Wireless Networks
https://doi.org/10.1007/s11276-020-02443-8

Reducing web latency with coding-based fast multi-path loss recovery

Yi Liu[1] · Guihua Zhou[1] · Guo Chen[1,2]

Abstract
TCP latency is critical to the performance of Web services. However, packet loss greatly impairs the TCP performance due to its poor loss recovery mechanisms. Recent work FUSO addressed this problem by leveraging multi-path diversity for proactive loss recovery, i.e., using "good" paths to proactively retransmit the potentially lost packet on "bad" paths before they are retransmitted after duplicate ACKs or timeout. Nevertheless, since it has no clue about which packet is (or will be) lost, FUSO simply proactively retransmits the oldest unACKed packet whenever there is a chance for proactive loss recovery. Through analysis and comprehensive experiments, we show that although FUSO behaves well in data center networks, which it is originally designed for, in the Internet scenario, such simple proactive retransmission of the oldest unACKed packet is not accurate enough to recover the lost packets, which causes performance penalty. To address the problem, this paper presents CoFUSO, a *Co*ding-Based *F*ast M*u*lti-Path Lo*s*s Rec*o*very. Different from FUSO, when there is a chance for proactive loss recovery, CoFUSO generates a coding packet that codes all (or multiple) unACKed packets together. As such, CoFUSO can always proactively retransmit the "right" lost packet, since the receiver side can decode the lost packet by combining the coding packet with other received packets. We implement CoFUSO in Linux kernel with $\sim$2K lines of code. Testbed and simulation results show that, under lossy condition, CoFUSO can greatly improve the average and 99th percentile flow completion time (FCT) by $\sim$12% and $\sim$59% in the testbed, and up to $\sim$16.9% and $\sim$54.5% in the simulation, respectively.

Keywords Packet loss · Transport loss recovery · Multi-path transport · Coding

1 Introduction

TCP is the underlying transport protocol of most modern online Web services [1–3]. Its transmission time is critical to the Web service performance [1] which greatly affects the company revenue [2, 4]. However, packet loss has been shown to be the most significant factor causing poor TCP performance especially for short TCP flows in Web services [1, 5], mainly due to TCP's ineffective loss recovery mechanism.

Previous works try to add aggressiveness to TCP thus to speed up loss recovery [1, 5]. They start proactive retransmission before packets are detected to be lost through ordinary duplicate ACKs or retransmission timeouts (RTO). However, deciding the degree of aggressiveness is hard since the network condition and traffic vary rapidly. Particularly, being too conservative will delay the loss recovery and the transmission time is still not improved enough; being too aggressive, however, may disturb TCP congestion control and impair performance due to increased congestion.

Our recent work FUSO [6, 7] solved the problem above by conducting proactive loss recovery using the opportunity when there is spare congestion window (*cwnd*) and no new data to send, which adds no aggressiveness to existing congestion control. By leveraging multi-path diversity that offers plenty of such opportunities for proactive loss recovery, i.e., using "good" paths which have spare *cwnd* to proactively retransmit potentially lost packets for "bad" paths, FUSO can be both *fast* and *cautious*.

Proactive retransmission can speed up loss recovery. However, as packets are not verified to be lost yet, it is

✉ Guo Chen
guochen@hnu.edu.cn

[1] College of Computer Science and Electronic Engineering, Hunan University, Changsha, Hunan, China

[2] Science and Technology on Parallel and Distributed Processing Laboratory (PDL), Changsha, China

Published online: 20 August 2020

challenging for FUSO to predict which unACKed packet is most likely to be lost. As such, it simply retransmits the oldest unACKed packet whenever there is an opportunity to conduct proactive retransmission. Nevertheless, the oldest unACKed packet is often not the lost packet. Simple testbed experiments (Fig. 4) show that FUSO typically ($\sim$50% flows) needs to retransmit 4–16 redundant packets before it finally hits the right one (which is lost).

Such patterns provide good performance in data center networks (which FUSO is originally designed for), since it has plenty of proactive retransmission opportunities due to high bandwidth and small round-trip-time (RTT). However, this proactive retransmission does not apply to the Internet scenario. Specifically, Internet paths (especially to mobile devices) often have relatively small bandwidth, which offers very limited chances to conduct proactive retransmission while obeying congestion control. For instance, our experiments (Fig. 5) show that, after all data has been sent out, $\sim$80% Web flows only have spare windows to send less than 4 packets in 20 ms, and $\sim$60% Web flows take more than 50 ms ($\sim$1 RTT) to accumulate enough spare transmission chances to send 4 or more proactive recovery packets. Therefore, with such limited opportunities, several miss-retransmissions (e.g., 4$\sim$16) would greatly weaken and even neutralize the effects of proactive loss recovery.

In this paper, we aim to address above problem through *Co*ding-Based *F*ast M*u*lti-Path Lo*s*s Rec*o*very, CoFUSO. CoFUSO adopts the same multi-path loss recovery manner and bears the same philosophy of being both fast and cautious as FUSO. But instead of retransmitting from the oldest unACKed packet, CoFUSO generates proactive recovery packets using an erasure code [8]. Specifically, when there is a chance to conduct proactive retransmission, CoFUSO will send a coding packet which codes all the potentially lost packets together. As such, the sender can always "retransmit" the right packet without accurately predicting which one is lost, since the receiver can decode the actually lost packet once it has received other packets. To control the coding overhead, we devise an algorithm to dynamically choose the appropriate coding rate according to the network condition, meanwhile keeping the recovery efficiency.

The major contributions of this work are summarized as follows:

- Through targeted testbed experiments and analysis, we show that FUSO's proactive retransmission (from the oldest unACKed packet) is inaccurate and ineffective for Internet Web services.
- We design a novel coding-based multi-path loss recovery scheme, which greatly improves the retransmission accuracy of FUSO using erasure code.
- We implement CoFUSO in Linux kernel with 2077 lines of code. Testbed and simulation results show that CoFUSO can greatly improve the loss recovery performance. Particularly, compared to the latest loss recovery scheme, CoFUSO reduces the average flow completion time (FCT) by $\sim$12% and the 99th percentile FCT by $\sim$59% in the testbed, and up to $\sim$16.9% and $\sim$54.5% for the average and 99th percentile FCT in the simulation, respectively.

2 Related work

The inefficiency of loss recovery is a well known problem that hurts TCP performance, especially when retransmission timeout occurs. Previous single-path works add different aggressiveness level to congestion control thus to speedup loss recovery before timeout. For example, Reactive recovery [1] transmits one prober after 2RTT to trigger duplicate ACKs, while Proactive [1] and RepFlow [9] aggressively transmit every duplicated packets or flow for excessive redundancy. However, there are two causes of packet loss [6, 7]: congestion and link/router's intrinsic character. As such, their fixed aggressiveness can not adapt to different network conditions. Particularly, a high aggressiveness degree may worsen congestion in congested scenario, while a low aggressiveness would delay the loss recovery for lossy networks such as WiFi.

Our previous work FUSO solves above problem by utilizing multi-path for loss recovery, and conducts proactive recovery while strictly following congestion control. CoFUSO improves the recovery accuracy of FUSO through coding, instead of simply retransmitting the oldest unACKed packet. Corrective recovery [1] also uses coding, but its constant aggressiveness and single-path character makes it perform much inferiorly than CoFUSO (see evaluation results before).

There are some previous works that combine coding techniques to improve TCP performance. TCP-NC [10] sends some coding packets to the receiver periodically controlled by a timer, and achieves good results under the circumstance that the packet loss rate is less than 10%. Based on TCP-NC, TCP-VON [11] adopts online coding to keep continuous decoding at the receiver, which reduces decoding delay. TCP-FNC [12] adds FCWL (Feedback-based Coding Window Lock) for redundancy compensation and EFU (Eliminate at Fewer Unseen) for decoding. All these works are different from our CoFUSO because they only worked for single-path TCP which didn't take advantage of multiple paths. Moreover, their coding algorithms produce fixed redundant data during transmission. However, CoFUSO works as a loss recovery scheme, only

generating redundancy when new data have all been sent out and there are potential loss.

There are works that apply coding to MPTCP. MPC-TCP [13] implements coding across sub-flows and encodes at the overall flow level. Different from CoFUSO 's sub-flow level coding, this kind of coding does not consider the difference between the sub-flows, which difficults the use of "good" paths to help "bad" paths. We have discussed this with more details in Sect. 4.4. NC-MPTCP [14] uses a mixture of regular and coding sub-flows. Regular sub-flows transmit the original data, and coding sub-flows transmit the coding data which can be decoded by the receiver. It selects dedicated sub-flows to transmit the coded packet, which cannot harness the path diversity if the chosen sub-flows happen to go through a "bad" path. In contrary, CoFUSO dynamically generates coding packets in each path according to its condition. Similarly, FMTCP [15] incorporates fountain code into MPTCP to improve the throughput and latency over Internet. Data packets in FMTCP are first encoded into symbols by adding redundancy, and then these encoded symbols are sent through multiple sub-flows of MPTCP. However, different from CoFUSO, FMTCP generates constant redundancy, which is not a loss recovery scheme like CoFUSO that only starts when guessing there are potentially lost packets while still obeying congestion control. FMTCP's redundant symbols generated through coding consume a constant bandwidth, which may increase the FCT when congestion happens. Moreover, to our knowledge, MPTCP-NC and FMTCP have only simulated their algorithm without building a real kernel implementation like CoFUSO.

Some recent works also enable multi-path to other transports such as RDMA [16, 17]. However, they do not apply coding techniques to improve loss recovery efficiency.

3 Background and motivation

3.1 FUSO background

FUSO is a fast (proactive) multi-path loss recovery scheme. Fig. 1 shows the architecture of fast multi-path loss recovery. It works on multi-path transport where a TCP flow is divided into multiple sub-flows.[1] If there is a spare *cwnd* and no more new data delivered from the upper layer application, FUSO utilizes this transmission opportunity for proactive recovery. Specifically, FUSO monitors each path's loss condition, and proactively/immediately recover those potentially lost packets on "bad" sub-flows, by utilizing "good" sub-flows. Proactive recovery packets are transmitted as normal data packets on the "good" sub-flows, thus under the congestion control without adding aggressiveness. Since a packet is still not verified to be lost, for every chance of proactive recovery, FUSO simply regards the oldest unACKed packet (not recovered before) on the worst sub-flow as the one most likely to be lost, and retransmits it. Algorithm 1 summarizes how fast multi-path loss recovery works.

Algorithm 1 Proactive multi-path loss recovery.

```
1: function TRY_SEND_RECOVERIES( )
2:     while BytesInFlight_Total < CWND_Total and no new data do
3:         res ← SEND_A_RECOVERY( )
4:         if res == NOT_SEND then
5:             break
1: function SEND_A_RECOVERY( )
2:     FIND_WORST_SUB-FLOW( )
3:     FIND_BEST_SUB-FLOW( )
4:     if no worst or no best sub-flow found then
5:         return NOT_SEND
6:     recovery_packet ← GENERATE_RECOVERY_PACKET( )
7:     Send the recovery packet through the best sub-flow
8:     BytesInFlight_Total += Size_recovery_packet
```

FUSO is originally designed for data center networks (DCN), which has plenty of parallel paths with loss diversity among them. It has been shown that FUSO can significantly outperform prior loss recovery mechanisms in DCN [6, 7].

3.2 Multi-path to access web services

Besides DCN, multi-path environment also exists for accessing online Web services, and such multiple paths have large diversity in loss rate. This also offers us a good opportunity to conduct fast multi-path loss recovery.

Taking mobile devices which contribute to the majority traffic of modern Web services [18] as an example, they are often multi-homed to the network (e.g., 4G and WiFi). Current mainstream mobile devices have been already equipped with multi-path transport stack [19, 20]. This further enables them to readily use multiple paths simultaneously to access online services.

Loss rates among those paths can be very different. For example, Fig. 2 shows the loss rate of a 4G and a WiFi access path measured in our Lab during one day. Our lab contains about $20 \sim 30$ students sitting in a ~ 30 m^2 room. Those students share campus WiFi through a single wireless access point. We use two mobile devices, one using campus WiFi and one using 4G, to simultaneously Ping several top websites in China. Results in Fig. 2 show that while WiFi often has high loss rate (up to $\sim 2\%$) in a dense

[1] "Sub-flows" and "paths" are interchangeably used in the paper.

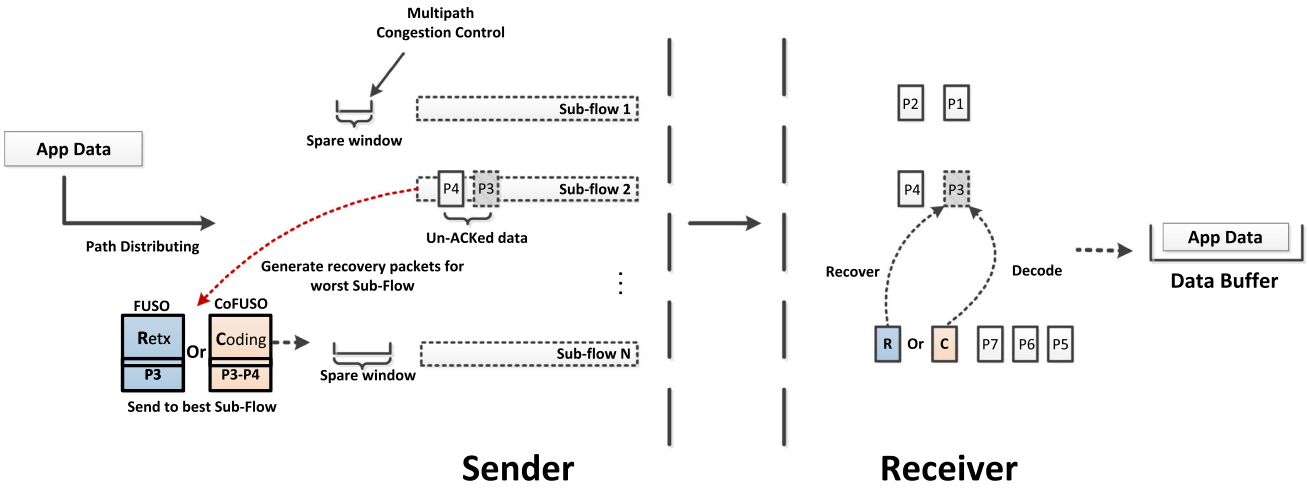


Fig. 1 Fast multi-path loss recovery: FUSO and CoFUSO

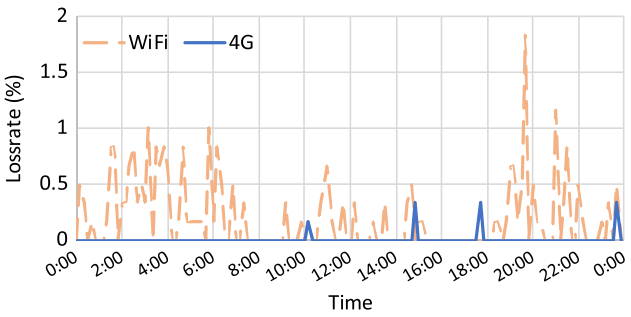


Fig. 2 Loss diversity is large among multiple paths for a typical mobile device to access Web services

environment (about 20 students working in our lab), 4G usually behaves very well and only has a very low loss rate (below 0.4%) in a few rare cases.

3.3 Retransmit the oldest unACKed packet is not enough

With the multi-path environment as introduced before, it is natural to use multi-path loss recovery such as FUSO to accelerate TCP[2] transmission time. However, FUSO conducts proactive recovery simply by early retransmitting from the oldest unACKed packet. This is not accurate enough to recover the lost packet, which makes FUSO perform inferiorly for Web services.

We conduct a targeted testbed experiment to show the effect of such inaccurate proactive retransmission. There are two paths between our client and server, both having an RTT of 50 ms, and we manually induce 2% random loss rate in one path (detailed testbed settings will be introduced in Sect. 6.1). The client requests the server a certain amount of data 5000 times. The data size is sampled from our measurement statistics of one-week real mobile search responses in Baidu [22] (see Fig. 3 for details).

Figure 4 shows, for those flows in which proactive recovery has successfully recovered at least one packet, the amount of extra recovery packets that have been transmitted before FUSO's proactive loss recovery could hit the lost packet. In FUSO, about 50% flows need to retransmit $4 \sim 16$ redundant recovery packets before they hit the right lost packet.

Such inaccuracy of proactive recovery leads to significant performance penalty. Figure 5 shows that after all data has been sent out, the amount of spare transmission chances the flows can have for proactive recovery, within 20 ms, 50 ms, and 100 ms, respectively. $\sim$78% flows have no more than 4 chances to send proactive recovery packets within 20 ms. If FUSO fails to retransmit the right one within these 4 packets, $\sim$60% flows take more than 50 ms ($\sim$1 RTT) to accumulate enough spare transmission chances to send more than 4 proactive recovery packets.

As shown in Fig. 4, through coding, CoFUSO can greatly improve the accuracy of proactive recovery. We will discuss the testbed results of CoFUSO with more details later in Sect. 6.1.2.

4 CoFUSO design

4.1 Overview

CoFUSO follows the core scheme of FUSO as described in Sect. 3.1, but differs in the manner of generating proactive recovery packets, as shown in Fig. 1. Specifically, at the sender side, when there is an opportunity for proactive recovery, CoFUSO generates a coding packet from

[2] For ease of presentation, in this paper, TCP refers to both TCP and multi-path transport such as MPTCP [21] used for accessing web services. They have the same basic loss recovery mechanism, i.e., through duplicate ACKs and RTOs.

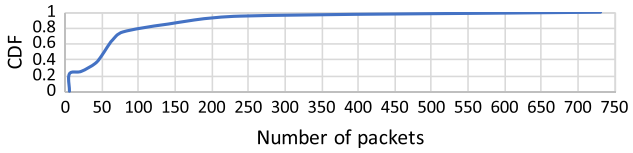


Fig. 3 Flow size distribution measured from one-week mobile search responses in Baidu [22]

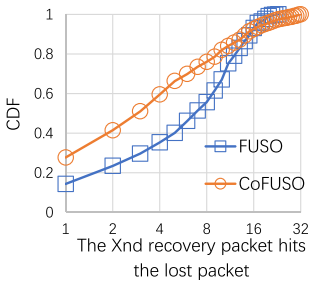


Fig. 4 The number of extra recovery packets transmitted before hitting the lost packet

multiple un-ACKed packets which are suspected as lost. The receiver will try to decode the lost packets using these coding packets and complete the data transmission, without waiting the retransmission of those lost packets by original loss recovery scheme.

To clarify better, we introduce CoFUSO sender and receiver design separately. In a CoFUSO connection, the sender is the end host which sends data, while receiver sends ACK. Note that both ends are simultaneously the sender and receiver in a two-way connection.

Algorithm 2 Generating coding packet in CoFUSO.

N: number of current unACKed packets on the worst subflow
N_{uc}: number of unACKed & uncoded packets on the worst subflow (init: N)
K_{max}: max coding block size (number of packets)
K: size of current coding block (init: 0)
M_{max}: max number of parities for one coding block
M: max number of parities for current coding block
i: the next parity sequence (init: 0)
L: estimated loss rate on the worst subflow

1: **function** GENERATE_RECOVERY_PACKET( )
2: **if** i==0 **then** ▷ A new coding block starts
3: N_{uc}=$N_{uc} - K$
4: K=min($\frac{M_{max}}{L}$, N_{uc}, K_{max})
5: M=$K \times L$
6: **if** $K > 0$ **then** ▷ Can generate coding packet
7: recovery_packet ← the i_{th} parity for the K unACKed packets in this coding block
8: $i = i + 1 \bmod M$
9: **else** ▷ Downgrade to FUSO
10: recovery_packet ← the next oldest unACKed packet
return recovery_packet

4.2 CoFUSO sender

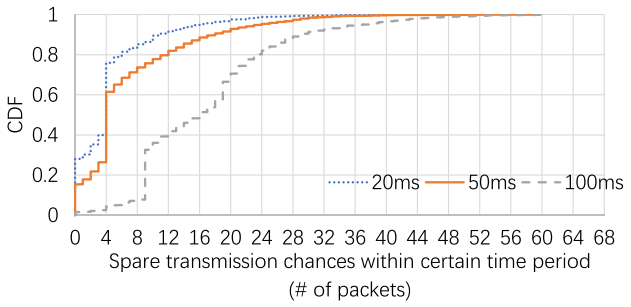


Fig. 5 Spare transmission chances within certain time period (20ms, 50ms, 100ms) to conduct proactive recovery

The CoFUSO sender processing is the same as FUSO, but uses a different function to generate recovery packets (line 6 in Algorithm 1) whenever there is a chance for proactive recovery. The left part of Fig. 6 illustrates how CoFUSO sender generates coding packets for multi-path proactive loss recovery, and the detailed process is shown in Algorithm 2.

4.2.1 Coding scheme

CoFUSO uses a version of systematic Reed-Solomon codes (RS-code) [8] to generate coding packets. RS-code is a kind of linear grouping cyclic redundancy code, which is the optimal erasure correction code with the property of maximum distance separable. Compared with other codes, it has stronger error-correcting capability without the problem of error layer and can recover more data with less redundancy. Specifically, we find the worst sub-flow that is most likely to have packets dropped, and generate M coding packets (also called parities) for K unACKed packets (called a coding block) (line 7 in Algorithm 2). Theses M parities can be used to recover up to M original packets. In theory, M can be arbitrarily large to recover arbitrary number of lost packets. However, in practice, the decoding complexity increases with $O(M^2)$. Also, K should be small thus to minimize the memory consumption on the receiver side to buffer packets for decoding. Since M dominates the decoding time, M is set with a upper bound of M_{max} considering both the computation overhead and the network RTT. Specifically, M_{max} is set to make the decoding time smaller than 1/4 RTT (not shown in

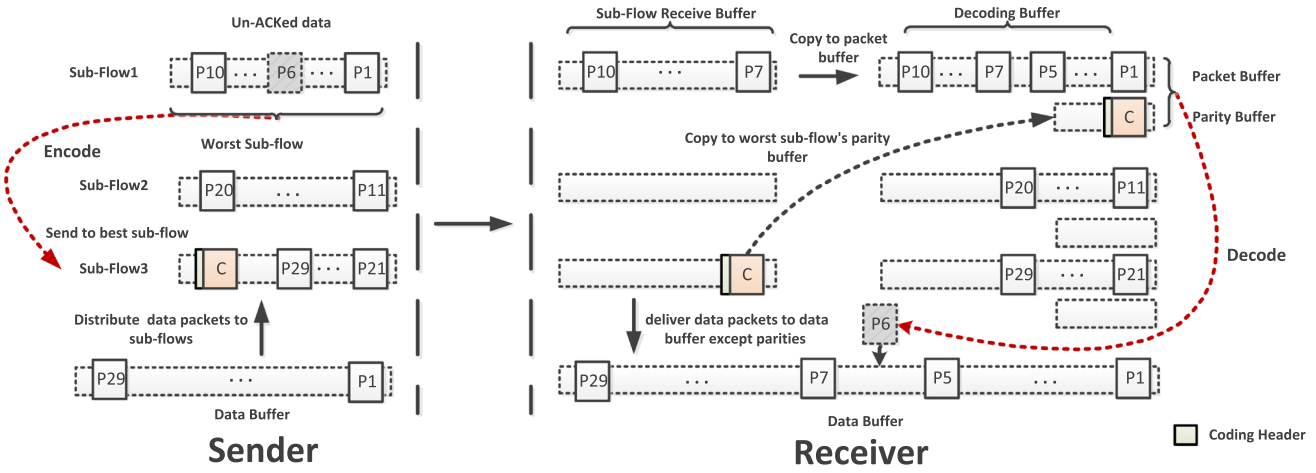


Fig. 6 The process of encoding and decoding in CoFUSO: an example

Algorithm 2), in order to be faster than simple retransmission.[3] Similarly, K is set with an upper bound of K_{max} considering the memory resource on the receiver side,[4] which can be typically very large thanks to the richness of host memory.

4.2.2 Dynamic coding rate

To further minimize the computation and memory overhead, we dynamically select appropriate coding rate (M and K) according to the network condition, as shown in line $3 \sim 5$ of Algorithm 2. Specifically, for each coding block, K is set to the min of $\frac{M_{max}}{L}$, N_{uc}, and K_{max}, where N_{uc} denotes the number of unACKed packets on the worst sub-flow which are also uncoded before, so we may need to recover them for potential loss. L is the estimation of the worst sub-flow's current loss rate (discussed in Sect. 4.2.4). Therefore, with high probability, M_{max} packets will get lost among $\frac{M_{max}}{L}$ packets ($\frac{M_{max}}{L} \times L = M_{max}$). As such, considering at most M_{max} parities can be generated, they are typically enough to recover the M_{max} lost packets in $\frac{M_{max}}{L}$ original packets, which is the desired coding block size. To reduce overhead, the parity parameter M for a coding block can be decreased to smaller than M_{max}, if there is not enough unACKed original packets left (line 5).

4.2.3 Fully utilizing transmission opportunity

In practice, M_{max} is usually small due to computation overhead. For instance, we set M_{max}=1 in our testbed to minimize the coding overhead (details in Sect. 5). As such, there is typically still spare *cwnd* left when parities have been generated for all the unACKed packets. As such, CoFUSO leverages such opportunity to conduct further proactive recovery. Specifically, if there are further transmission opportunities, CoFUSO will downgrade to FUSO and retransmit the currently oldest unACKed packet on the worst sub-flow (line 9–10). Note that an un-ACKed packet will be sent at most once by the proactive loss recovery scheme and the parities are neglected, thus to avoid adding too much unnecessary traffic to the network.

4.2.4 Path condition

CoFUSO needs to monitor path condition for the following two reasons: First, CoFUSO sender needs to monitor the loss rate of each path thus to select the appropriate coding rate (see Sect. 4.2.2). We estimate the loss rate L of a sub-flow as the same in FUSO. Specifically, $L = \alpha_1 lossrate_{overall} + \beta_1 lossrate_{last}$, which is the weighted sum of the overall packet loss rate $lossrate_{overall}$ and the most recent packet loss rate $lossrate_{last}$. $lossrate_{overall}$ is calculated as the ratio of total retransmitted packets *to* the total transmitted packets, and $lossrate_{last}$ is calculated as the ratio of one to the number of transmitted packets from (including) the last retransmission, which is updated every time CoFUSO detects a proactive recovery opportunity. Second, since Internet paths may differ a lot in terms of delay, different from FUSO, CoFUSO also considers RTT when selecting worst and best sub-flows for proactive recovery (line $2 \sim 3$ in function SEND_A_RECOVERY in Algorithm 1). Specifically, we normalize the loss rate and RTT across all sub-flows, and consider the weighted sum of both normalized loss rate and RTT (i.e., $\alpha_2 \overline{L} + \beta_2 \overline{RTT}$) as the metric to select the worst and best sub-flows.

[3] Encoding is quick in RS-code so we mainly consider decoding time.

[4] RS-code uses online encoding which requires no extra buffer at the sender side.

4.3 CoFUSO receiver

In multi-path transport protocol such as MPTCP, the receiver has a data-level receive buffer and each sub-flow has a virtual receive buffer that is mapped to the data-level receive buffer [21, 23]. As shown in the right part of Fig. 6, in CoFUSO receiver, we design a set of extra buffers for each sub-flow to decode packets, i.e., (1) a *packet buffer* to store original packets and (2) a *parity buffer* to store parities.

Since decoding requires a combination of data packets and parities, after receiving in-order data packets, CoFUSO will first deliver them to data-level receive buffer, and also copy them into the sub-flow packet buffer for potential decoding. Each packet is inserted into the buffer position based on its sequence number. Once the number of received data packets and parities equals the coding block size, CoFUSO receiver will decode the missing packets and deliver them to the data-level receive buffer. CoFUSO then clears the packets and parities belonging to current coding block in the packet buffer and parity buffer.

The packet buffer is a circular buffer with maximum size of K_{max}. This ensures that CoFUSO receiver has enough buffer space to decode the largest coding block. Packets with sequence that exceeds the buffer limit will wipe out the packets ahead to make space in the buffer. Correspondingly, the parity buffer has a maximum size of M_{max}. Parities are stored into the corresponding sub-flow's parity buffer sequentially according to their coding block sequences and then their sequences within a block.

4.4 Discussion

We discuss a few design points here.

Coding/Decoding at sub-flow level We currently choose coding/decoding packets at sub-flow level instead of flow level. Specifically, it is easy to use the metric introduced in Sect. 4.2.4 to find a sub-flow that is likely to drop packets, and conduct proactive recovery for it. On the contrary, it is not easy to identify which packet has higher probability to be lost at flow level.

Strictly following congestion control All proactive recovery packets transmitted on the good sub-flow (both coding and retransmission) are regarded as normal packets under congestion control. Moreover, packet recovery works above the bad sub-flows, thus will not generate ACKs in the bad sub-flows and not affect their congestion control. Note that lost packets will also be retransmitted by the bad sub-flows themselves, however, the flow is pre-emptively finished without waiting for the ordinary loss recovery in the bad sub-flows.

Prioritizing new data transmission Same as FUSO, CoFUSO follows the principle of prioritizing new data transmission over its proactive loss recovery, thus to avoid sacrificing throughput to transmit redundant recovery packets which can be used for data packets. Whenever new data has been pushed in, CoFUSO will stop generating subsequent parities even though the parities have not been all generated for this coding block. CoFUSO may continue to generate next parities if spare transmission opportunity comes again (or generate new coding block if last block has been ACKed).

5 Implementation

We implement CoFUSO in Linux kernel 3.18 with 2077 lines of code based on FUSO [24], which is built on top of MPTCP's Linux implementation v0.90 [23]. Next, we will describe the details of CoFUSO 's packet format, and how we implement the CoFUSO sender and receiver. Currently, we only implement a simple coding for CoFUSO, i.e., only generate one XOR coding packet for each coding block. Generating more coding packets would require to integrate the RS-code implementation within the Linux kernel, which is beyond the scope of this paper and will be our future work.

5.1 Packet format

To carry necessary coding information, CoFUSO inserts a coding header behind the original TCP and MPTCP header, as shown in Fig. 7. Note that the format of normal data packets are the same in CoFUSO and in MPTCP. Since CoFUSO may also use simple proactive retransmission for unACKed packet (see Sect. 4.2.3), we use a reserved bit **R_or_C** in the TCP header to identify whether this recovery packet is a coding packet (R_or_C=0) or a simple retransmitted packet (R_or_C=1).

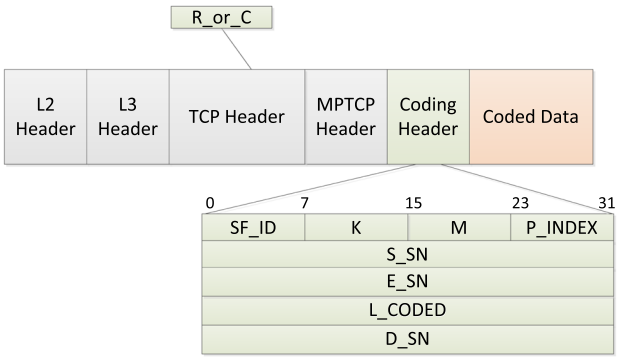


Fig. 7 Format of coding (parity) packets

The coding header consumes 20 bytes in total. Specifically, it contains the following fields:

- **SF_ID** (8 bits): This field identifies which sub-flow is this coding packet for. Both ends negotiate a consistent mapping from SF_ID to sub-flow while establishing the CoFUSO connection.
- **S_SN** (32 bits) and **E_SN** (32 bits): These two fields indicate the start and end sequence number of the data packets that are coded in this coding block. The sequence number is sub-flow-level sequence.
- **K** (8 bits), **M** (8 bits) and **P_Index** (8 bits): K indicates the coding block size, i.e., the number of data packets that are coded in this coding block. M indicates the max number of parity packets for this block, and P_Index shows this parity packet is the Xnd parity in this block.
- **L_CODED** (32 bits): This field is set with the result of encoding every data packet's size in this coding block.
- **D_SN** (32 bits): This field is set with the result of encoding every data packet's data-level sequence number (DSN) in this coding block.

5.2 CoFUSO sender

As shown in Algorithm 1, for CoFUSO sender, we rewrite the GENERATE_RECVOERY_PACKET() in FUSO's sender to implement our encoding process (Algorithm 2). Currently, we implement one parity (i.e., XOR) for each coding block, and set K_{max} to 30. α_1 is set to 0.05, β_1 is set to 0.95, and α_2, β_2 are set to 0.5 in testbed.

When generating parity (coding) packet, besides XOR the data, we also need to XOR two extra information of each data packet into the parity packet:

- The first is each packet's data length. Since different unACKed packets may have different packet length, some packet's data may be padded to the maximum length when calculating the XOR. As such, when encoding, we XOR all packet's length and assign the result to the field L_CODED in the coding header of the parity. Thus when decoding, we can also know the actual length of the recovered packet by decoding the field L_CODED, and remove the padding part in it.
- The second is each packet's data-level sequence number (DSN). When encoding, we also XOR all packet's DSN and assign the result to the field D_SN in the coding header of the parity. As such, CoFUSO receiver can decode the lost packet and directly put it into the data-level buffer.

There are a few more things that need to be mentioned. Since the extra coding header consumes 20 bytes, we set the MSS of data packets to be 20 bytes smaller than the actual MSS that can be supported by the network MTU. As such, the parity packet will not be fragmented by the underlying network hardwares. Furthermore, in our implementation, we keep the original MPTCP header for parity packets, thus to maximize the utilization of existing MPTCP processing logic (e.g., calculating MPTCP checksum and other various sub-flow-level processing). Specifically, we simply copy one data packet's MPTCP header to the parity packets as a "dummy" MPTCP header, and "reinject" the packet into the "good" sub-flow's send queue as normal data. At the receiver side, we will intercept those parity packets and process it according to the CoFUSO logic without considering the "dummy" MPTCP header.

5.3 CoFUSO receiver

As introduced before, CoFUSO receiver allocates two extra cyclic buffers besides the existing sub-flow receive buffer for decoding packets. When receiving data packets and parity packets, the processing logic is as follows:

- A data packet will be processed by the existing MPTCP logic, e.g., generating ACKs and updating transmission states in the sub-flow. Then it will be pushed into the sub-flow receive buffer and then to the data-level buffer.[5] Beyond that, CoFUSO will also copy these data packets into each sub-flow's packet buffer for potential decoding in the future.
- Coded (parity) packets are also processed by the existing MPTCP logic so the congestion control on the "good" sub-flow (who transmits them) behaves correctly. But at the last moment, CoFUSO will intercept those parity packets without delivering them to the data-level receive buffer, since their coded data has no meaning to the application. Instead, CoFUSO pushes them into corresponding sub-flow's (which may be proactively recovered by them) parity buffer for potential decoding.

When receiving either a parity or data packet, CoFUSO will check the packet buffer and parity buffer of the corresponding sub-flow to see whether there are enough data and parity packets arrived in a coding block for decoding.

6 Evaluation

We evaluate CoFUSO both in a small testbed and larger-scale NS2 [25] simulation. Basically, our evaluations aim to show the following points:

[5] Note that we also adopt the receiving side optimization in FUSO to directly push the sub-flow data packet into the data-level receive buffer.

- In the **testbed** experiments: (1) we use targeted packet-drop cases to give a concrete concept of how CoFUSO speeds up the loss recovery (Sect. 6.1.1); (2) we use empirical random loss and flow sizes sampled from real traffic to show that CoFUSO can significantly reduce the tail FCT under realistic conditions (Sect. 6.1.2); (3) we show that the encoding and decoding process in CoFUSO incur negligible overhead in normal transmission (Sect. 6.1.3).
- In the **simulation** experiments: (1) we compare with various latest loss recovery schemes and show that CoFUSO performs the best under more complex lossy conditions (Sect. 6.2.2); (2) we incur concurrent flows competing the shared links and show CoFUSO also performs very well under congestion conditions (Sect. 6.2.3); (3) we use several targeted scenarios to examine CoFUSO 's detailed behavior (Sect. 6.2.4).

Next, we will go into the testbed and simulation experiments, respectively.

6.1 Testbed

We build a small testbed as shown in Fig. 8. There are two hosts connected with a three-port router (the client connected through two links and the server through one). Both hosts are Linux virtual machine (Ubuntu 16.04.4 with Linux 3.18.20 kernel, 12 CPU cores and 20 GB memory), running on two different physical machines (Intel Xeon CPU E5-2650, 96 GB memory). The router is emulated using a physical machine (Intel Xeon CPU E5-2620, 64 GB memory) with multiple 1 Gbps NIC ports. We use Linux tool *tc* and *netem* to induce link delay and packet loss. The basic RTT in our testbed is about 50 ms and explicit congestion notification (ECN) is not enabled. The router queue length is set to 5400 KB, that is three bandwidth-delay-products (BDP) [26]. The initial *cwnd* is 10 segments [27] (for multi-path transports each sub-flow has an initial *cwnd* of 10 segments).

During the evaluation, the client sends multiple requests to the server and the server responds with certain amount of data for each request according to the requested data size. Each request and response pair is called a flow. Each MPTCP/FUSO/CoFUSO flow may use the two physical paths in our testbed (P1 and P2). This scenario emulates a mobile device accessing a Web server through two access links.

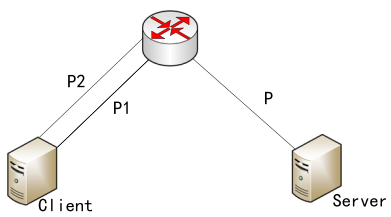


Fig. 8 Testbed topology

6.1.1 Targeted packet-drop

First, we manually drop some packets in a flow with certain size to see how CoFUSO can speedup the loss recovery through coding, compared to FUSO.

Setup Specifically, we evaluate two sample cases. In each case, the client requests a flow with 29 packets from the server. To clearly show the effect of coding recovery, before our experiments, we generate 1 KB data to warm-up each connection and wait for an idle time to reset the initial window, thus to activate all the sub-flows. We enable 3 sub-flows. Therefore, when the flow data has all been sent out, there will be one spare chance in the initial window for proactive recovery. Figure 6 illustrates such condition.

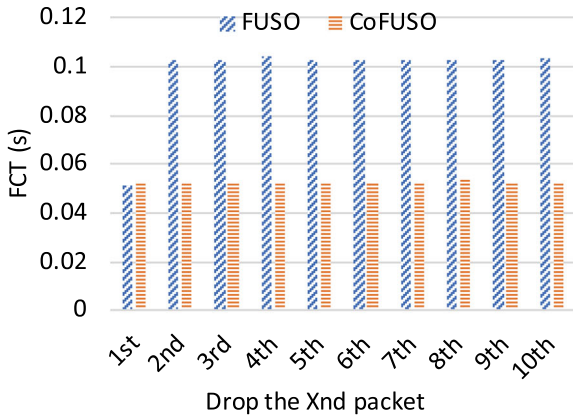
In the first case, we manually drop one of the 10 packets transmitted by the first sub-flow in turn. In order to show how coding recovery works, we ensure that both CoFUSO and FUSO always find this sub-flow as the worst sub-flow for proactive recovery. In the second case, we randomly drop one of the 29 packets on the three sub-flows, and no longer control the selection of the worst sub-flows (using the path selection algorithm described in Sect. 4.2.4).

Results Figure 9(a) and (b) show the FCT under the two packet-drop cases. As shown in Fig. 9(a), since FUSO simply retransmits the first un-ACKed packet on the sub-flow, it only matches the performance of CoFUSO when the first packet is actually lost. For other packet-drop cases, FUSO needs another RTT to recovery the lost packet. On the contrary, CoFUSO can always recover the right lost packet in the first RTT by encoding and decoding. As such, for most cases, CoFUSO has about 50% lower FCT than FUSO.

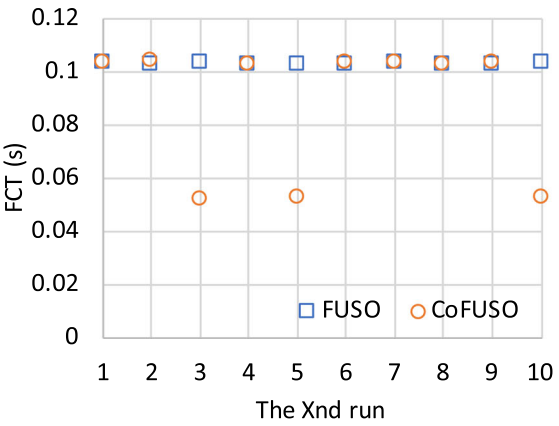
Figure 9(b) shows the FCT of 10 runs under the second case, i.e., randomly dropping a packet in the whole flow. For 70% runs, CoFUSO does not improve the loss recovery. This is because the selection of the worst sub-flow may be not accurate due to the lack of history loss information (Sect. 4.2.4). Note that in real scenarios, CoFUSO can continuously monitor history loss information during transmission, which gradually improves the accuracy of path selection (see results for larger flows in Sect. 6.1.2). For the other 30% runs, where CoFUSO selects the right sub-flow for proactive recovery, it reduces the FCT by $\sim$50% compared to FUSO.

6.1.2 Empirical traffic under random lossy condition

Next, we evaluate CoFUSO under more realistic scenarios.

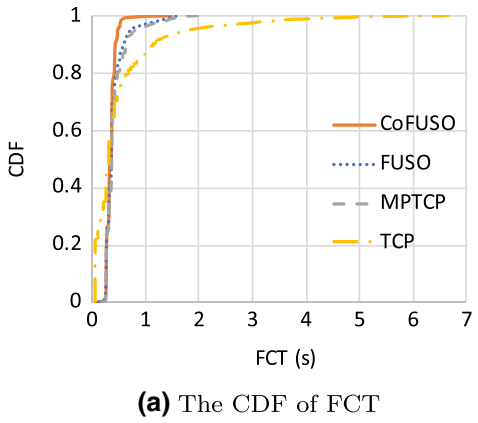


(a) Drop one certain packet in the first sub-flow.

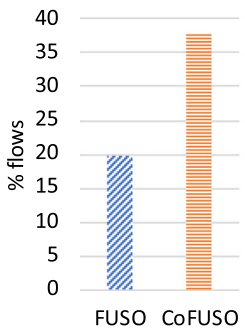


(b) Drop one random packet among all sub-flows.

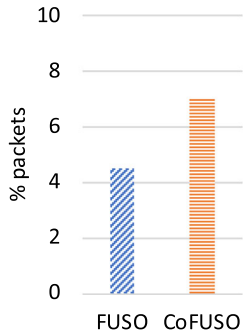
Fig. 9 Flow completion time under targeted packet-drop



(a) The CDF of FCT



(b) Ratio of flows successfully recovered by proactive recovery packets.



(c) Ratio of recovery packets that have successfully recovered data packets.

Fig. 10 The testbed results under empirical traffic and random lossy condition

Setup According to our measurement in Sect. 3.2, we induce 2% random packet loss on path P2 to emulate a WiFi link, and do not manually induce packet loss on path P1 to emulate a 4G link. The client generates $\sim$3000 requests to the server, with the requesting data size sampled from the real workload measured in Baidu (Fig. 3). Flows have no overlap (the next one starts after the former ends), and each flow uses a separate connection without warm-up. The random loss only occurs on data packets from the server to the client. We compare the FCT of using CoFUSO, FUSO, MPTCP and TCP in this scenario. Note that the connection setup/disconnect time is not counted in a flow's FCT. For TCP, a flow will randomly choose one of the two paths among P1 and P2 to transmit data. We enable 3 sub-flows for CoFUSO/FUSO/MPTCP, where P1 has two on it(Mobile devices prefer to use 4G links for Web traffic, so we set up two sub-flows on P1) and P2 has one.

Results Figure 10(a) shows the CDF of FCT for each method. Compared with FUSO, MPTCP and TCP, the 99th percentile FCT of CoFUSO is about 58.8%, 61.3% and 86.3% lower, respectively. Note that TCP has shorter FCT in the low percentiles because some of the TCP flows have not traversed the lossy link P2. Furthermore, CoFUSO reduces the average FCT by $\sim$12%-32% compared to the other three schemes.

To show the benefit of coding more clearly, we also compare CoFUSO with FUSO on how many extra recovery packets have been transmitted before they hit the lost packet. Figure 4 shows that in CoFUSO about 60% of flows have recovered a lost packet only by sending 4 or less recovery packets, which is about 1.6x higher than FUSO ($\sim$37% flows have recovered packets using $\leq$ 4 recovery packets). Figure 10(b) and (c) show the ratio of flows having lost packets successfully recovered by proactive recovery packets (The number of flows which successfully recover data packets divided by the total number of flows) and the ratio of recovery packets that have successfully recovered data packets (The number of recovery packets which successfully recover data packets divided by the total number of recovery packets), respectively. Since

CoFUSO improves the accuracy of proactive recovery, it has successfully recovered ∼18% more flows than FUSO (Fig. 10b), and ∼2.5% more recovery packets are useful, i.e., having recovered some data packets (Fig. 10c). In contrary, FUSO misses many chances of proactive recovery due to transmitting the wrong packets, so many of the lost packets are recovered by the ordinary loss recovery after receiving duplicate ACKs.

Note that, although in this scenario there are still many recovery packets in CoFUSO/FUSO that are useless, since they do not recover any data packet, these redundancies do not impair the flow FCT since CoFUSO/FUSO prioritize new data transmission over loss recovery (Sect. 4.4). Moreover, since CoFUSO/FUSO strictly follow congestion control, the redundancy will be automatically throttled if the network is congested. See results under congestion scenarios in Sect. 6.2.3 for more details.

6.1.3 Processing overhead

Finally, we evaluate the processing overhead of CoFUSO.

Setup We use the same settings as above experiment in Sect. 6.1.2, except incurring no packet loss on network links.

Results Figure 11 shows the CDF of FCT both in FUSO and CoFUSO. Results show that in terms of processing delay, the encoding/decoding process in CoFUSO almost incurs no overhead in a 1Gbps-bandwidth and 50ms-RTT network. This comes from our simple XOR coding implementation. We also observe no explicit difference on the CPU utilization for CoFUSO and FUSO, so we omit the result figure of CPU utilization here.

6.2 Simulation

To evaluate more complex conditions, we build a larger simulated network as shown in Fig. 12. There are 20 hosts with 10 (H1–H10) connected to router R1 and 10 (H11–H20) connected to S10. There are 10 routers in the whole network, forming multiple paths between hosts H1–H10 and H11–H20. The basic host-to-host RTT is 48ms and

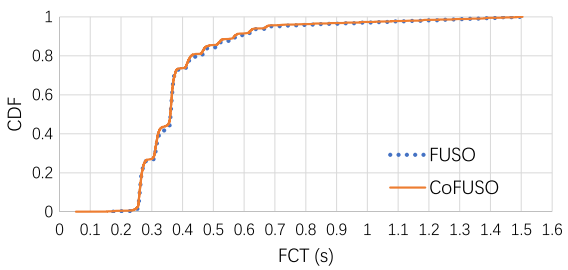


Fig. 11 Processing overhead of CoFUSO: FCT CDF under no lossy condition

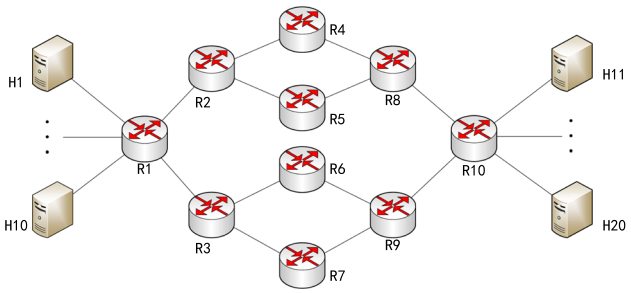


Fig. 12 The basic simulation topology

each link's bandwidth is 300 Mbps. ECN is not enabled. The router queue length is set to three bandwidth-delay-products (BDP) [26]. The initial *cwnd* is 10 segments [27] (for multi-path transports each sub-flow has an initial *cwnd* of 10 segments). We enable 5 sub-flows for CoFUSO/FUSO/MPTCP. Each sub-flow or single-path flow is randomly hashed on the physical links through equal-cost-multi-path (ECMP) algorithm [28]. For the coding parameters in CoFUSO, K_{max} is set to unlimited and M_{max} is 3, and $\alpha_1, \beta_1, \alpha_2, \beta_2$ are all set to 0.5.

During the evaluation, hosts H11–H20 send flows to hosts H1–H10, with the flow size sampled from Baidu workload (Fig. 3). We randomly pick 4 links between routers R2–R9 and induce random-drop on data packet through them.

6.2.1 Schemes compared

Besides FUSO and MPTCP, we compare the following schemes with CoFUSO in our simulation experiments. We implement all the following schemes in ns-2 [25] simulator.

Reactive [29] The latest single-path TCP enhancement scheme using prober to accelerate loss recovery. The sender transmits one more packet after approximately twice the smoothed RTT 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.

Corrective [1] The latest single-path TCP enhancement scheme using both a 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 Reactive. According to the authors' recommendation [1], we set the coding timebin to be 1/4 RTT and the maximum coding block to be 16.

RepFlow [9] A simple multi-path latency improvement scheme by proactively transmitting two duplicated flows.

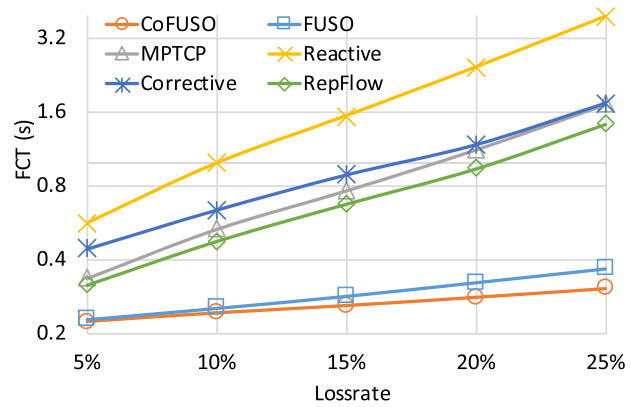
We have implemented RepFlow in the application layer according to [9].

6.2.2 Random loss

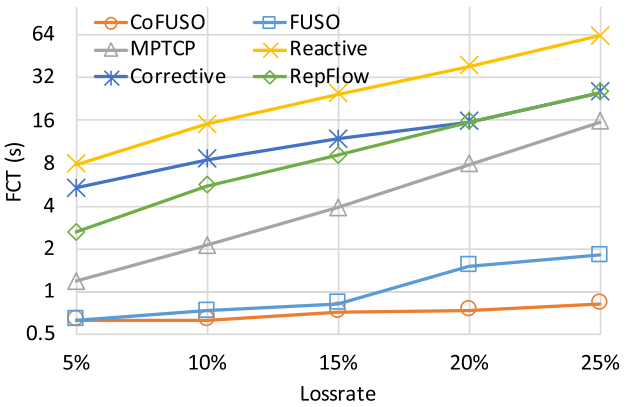
First, we evaluate CoFUSO 's performance under more complex lossy condition, and compared with multiple latest loss recovery schemes.

Setup We choose two hosts H1 and H11 under routers R1 and R10, respectively, and let H11 send flows to H1. The H11 generates 3000 flows to the server. We incur no congestion in this scenario, i.e., flows have no overlap (the next one starts after the former ends). Each flow uses a separate connection without warm-up. We induce various random loss rate (5–25%) on the links.

Results Figure 13(a) and (b) show the average and 99th percentile FCT under various loss rates. Thanks to coding, compared to FUSO, CoFUSO reduces the average FCT by $\sim$2.5–16.9%, and the 99th percentile FCT by $\sim$1–54.5%, under various loss rates.



(a) The average FCT (log scale)



(b) The 99^{th} percentile FCT (log scale)

Fig. 13 The average and 99th percentile FCT under various loss rates

Benefiting from proactive multi-path loss recovery, both CoFUSO and FUSO perform significantly better than other single-path loss recovery mechanisms (Reactive and Corrective). Moreover, CoFUSO and FUSO also perform much better than RepFlow, because there is fairly large chance for the two replicated flows in RepFlow to be hashed to the same lossy link. Despite utilizing multiple paths, MPTCP performs inferiorly because it does not conduct proactive loss recovery.

6.2.3 With congestion

Next, we consider scenarios with congestion.

Setup The 10 hosts under R10 (H11–H20) each send 50 flows concurrently to one different host under R1 (H1–H10), respectively. The flow inter arrival time obeys the Poisson process. We incur different congestion level by adjusting the flow interarrival time. Specifically, we vary the network load from 10% to 60%. The load is the ratio of the aggregate data generating speed *to* the whole bandwidth of all the sending hosts' (H11–H20) access links to the network. Since there is high oversubscription in the network (10 access links from the host and only 2 links to the network), these are actually very high loads for the network. In this experiment, we set the link loss rate to be 10%.

Results Figure 14(a) and (b) show the average and 99th percentile FCT under different congestion level. Since the congestion will automatically throttle the chance of proactive recovery, this makes CoFUSO perform similarly to FUSO. But CoFUSO still has $\sim$1.5–5% shorter average FCT than FUSO. The tail performance of CoFUSO and FUSO are almost the same when the load is high, since congestion may lead to multiple successive packet losses, which typically can not be recovered by a few coding packets.

Schemes such as RepFlow using aggressive loss recovery have much worse performance when the network is congested, because the aggressiveness added even increases the packet loss.

6.2.4 CoFUSO deep dive

Finally, we dive into several design details of CoFUSO, and evaluate CoFUSO 's performance with various settings. Specifically, we evaluate CoFUSO with:

1. Dynamic and fixed coding rate
2. Different number of parities
3. Different number of sub-flows

Dynamic and fixed coding rate To demonstrate the effectiveness of our dynamic coding (Sect. 4.2.2), we compare our CoFUSO with dynamic coding rate with a

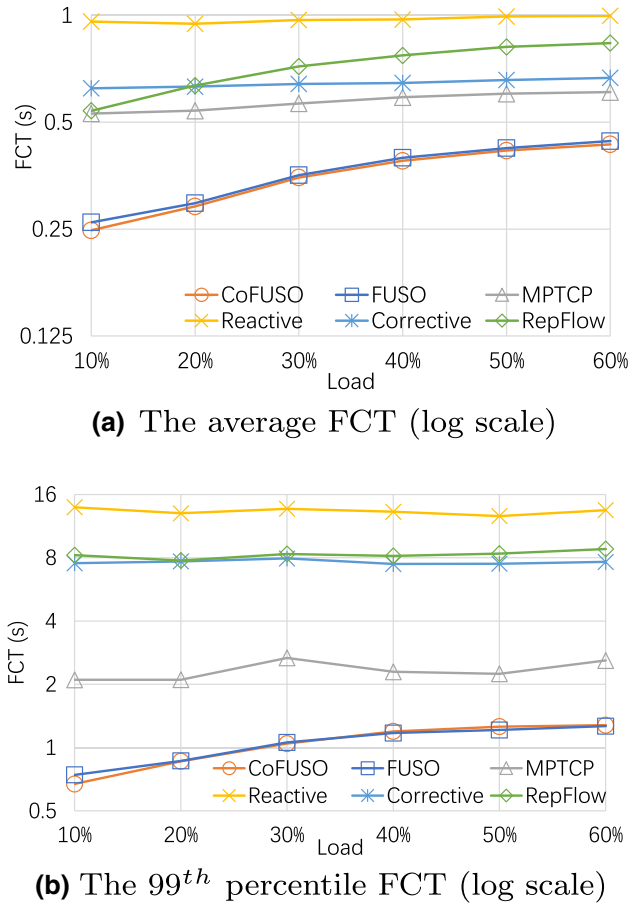


Fig. 14 The average and 99th percentile FCT under congestion

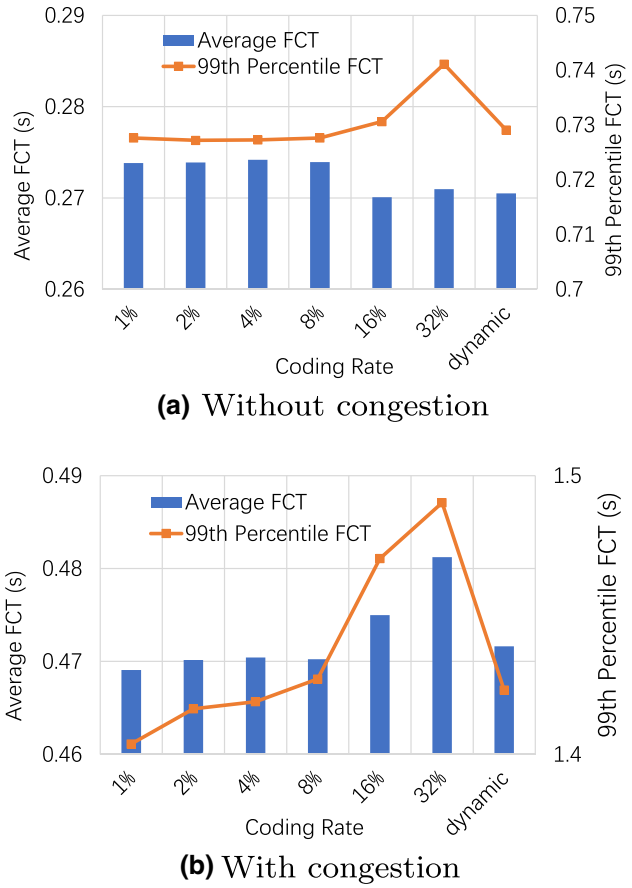


Fig. 15 The average FCT and 99th percentile FCT using various coding rates under both no-congestion and congestion conditions

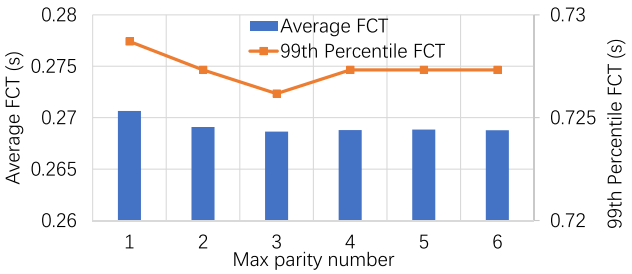
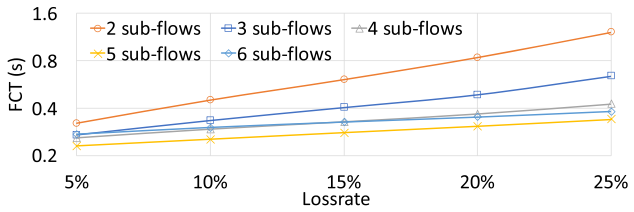


Fig. 16 The average FCT and 99th percentile FCT using different number of parities

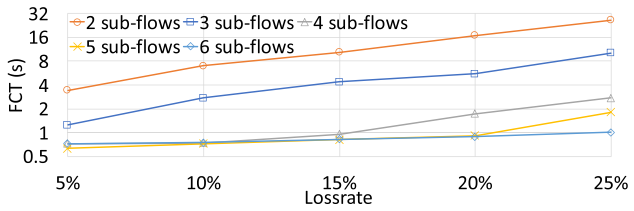
version of CoFUSO using fixed coding rate, i.e., generating fixed number of parities for fixed number of unACKed packets. Specifically, we compare with various coding rates including 1%, 2%, 4%, 8%, 16%, 32%. The coding rate is defined as the ratio of the parity number to the block size. We examine the average and 99th percentile FCT under scenarios without congestion and with congestion. In both conditions, the random loss rate is 15%. The load is 70% in the congestion scenario. Figure 15(a) and (b) show the results under no-congestion and congestion, respectively. Our dynamic coding rate can well adapt to various network conditions. On the contrary, neither a high or low fixed coding rate can offer good performance under both no-congestion and congestion conditions.

Different number of parities In former simulations, we set the maximum number of parities generated in a coding block (M_{max}) to 3. Now we change M_{max} from 1–6 to evaluate its impact on CoFUSO 's performance. We use the same random loss settings as in Sect. 6.2.2, and set the loss rate to 15%. Note that we do not incur computation delay for encoding/decoding in our simulation. Figure 16 shows the average FCT and 99th percentile FCT using different M_{max}. The average performance are very similar for different M_{max} values, but M_{max}=3 gives the best 99th percentile FCT. Since generating more parities does not improve the loss recovery performance but increases computation overhead, we choose M_{max}=3 in our simulation experiments and M_{max}=1 in the testbed.

Different number of sub-flows Now we evaluate the impact of the number of sub-flows. Specifically, we use the same settings in Sect. 6.2.2, and vary the number of sub-flows in CoFUSO from 2–6. Figure 17(a) and (b) show the average and 99th percentile FCT under various loss rates. 5 sub-flows perform better than other conditions both in



(a) The average FCT (log scale)



(b) The 99^{th} percentile FCT (log scale)

Fig. 17 Various number of sub-flows

average and 99th percentile. On one hand, more sub-flows give more chances for CoFUSO to avoid lossy links; on the other hand, too many sub-flows will increase the flow burstiness due to MPTCP's congestion control [30]. As such, we chose 5 in our simulation.

7 Conclusion

Inefficient loss recovery is a well-known problem that impairs TCP performance especially for short Web flows. Although FUSO shows a great potential on addressing this problem by conduct proactive recovery over multi-path in data center networks, we show through comprehensive analysis and testbed experiments that, its simple proactive retransmission is not effective for the Internet scenario. This paper presents CoFUSO, which improves the recovery accuracy using erasure codes. We implement CoFUSO in Linux kernel with $\sim$2K lines of code. Both testbed and simulation experiments show that through coding, CoFUSO can further reduce the average and 99th percentile FCT of FUSO by up to $\sim$17% and $\sim$59% respectively.

Acknowledgements We thank Xiaoning Zhan for his help on refining the paper. This work was supported in part by the National Natural Science Foundation of China under Grant 6187060280, in part by the Tencent Rhino-Bird Open Research Fund, and in part by the Fundamental Research Funds for the Central Universities.

References

1. Flach, T., Dukkipati, N., Terzis, A. et al. (2013). Reducing web latency: the virtue of gentle aggression. In *Proceedings of the ACM SIGCOMM, 2013 conference on SIGCOMM* (pp. 159–170). Hong Kong, China: ACM.
2. Chen, Y., Mahajan, R., Sridharan, B., et al. (2013). A provider-side view of web search response time. *ACM SIGCOMM Computer Communication Review*, *43*(4), 243–254.
3. Liu, D., Zhao, Y., & Sui, K., et al. (2016). FOCUS: shedding light on the high search response time in the wild. In *The 35th annual IEEE international conference on computer communications* (pp. 1–9). San Francisco, CA, USA: IEEE.
4. Arapakis, I., Bai, X., & Cambazoglu, B.B. (2014). Impact of response latency on user behavior in web search. In *The 37th international ACM SIGIR conference on research & development in information retrieval*. New York, USA: ACM. 103 $\sim$ 112.
5. Zhou, J., Wu, Q., & Li, Z., et al. (2015). Demystifying and mitigating TCP stalls at the server side. *The 11th ACM conference on emerging networking experiments and technologies* (pp. 1–13). Heidelberg, Germany: ACM.
6. Chen, G., Lu, Y., & Meng, Y. et al. (2016). Fast and cautious: leveraging multi-path diversity for transport loss recovery in data centers. In *USENIX conference on usenix annual technical conference* (pp. 29–42). Berkeley, CA, USA: USENIX.
7. Chen, G., Lu, Y., Meng, Y., et al. (2018). FUSO: fast multi-path loss recovery for data center networks. *IEEE/ACM Transactions on Networking*, *26*(3), 1376–1389.
8. Reed, S., & Slolmon, G. (1960). Polynomial codes over certain finite fields. *Society of Industrial and Applied Mathematics*, *8*(2), 300–304.
9. Xu, H., & Li, B. (2013). RepFlow: minimizing flow completion times with replicated flows in data centers. In *IEEE INFOCOM 2014-IEEE conference on computer communications* (pp. 1581–1589). Toronto, ON, Canada: IEEE.
10. Fan, X., Li, H., & (2014). Design and implementation of TCP with network coding. In *The 2014 2nd international conference on systems and informatics* (pp. 570–575). Shanghai, China: IEEE.
11. Bao, W., Shah-Mansouri, V., & Wong, V. W. S., et al. (2012). TCP VON: joint congestion control and online network coding for wireless networks. In *2012 IEEE global communications conference* (pp. 125–130). Anaheim, CA: IEEE.
12. Sun, J., Zhang, Y., & Tang, D., et al. (2015). TCP-FNC: a novel TCP with network coding for wireless networks. In *2015 IEEE international conference on communications* (pp. 2078–2084). London: IEEE.
13. Ageneau, P., Boukhatem, N., & Gerla, M., (2017). Practical random linear coding for MultiPath TCP: MPC-TCP. In *2017 24th international conference on telecommunications* (pp. 1–6). Limassol: IEEE.
14. Li, M., Lukyanenko, A., & Cui, Y. (2012). Network coding based multipath TCP. In *2012 proceedings IEEE INFOCOM workshops* (pp. 25–30). Orlando, FL: IEEE.
15. Cui, Y., Wang, L., Wang, X., et al. (2015). FMTCP: a fountain code-based multipath transmission control protocol. *IEEE/ACM Transactions on Networking*, *23*(2), 465–478.
16. Lu Y, Chen G, & Li B, et al. (2018). Multi-path transport for RDMA in datacenters. In *15th USENIX symposium on networked systems design and implementation (NSDI 18)* (pp. 357-371). Renton, USA: USENIX.
17. Chen, G., Lu, Y., Li, B., et al. (2019). MP-RDMA: enabling RDMA with multi-path transport in datacenters. *IEEE/ACM Transactions on Networking*, *27*(6), 2308–2323.
18. Marketing Land. (2016). Mobile devices now driving 56 percent of traffic to top sites. https://marketingland.com/mobile-top-sites-165725. Accessed 23 Feb 2016.
19. Apple. (2016). MUse multipath TCP to create backup connections for iOS. https://support.apple.com/en-us/HT201373. Accessed 10 Aug 2016.

20. Huawei. (2018). Exclusive: how link turbo works on the Honor V20. https://www.gizmochina.com/2018/12/26/exclusive-how-link-turbo-works-on-the-honor-v20/. Accessed 26 Dec 2018.
21. Ford, A., Raiciu, C., & Handley, M., et al. (2013). TCP extensions for multipath operation with mul-tiple addresses. *RFC 6824*.
22. Nie, X., Zhao, Y., & Pei, D., (2018). Reducing web latency through dynamically setting TCP initial window with reinforcement learning. In *2018 IEEE/ACM 26th international symposium on quality of service* (pp. 1–10). Banff, AB, Canada: IEEE.
23. Paasch, C., Detal, G., & Nalayama, K., et al. (2019). Exclusive: How Link Turbo works on the Honor V20. https://github.com/multipath-tcp/mptcp. Accessed 5 August 2019.
24. Chen G. (2016). FUSO. https://github.com/1989chenguo/FUSO. Accessed 13 June 2016.
25. Legout, A., Amir, E., Balakrishnan, H., et al. (2011). The network Simulator-ns-2. http://www.isi.edu/nsnam/ns/. Accessed 4 Nov 2011.
26. Wu, H., Ju, J., & Lu, G., et al. (2012). Tuning ECN for data center networks. In *The 8th international conference on emerging networking experiments and technologies* (pp 25–36). Nice, France: ACM.
27. Dukkipati, N., Refice, T., Cheng, Y., et al. (2010). An argument for increasing TCP's initial congestion window. *ACM SIGCOMM Computer Communication Review*, *40*(3), 26–33.
28. Hopps, Christian E. (2000). Analysis of an equal-cost multi-path algorithm. *RFC 2992*.
29. Dukkipati, N., Cardwell, N., & Cheng, Y., et al. (2013). Tail loss probe (TLP): an algorithm for fast recovery of tail losses. https://tools.ietf.org/html/draft-dukkipati-tcpm-tcp-loss-probe-01. Accessed 25 Feb 2013.
30. Alizadeh, M., Edsall, T., & Dharmapurikar, S., et al. (2014). CONGA: distributed congestion-aware load balancing for datacenters. In *2014 ACM conference on SIGCOMM* (pp. 503–514). New York, USA: ACM.

Yi Liu is a master student at Hunan University, China. She will join Huawei as a software engineer starting from 2020. Her research interest is computer networking.

Guihua Zhou is a master student at Hunan University, China. He got his B.S. degree from Xiangtan University, China, in 2018. His research interests are networked system and data center networking.

Guo Chen is an associate professor at Hunan University, China. He received his Ph.D. degree from Tsinghua University in 2016. Before joining Hunan University, he worked as an associate researcher in Microsoft Research, Asia, from 2016 to 2018. His current research interests lie broadly in networked systems, with a special focus on data center networking.