's multi-path loss recovery scheme compensates this burstiness and remains an approximately linear growth of completion time in FUSO. The performance begins to degrade for all methods when the fanout exceeds 48. FUSO keeps performing the best, and keeps the completion time below 51.2ms even when the fanout becomes 70.

RepFlow and Proactive always take roughly twice the time to complete the query even when fanout is low (*e.g.*, <23), because they duplicate every flow (or packet) and add a certain excessive extra load to the network. As the fanout becomes larger, many timeouts occur and significantly impair the performance of them. For example, for a fanout of 30, RepFlow and Proactive need ∼47ms and ∼58ms to complete the request, respectively, while FUSO only needs less than 25ms. Although duplicating small flows can help to improve their performance under some lossy cases, it is not adaptive to complicated DCN environments, and even deteriorates the performance especially when the network is congested by



(a) 99th FCT (b) Timeout flows (%)

Figure 7: Various number of sub-flows: 99th FCT (log scale) and timeout fraction of the latency-sensitive flows, while one network path is lossy.

the small flows themselves. On the contrary, Fig. 6(b) shows that FUSO can gracefully adapt to the network condition and throttle the extra load in such extremely congested scenarios.

5.2.4 Various Number of Sub-flows

Now, we investigate the impact of the number of sub-flows on FUSO's performance. The settings are the same as in the network loss experiment in §5.2.1. We compare FUSO with 1,2,4 and 8 sub-flows, denoted as $FUSO_{1,2,4,8}$. Note that $FUSO_1$ simply retransmits the suspected lost packets in the same flow after the original packets before standard TCP loss recovery begins, without using multi-path.

Results: Fig. 7 shows the results. We can see that FUSO behaves better as it explores more paths using more sub-flows. Only adding redundancy without leveraging multi-path diversity causes the inferior performance of $FUSO_1$. $FUSO_4$ can offer a fairly good performance that is very close to $FUSO_8$, which means 4 sub-flows is enough for our small testbed with 3 parallel paths. On the contrary, MPTCP behaves worse as the number of sub-flows grows, lagged by the last completed sub-flow on the worst path (see §2.4).

5.3 Large-scale Simulations

Simulation settings: Besides testbed experiments, we also use ns-2 [30] to build a larger 3-layer, 4-pod simulated Fat-tree topology. The topology consists of 4 Spine switches and 4 pods below them, each containing 2 Aggregation and 2 ToR switches. All switches have 4 40Gbps ports, and each ToR switch has a rack of 8 10Gbps servers below. Each switch has 1MB buffer per port, with ECMP and ECN enabled. The whole topology contains 20 switches and 64 servers, *i.e.*, the largest scale for detailed packet-level simulation that could be finished in an acceptable time on our servers. The base RTT without queueing is $\sim 240\mu$s. Given that, the ECN threshold is set to be 300KB [7]. We set the TCP minRTO to be $5ms$ [3, 16]. The input traffic is generated the same as in §5.2.2, letting all the clients request both latency-sensitive queries and background data from randomly chosen servers. Besides web-search [1], we also evaluate another empirical data-mining workload [10]. Both FUSO and MPTCP use 8 sub-flows to adapt to the large topology. The results are from 10 runs in total, with 32K flows generated in each run.

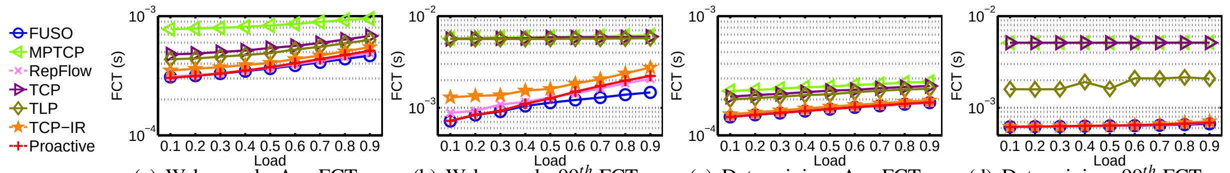
Empirical failure loss: To emulate the real condition in production data centers, we randomly set 5% links to be lossy. The loss rate of each lossy link is sampled from the distribution measured in §2.1 (Fig. 1(a)). Note that we have excluded the part in the distribution with exceptionally high loss rate (right most part in Fig. 1(a) with loss rate $> 60\%$) for sampling. It is because that standard TCP flows almost cannot finish and often upper-layer applications operations are triggered (*e.g.*, requesting resources from other machines) under such high loss rates. We randomly generate those lossy links at different locations including the edge and network, according to the real location distribution[7] in §2.1 (Fig. 1(b)).

Results: The results in Fig. 8 confirm that FUSO can gracefully scale to large topologies and complex lossy conditions. Under all loads, the average FCT of FUSO is ~10.4%-60.3% lower than TCP, MPTCP, TLP and TCP-IR in web search workload, and ~4.1%-39.4% lower in data mining workload. Also, the 99th percentile is ~29.2%-87.4% and ~0%-87.9% lower in the two workloads respectively. TCP-IR chooses a more aggressive loss recovery manner than TLP. This improves the performance, but TCP-IR still has ~29.2%-46.5% and ~0%-6.1% higher 99th FCT than FUSO under two workloads, respectively. Lacking multi-path makes TCP-IR's loss recovery less efficient, because the recovery packets may be also dropped while traversing the same lossy path as the former dropped data packets. Compared with RepFlow and Proactive which use certain excessive redundancy rate, FUSO still has up to ~33.9% and ~2.6% better 99th percentile FCT under the two workloads respectively, due to the reasons discussed before. Because the simulated topology has a much higher capacity in the fabric link (40G) than the access link (10G), the congestion is significantly alleviated compared to the small testbed topology in §5.2.2. Thus TCP performs better than MPTCP for small flows in this scenario, because their performance depends more on the failure loss.

6 Discussion

FUSO follows the principle of prioritizing new data transmission over loss recovery, utilizing spare opportunities to conduct proactive loss recovery when there is currently no new data. As such, FUSO avoids sacrificing throughput to transmit redundant recovery packets ahead of new data, which would increase the FCT. Moreover, FUSO can dynamically adapt its redundancy rate to the network condition (Fig. 5(c) and Fig. 6(b)), by strictly following the congestion control constraint. Thus, FUSO naturally generates relatively more redundancy to accelerate loss recovery when the congestion

[7]There are only 3 layers in our simulation topology, thus we merge the portion of those lossy links *at* and *above* the 3rd layer in the real topology into one layer in the simulated topology.



(a) Web-search: Avg FCT (b) Web-search: 99th FCT (c) Data-mining: Avg FCT (d) Data-mining: 99th FCT

Figure 8: Simulations under 10Gbps fat tree: Average and 99th FCT (log scale) of latency-sensitive flows from web-search and data-mining workloads. Lossy links are randomly generated according to realistic measurements in §2.1.

is light, and becomes conservative when the congestion is heavy, which outperforms other methods' (*e.g.*, TCP-IR, Repflow, and Proactive) fixed redundancy rate. As such, FUSO's redundancy helps to greatly decrease the FCT, meanwhile only slightly affecting the overall network throughput (§5.2.2).

Although we focus on how FUSO improves the performance of small latency-sensitive flows in §5, FUSO is also applicable to long flows, which are typically bandwidth greedy and *cwnd* limited. Therefore, the necessary condition for proactive multi-path loss recovery in FUSO (when there is no more new data to be sent and the flow has spare *cwnd* slots) is only triggered at the end of the long flows. Previous studies [13] have shown that MPTCP provides a very good performance for long flows. We first ran experiments to compare FUSO and MPTCP, and the results (omitted for space limitation) confirm that FUSO incurs a negligible overhead and behaves almost exactly the same as the original MPTCP for long flows. We then enabled FUSO for all small and long flows and reran all experiments in §5, and observed that FUSO still outperforms other methods at the tail FCT. We also note that enabling MPTCP in general in data centers may hurt the average FCT of small flows for the following reason (also revealed in [36,41]): Originally designed for improving long flow's throughput, MPTCP's current congestion control can cause burstiness of multiple sub-flows and drive up queue lengths on all paths. How to improve multi-path congestion control, however, is orthogonal to FUSO and beyond the scope of this paper.

7 Related Work

Besides the works [4, 13–15, 22, 23, 31] that we have previously discussed in-depth, there is a rich literature on the general TCP loss recovery (*e.g.*, [18, 19, 32, 42]), short flows' tail FCT in both DCN (*e.g.*, [43–45]) and Internet (*e.g.*, [46, 47]), and utilizing multi-path in the transport (*e.g.*, [48–51]). Due to space limitation, we do not review these works in details. The key difference between FUSO and these works is that, to the best of our knowledge, FUSO is the first work to address the long tail FCT of short flows in DCN caused by failure-packet-loss-incurred timeout. FUSO is also the first systematic work to utilize multi-path diversity to conduct proactive transport loss recovery in DCN.

It is noteworthy that several data centers have recently deployed Remote Direct Memory Access (RDMA) [52, 53], a complementary technique to TCP. It relies on Priority-based Flow Control (PFC) to remove congestion drops. However, RDMA would perform badly in face of failure-induced loss (*e.g.*, even a slight 0.1%) due to its simple go-back-N loss recovery schemes [3]. FUSO is able to deal with both congestion-induced loss and failure-induced loss, and works for the widely used TCP in DCN [3, 6, 16]. We will study how to apply the principle of FUSO to RDMA/PFC in the future.

8 Conclusion

The chase for ultra-low FCT in data center networks has been a very active research area, and the solutions range from better topology and routing designs, optical switching, flow scheduling, congestion control, to protocol architectures (*e.g.*, RDMA/PFC), *etc.* This paper adds an important thread to this area, which is to properly leverage the inherent multi-path diversity for transport loss recovery, to deal with both failure-induced and congestion-induced packet loss in DCN. In our proposed FUSO, when a multi-path transport sender suspects loss on one sub-flow, recovery packets are immediately sent over another sub-flow that is not or less lossy *and* has spare congestion window slots. Our experiments show that the *fast* yet *cautious* FUSO can decrease the tail FCT by up to ∼82.3% (testbed) and ∼87.9% (simulation).

Acknowledgment

We thank Albert Greenberg, Ming Zhang, Jennifer Rexford, Fengyuan Ren, and Zhen Xiao for their valuable feedbacks on the paper's initial version, and Xiaohui Nie, Dapeng Liu, Kaixin Sui, and Hao Wang for their suggestions on improving this work. We thank Wei Bai for the help on the implementation and experiments, and Yousef Azzabi for the proofreading. We thank our shepherd Noa Zilberman and the conference reviewers for the helpful comments. This work was partly supported by the State Key Program of National Science of China under grant 61233007, the National Natural Science Foundation of China under grant 61472214 and 61472210, the National High Technology Development Program of China (863 program) under grant 2013AA013302, the National Key Basic Research Program of China (973 program) under grant 2013CB329105, the Tsinghua National Laboratory for Information Science and Technology key projects, and the China's Recruitment Program (Youth) of Global Experts.

References

[1] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data Center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 63–74. ACM, 2010.

[2] Nandita Dukkipati and Nick McKeown. Why Flow-completion Time is the Right Metric for Congestion Control. *SIGCOMM Computer Communnication Review*, 36(1):59–62, January 2006.

[3] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis. In *Proceedings of the ACM SIGCOMM 2015 Conference*, SIGCOMM '15, pages 63–74. ACM, 2015.

[4] Tobias Flach, Nandita Dukkipati, Andreas Terzis, Barath Raghavan, Neal Cardwell, Yuchung Cheng, Ankur Jain, Shuai Hao, Ethan Katz-Bassett, and Ramesh Govindan. Reducing Web Latency: The Virtue of Gentle Aggression. In *Proceedings of the ACM SIGCOMM 2013 Conference*, SIGCOMM '13, pages 159–170. ACM, 2013.

[5] Yanpei Chen, Rean Griffith, Junda Liu, Randy H. Katz, and Anthony D. Joseph. Understanding TCP Incast Throughput Collapse in Datacenter Networks. In *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking*, WREN '09, pages 73–82. ACM, 2009.

[6] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hoelzle, Stephen Stuart, and Amin Vahdat. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Googles Datacenter Network. In *Proceedings of the ACM SIGCOMM 2015 Conference*, SIGCOMM '15, pages 63–74. ACM, 2015.

[7] Haitao Wu, Jiabo Ju, Guohan Lu, Chuanxiong Guo, Yongqiang Xiong, and Yongguang Zhang. Tuning ECN for Data Center Networks. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, pages 25–36. ACM, 2012.

[8] Matt Calder, Rui Miao, Kyriakos Zarifis, Ethan Katz-Bassett, Minlan Yu, and Jitendra Padhye. Don'T Drop, Detour! In *Proceedings of the ACM SIGCOMM 2013 Conference*, SIGCOMM '13, pages 503–504. ACM, 2013.

[9] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A Scalable, Commodity Data Center Network Architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference*, SIGCOMM '08, pages 63–74. ACM, 2008.

[10] 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 *Proceedings of the ACM SIGCOMM 2009 Conference*, SIGCOMM '09, pages 51–62. ACM, 2009.

[11] Chuanxiong Guo, Guohan Lu, Dan Li, Haitao Wu, Xuan Zhang, Yunfeng Shi, Chen Tian, Yongguang Zhang, and Songwu Lu. BCube: A High Performance, Server-centric Network Architecture for Modular Data Centers. In *Proceedings of the ACM SIGCOMM 2009 Conference*, SIGCOMM '09, pages 63–74. ACM, 2009.

[12] Yuval Bachar. Disaggregation - The New Way to Build Mega (and Micro) Data Centers. In *Architectures for Networking and Communications Systems (ANCS), 2015 ACM/IEEE Symposium on*, pages 1–1. IEEE, 2015.

[13] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. Improving Datacenter Performance and Robustness with Multipath TCP. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 266–277. ACM, 2011.

[14] Costin Raiciu, Christoph Paasch, Sebastien Barre, Alan Ford, Michio Honda, Fabien Duchene, Olivier Bonaventure, and Mark Handley. How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI '12)*, pages 29–29. USENIX Association, 2012.

[15] Alan Ford, Costin Raiciu, Mark Handley, and Olivier Bonaventure. *TCP extensions for multipath operation with multiple addresses*. Internet Engineering Task Force, January 2013. RFC 6824.

[16] Glenn Judd. Attaining the Promise and Avoiding the Pitfalls of TCP in the Datacenter. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15)*, May 2015.

[17] M Allman, V Paxson, and E Blanton. *TCP congestion control*. Internet Engineering Task Force, September 2009. RFC 5681.

[18] E Blanton, M Allman, L Wang, I Jarvinen, M Kojo, and Y Nishida. *A Conservative Loss Recovery Algorithm Based on Selective Acknowledgment (SACK) for TCP*. Internet Engineering Task Force, 2012. RFC 6675.

[19] M Allman, K Avrachenkov, U Ayesta, J Blanton, and P Hurtig. *Early retransmit for TCP and stream control transmission protocol (SCTP)*. Internet Engineering Task Force, April 2010. RFC 5827.

[20] Netem page from Linux foundation. `http://www.linuxfoundation.org/collaborate/workgroups/networking/netem`.

[21] Stephen Hemminger et al. Network emulation with NetEm. In *Linux conf au*, pages 18–23. Citeseer, 2005.

[22] N Dukkipati, N Cardwell, Y Cheng, and M Mathis. *Tail loss probe (TLP): An algorithm for fast recovery of tail losses*. Internet Engineering Task Force, April 2013. RFC 5827.

[23] Tobias Flach, Yuchung Cheng, Barath Raghavan, and Nandita Dukkipati. *TCP Instant Recovery: Incorporating Forward Error Correction in TCP*. Internet Engineering Task Force, April 2013. RFC 5827.

[24] Peng Cheng, Fengyuan Ren, Ran Shu, and Chuang Lin. Catch the Whole Lot in an Action: Rapid Precise Packet Loss Notification in Data Center. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14)*, April 2014.

[25] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *7th USENIX Conference on Networked Systems Design and Implementation*, NSDI '10, pages 19–19. USENIX Association, 2010.

[26] Christian E Hopps. *Analysis of an equal-cost multi-path algorithm*. Internet Engineering Task Force, 2000. RFC 2992.

[27] Shuihai Hu, Kai Chen, Haitao Wu, Wei Bai, Chang Lan, Hao Wang, Hongze Zhao, and Chuanxiong Guo. Explicit path control in commodity data centers: Design and applications. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15)*, May 2015.

[28] S Floyd, J Mahdavi, M Mathis, and A Romanow. *TCP Selective Acknowledgment Options*. Internet Engineering Task Force, 1996. RFC 2018.

[29] Christoph Paasch, Gregory Detal, Kenjiro Nakayama, Gucea Doru, Matthieu Baerts, Jaehyun Hwang, Sbastien Barr, and et al. MPTCP Linux kernel implementation. git://github.com/multipath-tcp/mptcp.

[30] The Network Simulator - ns-2. *http://www.isi.edu/nsnam/ns/*.

[31] Hong Xu and Baochun Li. RepFlow: Minimizing Flow Completion Times with Replicated Flows in Data Centers. In *Proceedings of the IEEE INFOCOM 2014 Conference*, pages 1581–1589, April 2014.

[32] Sally Floyd, Hari Balakrishnan, and Mark Allman. *Enhancing TCP's loss recovery using limited transmit*. Internet Engineering Task Force, January 2001. RFC 3042.

[33] Net-tcp-fec: Modifications to the Linux networking stack to enable forward error correction in TCP. *http://tflach.github.io/net-tcp-fec/*.

[34] Wei Bai, Li Chen, Kai Chen, and Haitao Wu. Enabling ECN in Multi-Service Multi-Queue Data Centers. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI '16)*, March 2016.

[35] Edgar N Gilbert. Capacity of a Burst-Noise Channel. *Bell system technical journal*, 39(5):1253–1265, 1960.

[36] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. CONGA: Distributed Congestion-aware Load Balancing for Datacenters. In *Proceedings of the ACM SIGCOMM 2014 Conference*, SIGCOMM '14, pages 503–514. ACM, 2014.

[37] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[38] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. Better Never Than Late: Meeting Deadlines in Datacenter Networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 50–61. ACM, 2011.

[39] Vijay Vasudevan, Amar Phanishayee, Hiral Shah, Elie Krevat, David G. Andersen, Gregory R. Ganger, Garth A. Gibson, and Brian Mueller. Safe and Effective Fine-grained TCP Retransmissions for Datacenter Communication. In *Proceedings of the ACM SIGCOMM 2009 Conference*, SIGCOMM '09, pages 303–314. ACM, 2009.

[40] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pFabric: Minimal Near-optimal Datacenter Transport. In *Proceedings of the ACM SIGCOMM 2013 Conference*, SIGCOMM '13, pages 435–446. ACM, 2013.

[41] Jiaxin Cao, Rui Xia, Pengkun Yang, Chuanxiong Guo, Guohan Lu, Lihua Yuan, Yixin Zheng, Haitao Wu, Yongqiang Xiong, and Dave Maltz. Per-packet Load-balanced, Low-latency Routing for Clos-based Data Center Networks. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '13, pages 49–60. ACM, 2013.

[42] Matthew Mathis and Jamshid Mahdavi. Forward Acknowledgement: Refining TCP Congestion Control. In *ACM SIGCOMM Computer Communication Review*, volume 26, pages 281–291. ACM, 1996.

[43] David Zats, Tathagata Das, Prashanth Mohan, Dhruba Borthakur, and Randy Katz. DeTail: Reducing the Flow Completion Time Tail in Datacenter Networks. In *Proceedings of the ACM SIGCOMM 2012 Conference*, SIGCOMM '12, pages 139–150. ACM, 2012.

[44] Balajee Vamanan, Jahangir Hasan, and T.N. Vijaykumar. Deadline-aware Datacenter TCP (D2TCP). In *Proceedings of the ACM SIGCOMM 2012 Conference*, SIGCOMM '12, pages 115–126. ACM, 2012.

[45] Matthew P Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert NM Watson, Andrew W Moore, Steven Hand, and Jon Crowcroft. Queues Dont Matter When You Can JUMP Them! In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15)*, pages 1–14, 2015.

[46] Qingxi Li, Mo Dong, and P Brighten Godfrey. Halfback: Running Short Flows Quickly and Safely. In *Proceedings of the 11th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '15. ACM, 2015.

[47] Jianer Zhou, Qinghua Wu, Zhenyu Li, Steve Uhlig, Peter Steenkiste, Jian Chen, and Gaogang Xie. Demystifying and Mitigating TCP Stalls at the Server Side. In *Proceedings of the 11th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '15. ACM, 2015.

[48] Ming Zhang, Junwen Lai, Arvind Krishnamurthy, Larry L Peterson, and Randolph Y Wang. A Transport Layer Approach for Improving End-to-End Performance and Robustness Using Redundant Paths. In *USENIX Annual Technical Conference, General Track*, pages 99–112, 2004.

[49] Peter Key, Laurent Massoulié, and Don Towsley. Path Selection and Multipath Congestion Control. In *Proceedings of the IEEE INFOCOM 2007 Conference*, pages 143–151. IEEE, 2007.

[50] Yong Cui, Lian Wang, Xin Wang, Hongyi Wang, and Yining Wang. FMTCP: A Fountain Code-based Multipath Transmission Control Protocol. *Networking, IEEE/ACM Transactions on*, 23(2):465–478, 2015.

[51] Morteza Kheirkhah, Ian Wakeman, and George Parisis. MMPTCP: A Multipath Transport Protocol for Data Centers. In *Proceedings of the IEEE INFOCOM 2016 Conference*, April 2016.

[52] 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 *Proceedings of the ACM SIGCOMM 2015 Conference*, SIGCOMM '15, pages 523–536. ACM, 2015.

[53] 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 *Proceedings of the ACM SIGCOMM 2015 Conference*, SIGCOMM '15, pages 537–550. ACM, 2015.