LinkGuardian: Mitigating the impact of packet corruption loss with link-local retransmission

Raj Joshi, Qi Guo, Nishant Budhdev, Ayush Mishra, Mun Choon Chan, Ben Leong
School of Computing, National University of Singapore
Singapore

ABSTRACT

Packet corruption loss is a serious problem in datacenter networks. A large-scale study by Microsoft reported that the number of packets lost due to corruption is comparable to those lost due to congestion. Previous attempts to mitigate the impact of packet corruption loss seek to avoid the faulty links by routing around them, at the cost of reduced link capacities and disruption to the rest of the network.

In this paper, we investigate the feasibility and tradeoffs of the classical loss recovery strategy of link-local retransmissions in the context of datacenter networks. We present the design and implementation of *LinkGuardian*, a dataplane-based protocol that detects the packets lost due to corruption and simply retransmits them out-of-order. Our preliminary results show that a naïve out-of-order retransmission strategy is effective in mitigating the impact of packet corruption loss for both throughput-sensitive and latency-sensitive flows. Our long-term goal is to extend LinkGuardian so that the end hosts can be made completely oblivious to packet corruption losses in the network.

CCS CONCEPTS

• **Hardware → Failure recovery, maintenance and self-repair**;
• **Networks → In-network processing**; **Physical links**.

KEYWORDS

packet corruption, link failure, retransmission, programmable switches

ACM Reference Format:
Raj Joshi, Qi Guo, Nishant Budhdev, Ayush Mishra, Mun Choon Chan, Ben Leong. 2022. LinkGuardian: Mitigating the impact of packet corruption loss with link-local retransmission. In *6th Asia-Pacific Workshop on Networking (APNet 2022), July 1–2, 2022, Fuzhou, China.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3542637.3542643

1 INTRODUCTION

In modern datacenter networks, switch-to-switch links are typically optical [27, 30] as they can support high link speeds (10-400 Gbps) over long distances compared to electrical links. However, optical links are susceptible to packet corruption, as the optical receiver sometimes fails to correctly decode the transmitted bits. Optical decoding errors can occur due to a variety of reasons such as fiber bending, connector or fiber tip contamination by airborne dirt particles, decaying laser transmitters, etc. [27, 30]. When an optical decoder decodes the bits of a packet incorrectly, the Ethernet frame checksum (FCS) fails and the receiving MAC drops the packet. With tens of thousands of optical links in a datacenter, packet corruption loss can be significant.

A large-scale study consisting of 350K links across 15 Microsoft datacenters reported that the number of packets lost due to link corruption is on par with the number of packets lost due to congestion [30]. It is well known that packet loss (regardless of the cause) hurts application performance leading to a revenue loss of millions of dollars [4, 28, 29]. Alibaba's recent study of hundreds of real-world service tickets revealed that among all the packet drops that led to network performance anomalies and affected its customers, 18% were due to packet corruption [28].

Packet corruption loss is different than congestion packet loss because it does not go away even when the end hosts reduce their transmission rates. Unless mitigated, packet corruption will continue to cause degradation to application performance and affect a cloud provider's SLAs (Service Level Agreements) [28]. With a corruption loss rate of 1%, throughput-intensive applications can suffer more than 50% degradation in throughput since the TCP senders reduce `cwnd` in response to the packet losses due to corruption [15]. Similarly, latency-sensitive applications incur increased flow completion times (FCTs) due to delays caused by the retransmissions of lost packets [27, 31]. Overall, packet corruption negatively impacts both latency-sensitive and throughput-sensitive applications.

Currently, packet corruption can only be fixed by physically repairing the corrupting link, which could take between several hours to days. Meanwhile, applications continue to suffer performance degradation [30]. Until a physical repair is carried out, one can only *mitigate* the effects of packet corruption on application performance. The most common way to do so is to disable the corrupting link(s) at the cost of reduced network capacity [27, 30]. However, the resulting re-routing of traffic can impact the rest of the network beyond the corrupting link [27]. End-to-end redundancy approaches [14, 26, 31] also incur similar impact due to overhead along the entire path and often require end-host changes which are typically beyond a cloud operator's control. It is possible to localize the impact of corruption by using link-local forward error correction (FEC) [15]. But this adds significant overhead (e.g. 15% overhead for 1% loss rate [15]) as redundancy is added to *all* the packets, even though only a small fraction of the packets are corrupted.

The drawbacks of previous approaches and the overheads of FEC can be mitigated with a straightforward approach of link-local retransmissions. It is surprising that, to the best of our knowledge, we are the first to revisit this approach. Link-local retransmissions have been studied extensively [7, 24] and widely used in wireless networks [1, 2, 17, 18]. It is likely that link-local retransmission

APNet 2022, July 1–2, 2022, Fuzhou, China

ACM ISBN 978-1-4503-9748-3/22/07.
https://doi.org/10.1145/3542637.3542643

for wired networks has not been proposed earlier because it could not be easily implemented in earlier fixed-function switches. In this paper, we show that link-local retransmissions can be easily implemented with modern dataplane-programmable switches.

LinkGuardian is our attempt at investigating the feasibility and tradeoffs of a classic loss recovery strategy. Link-local retransmissions are clearly desirable since they localize the impact of recovering from corruption only to the corrupting link. Furthermore, the overheads would be proportional since retransmissions happen only when there is packet loss. They also do not require end-host modifications and can be under a cloud operator's control.

Our long-term goal is to not only implement a link-local retransmission scheme that can fully recover from packet corruption losses, but one that also preserves packet ordering. In this paper, we present the results of our early-stage implementation that detects the packets lost due to corruption and naïvely retransmits them out-of-order. Even with this basic implementation, LinkGuardian is able to mitigate the impact of corruption for both throughput-sensitive flows and latency-sensitive flows. We found that the key reason why out-of-order retransmission works well is that TCP has a built-in "reordering tolerance" i.e. a TCP sender waits until it receives triple duplicate ACKs (or SACK-equivalent) [5, 8] before considering a packet lost. Since we are able to retransmit packets fast enough, we can effectively convert the packet loss events into reordering events, thereby preventing `cwnd` reduction from being triggered at the TCP sender.

Our current implementation is a work in progress that achieves good performance for the common case but does not work so well beyond the 95th percentile for latency-sensitive flows. We believe that this is because we are yet to implement mechanisms to deal with (i) consecutive losses, (ii) tail losses, and (iii) preserving packet ordering. Moving forward, we will extend LinkGuardian so that the end hosts can be made *completely oblivious* to packet corruption losses. Doing so would require additional packet buffering and we argue with back-of-the-envelope calculations that there is sufficient packet buffer in existing dataplane-programmable switches to support the same.

The remaining paper is organized as follows. In §2, we conduct a small measurement study that shows that a simple out-of-order retransmission scheme is sufficient to mitigate the impact of corruption packet loss on latency-sensitive and throughput-sensitive flows. In §3, we describe the design and implementation of LinkGuardian and we evaluate its performance in §4. We discuss future directions in §5 and summarize the related work in §6. Finally, we conclude in §7.

2 CASE FOR LINK-LOCAL RECOVERY

In this section, we investigate the effectiveness of the following simple link-local retransmission scheme that is implemented by 2 neighboring switches: before sending a packet on the link, the sending switch adds a sequence number to the packet and also retains a copy of the transmitted packet for potential future retransmission. Using the sequence numbers in the received packets, the receiving switch detects if there was packet corruption loss and notifies the sending switch. The sending switch immediately retransmits the lost packet with *high priority*.

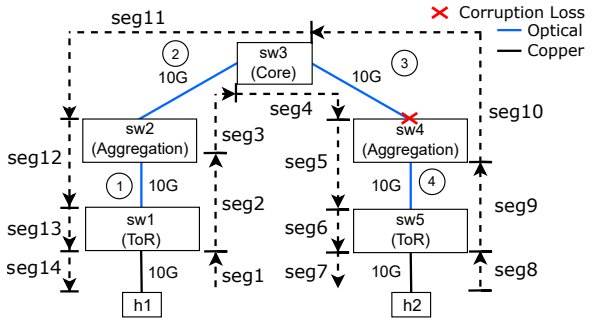

Figure 1: Testbed Setup: 3-stage Clos network with unidirectional corruption on an inter-pod path.

Recovery Delay. We define the *recovery delay* to be the time between when a packet was dropped due to corruption at a particular point in the network till when the packet is recovered back at the same point in the network. After the recovery, the packet continues on the remaining network path. This definition of recovery delay essentially captures the *additional* delay penalty incurred in delivering the lost packet to the intended receiver host. In the following subsection, we describe a simple measurement study to estimate the recovery delay for a link-local retransmission scheme implemented in the network dataplane.

2.1 Estimating the Recovery Delay

Our testbed setup, where we emulate an inter-pod path of a 3-stage Clos network connecting hosts `h1` and `h2`, is shown in Figure 1. The switches are emulated using loopback cables on a single Wedge100BF-32X Intel Tofino switch. Each host is equipped with a 10 Gbps DPDK-capable NIC. Both the hosts run Linux kernel 5.4.0-91-lowlatency on Ubuntu 20.04.3. The network's MTU is 1,500 byte (maximum packet size of 1,518 byte). In all our experiments, there is no cross traffic.

We consider a round-trip path where a packet from `h1` travels to `h2` and back. We instrument the switches to record ingress and egress timestamps which divide the RTT into 14 different time segments as shown in Figure 1. If link 3 is corrupting packets in the `sw3` →`sw4` direction, then a corrupted packet would be dropped at `sw4`. `sw4` would then detect the loss immediately after receiving a subsequent packet and then notify `sw3` with a minimum-sized packet. This loss notification would require `seg10` (64 byte) worth of time, as the time for each segment varies with packet size due to the serialization. Once `sw3` receives this packet, it will immediately retransmit the lost packet to `sw4` with *high priority*. Thus, the total recovery time for the link-local scheme would then be `seg10` (64 byte) + `seg4` (1,518 byte). In comparison, any end-host based recovery scheme would require ~1 RTT worth of recovery time considering a 64 byte (S)ACK from `h2` to `h1` and a 1,518 byte retransmitted packet from `h1` to `h2`. We use `ping` to send packets from `h1` to `h2` one at a time and record the RTT reported by `ping`. We use 64 byte and 1518 byte packet sizes and send ~1M packets each. Using the measured RTT and the timestamps from the switches, we then estimate the link-local recovery delay. Table 1 shows the distribution of the expected recovery delay.

In Table 1, the high end-host based recovery delays are due to the high end-host stack latencies and we had similar results when

Table 1: Expected recovery delay distribution (in μs) for packet corruption loss on link 3 in Figure 1

	End-host based (kernel)	End-host based (DPDK)	Link-local Retransmission
mean	32.73	33.44	2.59
50%	32.50	33.52	2.59
99%	44.00	46.25	2.60

using a kernel-bypassed (DPDK-based) ping client and responder. In contrast, the estimated link-local recovery delays are an order of magnitude smaller and do not vary much. The recovery delays are similar if a packet corruption loss were to happen on the other optical links shown in Figure 1. Note that the time segments include the switch pipeline latency which is a function of the dataplane programming used in this experiment.

Implications for TCP. For short latency-sensitive flows, corruption loss will typically increase the FCT by an amount equal to the recovery delay. Table 1 shows that the increase in FCT due to packet corruption can be potentially reduced by up to 10x. For throughput-sensitive flows, corruption loss triggers a loss event at the TCP sender causing it to reduce its `cwnd` and thereby degrade throughput. However, we note that a lost packet does not immediately trigger a loss event. A TCP sender has some "reordering tolerance" because it generally waits until it receives triple duplicate ACKs (or SACK-equivalent) [5, 8] before considering a packet to be lost. In other words, the TCP sender waits until ~3 MSS subsequent bytes are delivered to the TCP receiver after the lost packet. Since the inter-packet gap for a TCP flow saturating a 10 Gbps NIC is ~1.23 μs, the TCP's "reordering window" of 3 packets is ~3.69μs. From Table 1, we see that the average estimated link-local recovery delay is ~2.59 μs which is less than ~3.69μs. This means that it is possible that the lost packet might be successfully retransmitted before the next 3 MSS bytes. In such a case, the TCP sender would observe packet reordering instead of packet loss, and it would not reduce its `cwnd`. Note that since the lost packet is retransmitted with high priority, it *preempts* the subsequent packets of the TCP flow and thereby helps in achieving link-local recovery within the "reordering window".

3 SYSTEM DESIGN

In this section, we present the design and implementation of LinkGuardian. A network equipped with LinkGuardian functions normally (with LinkGuardian deactivated) as long as no link is found to be corrupting packets beyond an operator-specified threshold loss rate (usually ~10^{-6} [30]). A state-of-art control plane-based technique [15, 30] is used to monitor the links and when a link is found to be corrupting packets beyond the specified threshold, LinkGuardian is activated on the link. LinkGuardian can be configured to protect all traffic or only a select class of traffic.

3.1 Protocol Overview

Since link corruption is typically unidirectional [30], LinkGuardian can be viewed as a protocol running between a "sender switch" and a "receiver switch". For each packet that is transmitted on the corrupting link, the sender switch adds a monotonically increasing sequence number and creates a copy of the packet for buffering.

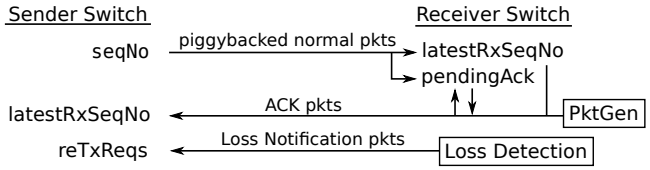

Figure 2: Protocol Overview: Arrows indicate the state variables read and updated by different packet types.

The receiver switch checks if the sequence numbers of the arriving packets are in order.

In the absence of any packet loss, the sequence numbers would be in order and the receiver switch maintains a steady *stream* of ACKs informing the sender switch of the latest received sequence number. On receiving such ACKs, the sender switch drops the buffered packets with sequence numbers less than or equal to the latest received sequence number. In the event of packet corruption, the receiver switch detects which packet(s) was/were lost based on the gaps in the sequence numbers of the arriving packets. On detecting a loss, the receiver sends a high-priority *loss notification* to the sender switch informing of the lost packets and also the latest received sequence number. The sender switch then retransmits (with high priority) the lost packets and drops the *remaining* buffered packets with sequence numbers less than or equal to the latest received sequence number.

LinkGuardian is implemented as 2 components: (i) loss detection and notification, and (ii) sender-side buffering and retransmission. Our design assumes that the switch supports capabilities such as mirroring, multicast and recirculation as defined in the Portable Switch Architecture (PSA) [16].

3.2 Loss Detection and Notification

In Figure 2, we list the state maintained by the sender and receiver switches, and the different types of packets that read and update them. The sender switch maintains a monotonically increasing `seqNo` while the receiver switch maintains a `latestRxSeqNo` which records the latest received `seqNo`. A copy of the `latestRxSeqNo` is also maintained at the sender switch, which the receiver switch keeps updating. The sender switch also maintains a lookup table called `reTxReqs`, which records the sequence numbers of the packets whose retransmission is requested by the receiver switch.

For each normal packet that is to be protected on the corrupting link, the sender switch adds the current `seqNo` to the packet (using a custom header) and then increments `seqNo` by 1. The sender switch makes a copy of the packet along with the added sequence number and buffers it until the receiver switch notifies that the packet was received successfully (buffering-related details in §3.3). On the receiver switch, when a protected packet is received, it updates the `latestRxSeqNo` to the sequence number in the packet and also sets the `pendingAck` to 1. `pendingAck` set to 1 denotes that the copy of `latestRxSeqNo` on the sender switch is yet to be updated.

No Loss Scenario. Since the corruption loss rates are typically low, we generally expect no losses to occur on the link. In a no-loss scenario, the `latestRxSeqNo` on the receiver switch would increase by 1 on receiving a protected packet. On every update of the `latestRxSeqNo`, the receiver switch updates the `latestRxSeqNo`

on the sender switch so that the sender switch can drop the buffered packets that are successfully delivered. Doing this update timely can ensure that LinkGuardian's use of the packet buffer at the sender switch is kept to a minimum.

To achieve this, we use the packet generator in the receiver switch's dataplane to generate ACK packets at a fixed interval (δ) which carries the updated `latestRxSeqNo` to the sender switch. To ensure timely update while incurring minimal overhead in the reverse direction, an ACK packet is sent only if `pendingAck` is 1 and then the `pendingAck` is reset to 0. This mechanism ensures that an ACK packet is sent only if the `latestRxSeqNo` on the receiver switch was updated after an ACK packet was previously sent.

Loss Scenario. When a protected packet gets corrupted and is dropped by the receiving MAC, the receiver switch observes that the `latestRxSeqNo` is incremented by more than 1. It then generates a new "loss notification" packet which contains the missing sequence number as well as the recent value of the `latestRxSeqNo`. This loss notification packet is sent to the sender switch through a high-priority queue to ensure timely recovery. On reaching the sender switch, the missing sequence number is added to the lookup table `reTxReqs`.

3.3 Sender-side Buffering & Retransmission

As described in §3.2, the receiver switch ensures that the copy of `latestRxSeqNo` on the sender switch is up-to-date and it additionally updates the lookup table `reTxReqs` in case of a packet corruption loss. Meanwhile, the sender switch buffers a copy of the protected packet along with its sequence number. This packet buffering is realized through *recirculation*, i.e., the buffered copy of the protected packet is sent to the recirculation port of the switch dataplane. Each time the buffered packet completes a recirculation loop, the sender switch checks two conditions: (i) if the buffered packet's sequence number is less than or equal to the updated `latestRxSeqNo`; and (ii) whether the buffered packet's sequence number is present in the lookup table `reTxReqs`.

If the first condition is not met, it means that we do not yet know if the packet was successfully delivered. In this case, the sender switch continues to buffer the packet via recirculation. If the first condition is true while the second condition is false, then it means that the packet was successfully delivered and so the sender switch drops the buffered packet. Finally, if both the conditions are true, then the sender switch retransmits the buffered packet through a high-priority queue and clears the packet's sequence number from the `reTxReqs` table.

Buffer requirement. Notice that the ACK interval (δ) is the *expected* time for which a packet would be buffered. During this time, if the link is fully utilized, then the additional bytes that would be added to the buffer would be the $\delta \times C$, where C is the link capacity. This is the expected buffer requirement which can be kept low by keeping the ACK interval small.

Handling Retransmission Losses. It is plausible that a packet retransmitted could also be dropped due to corruption. LinkGuardian handles this situation by retransmitting 2 copies of the buffered packet to improve the odds that the retransmission will be successful. However, 2 copies are required only when the link's corruption loss rate is $>10^{-3}$ such that the effective loss rate is $\leq 10^{-6}$.

Table 2: Steady-state Throughput (Gbps) on the lossy link when protected by Wharf [15] or LinkGuardian

Loss Rate →	0 (Baseline)	10^{-5}	10^{-4}	10^{-3}	10^{-2}
CUBIC	9.49	9.48	7.28	3.43	1.33
+ Wharf [15]	n/a	9.13	9.13	9.13	7.91
+ LinkGuardian	9.47	9.47	9.47	9.46	9.28
DCTCP	9.49	9.46	7.88	3.82	1.66
+ LinkGuardian	9.47	9.47	9.47	9.46	9.29

3.4 Implementation

We implement LinkGuardian on an Intel Tofino programmable switch in ~900 lines of P4 code each for the sender and receiver switch's protocol logic. For each protected packet, the sender switch adds a 3-byte custom header that contains a 16-bit `seqNo` and the packet type (original or retransmitted). The ACK packet is a 64 byte packet generated by the receiver switch's packet generator every 5 μs. Since the sequence number is of a fixed bit width, it will periodically wrap around and restart at zero. Therefore, to correctly compare the sequence numbers, we need an additional bit – which we call an "era bit" – that toggles between 0 and 1 after each wrap around. This "era bit" is sufficient for us to detect when the sequence number overflows and react accordingly.

The sender switch uses egress mirroring to create a copy of the protected packet. For creating multiple copies of the retransmitted packet (in case of a high loss rate), it uses the multicast primitive. For generating the loss notification, the receiver switch uses ingress mirroring.

4 EVALUATION

In this section, we attempt to answer the following questions: (i) How well does LinkGuardian mitigate the effects of packet corruption for latency-sensitive and throughput-sensitive flows? and (ii) What is the overhead incurred? For evaluation, we use the same testbed setup shown in Figure 1 except that *sw*3 and *sw*4 are two independent physical switches functioning as LinkGuardian sender and receiver switches, respectively. In all experiments, `h1` uses `iperf` to send CUBIC or DCTCP [4] flows to `h2` with TSO and SACK enabled. RTO_{min} is set to 1 ms as is the current practice [21, 23], while the ECN marking threshold for DCTCP is set to 100 KB [11]. On `sw4`, we use Tofino's random number generator (similar to Wharf [15]) to emulate unidirectional corruption loss at different rates in the `sw3` →`sw4` direction. To accurately track the "affected" TCP flows, `sw4` redirects the to-be-dropped packets to a monitoring server.

4.1 Mitigating the effects of Corruption

For throughput-sensitive CUBIC flows, we compare LinkGuardian to Wharf [15], a link-local FEC scheme. We reproduce Wharf's results *numerically* by picking the FEC parameters that gave the best throughput for each loss rate (c.f. Figure 8 in [15]). We could not reproduce Wharf's results experimentally because we did not have access to the required FPGA hardware.

In Table 2, we present the steady-state throughput achieved by DCTCP/CUBIC for different loss rates (measured over 90 seconds). We see that LinkGuardian achieves consistently better throughput for all loss rates. Further, the throughput for LinkGuardian degrades

Table 3: FCT distribution (in μs) for 45 KB "affected" DCTCP flows with and without LinkGuardian (LG)

Loss rate	0	10^{-4}		10^{-3}		10^{-2}	
	(Base)	Lossy	LG	Lossy	LG	Lossy	LG
min	113	198	174	193	143	180	155
mean	197	701	389	707	375	730	401
50%	180	423	268	419	258	427	267
90%	311	2016	442	2421	424	2472	434
95%	315	3182	500	3216	455	3294	484
99%	329	4114	3412	4192	3540	4271	3682

gradually with increasing loss rate as LinkGuardian does not add constant redundancy, unlike Wharf. At 10^{-2} (1%) loss rate, LinkGuardian retransmits 2 copies for every lost packet and therefore incurs a proportional degradation of ~2% compared to the baseline.

Next, for latency-sensitive (≤100 KB [3]) flows, we consider 3 representative sizes of 15 KB, 45 KB, and 105 KB as these represent flows requiring 1, 2, and 3 RTTs to complete (under no loss conditions). Table 3 shows the distribution of FCTs (in μs) for 45 KB DCTCP flows that were affected by corruption loss. Compared to a lossy link, we see that LinkGuardian reduces the 50th and 95th percentile FCT by ~35% and ~85% respectively, for all the loss rates. However, for the same percentiles, the FCT after LinkGuardian's protection is still higher compared to the baseline. On investigating, we found that, when a TCP sender detects reordering, it stops increasing its `cwnd` until it receives an ACK indicating that the reordering has been resolved. For short flows, since the TCP sender is typically in the slow-start state, such a momentary pause in the increase of `cwnd` leads to a momentary pause in data transmission, thereby increasing the FCT. Also, at the 99th percentile, LinkGuardian cannot mitigate the increase in FCT since it does not currently handle tail packet loss. The overall FCT distribution does not vary significantly across the loss rates since we consider only the "affected" flows which typically face a single packet loss for all the loss rates. The FCT results for CUBIC are similar since both CUBIC and DCTCP follow similar TCP slow-start [4]. For other flow sizes, we observe qualitatively similar results.

4.2 Overheads

LinkGuardian maintains state for each port on the switch and requires ~2% of the total SRAM memory when provisioned for 256 ports per switch. In the direction of corruption, LinkGuardian incurs a throughput overhead of ~0.2% due to the custom header (see Table 2 baseline). In the reverse direction, the 64 byte ACK packets sent every 5 μs add a maximum throughput overhead of ~100 Mbps (1%). Across all the throughput experiments, the buffered packets at the sender switch consumed a maximum of 5.44 KB (3-4 packets) packet buffer. Overall, our current implementation of LinkGuardian is able to keep the overheads low.

5 DISCUSSION AND FUTURE WORK

LinkGuardian is amenable to incremental deployment. Operators can prioritize the deployment at specific network locations where a corrupting link could impact many paths in the network. LinkGuardian does not eliminate the need to physically repair a corrupting link but instead buys time for the repair while *locally* mitigating the impact of corruption and reducing the likelihood of violating SLAs. A system like CorrOpt [30] would still be needed to ensure that the network capacity requirements are met during the physical repair and LinkGuardian can be used in combination with it.

Current limitations and path forward. LinkGuardian is a work in progress and currently focuses on the common cases, i.e. we seek not to completely eliminate packet corruption loss but to reduce it to a tolerable level. In particular, we assume that the packets from the receiver switch are not corrupted since corruption loss is unidirectional ~90% of the time [30]. In the future, for a link with bi-directional corruption, we could protect the normal traffic in the reverse direction by implementing an independent *instance* of LinkGuardian working in the reverse direction. The loss notification packets in the reverse direction can be ensured to reach the sender switch by creating multiple copies of them.

LinkGuardian also does not currently handle the loss of *consecutive* packets or *tail* packets (in case of short flows). Our preliminary measurements of corruption loss characteristics using a Variable Optical Attenuator (VOA) suggest such occurrences to be uncommon. However, as observed in Table 3, LinkGuardian would need to handle tail losses to prevent violation of higher levels of service guarantees. The main challenge in handling tail losses is in detecting them. We plan to detect tail losses by injecting dummy packets on the link each time the link is idle.

Preserving packet ordering. In §4.1, we see that LinkGuardian can only partially mitigate the impact of packet corruption for latency-sensitive flows. To completely eliminate the impact of packet corruption, LinkGuardian would need to preserve packet ordering so that the corruption loss recovery is completely transparent to the TCP endpoints. This would require the receiver switch to buffer the out-of-order packets and to retransmit them in order. Since link-local recovery delays are small (see §2), considering other overheads, we estimate the receiver switch to require ~12.5 KB (~8 packets) of packet buffer to temporarily hold the out-of-order packets for a 10G link. As this is a very small fraction of the packet buffer available on switches [25], we foresee that in-order packet recovery is perfectly plausible. Once we achieve this, LinkGuardian can then also be employed to make RDMA traffic, which is more sensitive to loss and reordering, resilient to packet corruption losses.

Scalability. LinkGuardian works locally at an individual link and is therefore agnostic to the overall size of a network. In other words, LinkGuardian will work for a large petabit-scale network if we can get it to work fast enough for an individual corrupting link. Although datacenter links are becoming faster, the sending rate of an individual TCP flow determines whether LinkGuardian's out-of-order retransmission is able to mitigate the impact of packet corruption loss. For example, our current results suggest that LinkGuardian's out-of-order retransmission mechanism is able to mitigate the impact of packet corruption loss even on a 400G link, as long as the sending rate of each of the individual TCP flows is not greater than 10 Gbps. We plan to determine the maximum TCP sending rate that our current out-of-order retransmission mechanism can support and explore the option of in-order packet recovery (discussed above) and its trade-offs for throughput-sensitive TCP flows.

Efficient packet buffering in the dataplane. LinkGuardian currently implements packet buffering via recirculation which incurs dataplane pipeline processing overhead. This overhead is incurred only for packets that are sent on the corrupting links, and a very small number of links connected to the same switch pipeline could be corrupting packets at the same time [30]. Since switch pipelines typically have extra processing headroom [20], we expect this overhead to be acceptable. In the future, this overhead could be avoided if we could programmatically hold and release packets from the switch's packet buffer without having to recirculate them. This is possible with the next generation programmable switches such as the Intel Tofino2 [22] and we plan to validate this approach once we are able to secure the hardware.

Monitoring links for corruption. In order to activate LinkGuardian, we currently rely on existing techniques [15, 30] to monitor and detect when a link starts corrupting packets. In particular, Wharf's link monitoring agent [15] infers corruption by polling network port counters locally in the switch control plane and running a protocol between two neighboring switches. While we plan to investigate link monitoring techniques in the future, we expect the detection latency for such link-local techniques to be in the order of a few milliseconds. This is a one-time cost and a small fraction of the total time between when the packet corruption starts and when the link's physical repair is performed. LinkGuardian mitigates the impact of corruption packet loss for the entire duration after packet corruption is detected.

Serialization and propagation delays. In our analysis in §2, we ignored the propagation delay for simplicity. Strictly speaking, for out-of-order retransmission to work, it is necessary for the serialization delay of the subsequent two TCP data packets to be greater than the sum of the propagation and serialization delays of the loss notification packet. While this holds true in our testbed, the propagation delay is expected to increase with longer fibers and the serialization delay will decrease when the link speed goes up. Our calculations show that the current out-of-order retransmission mechanism should work on a 10G fiber link of up to ~238 m. For a 100G fiber link, this limit will drop to ~24 m (assuming a 100G TCP sender). While these requirements are likely to change over time as the underlying hardware improves, caution needs to be exercised when deploying LinkGuardian on longer or high-speed fiber links.

Loss-oblivious congestion control. Our evaluations show that TCP benefits from LinkGuardian because it is not resilient to random losses. It was therefore a question of whether loss-oblivious congestion control algorithms like BBR [9] would also benefit from LinkGuardian. Our preliminary experiments suggest that throughput-sensitive BBR flows are not impacted much by packet corruption losses, but short BBR flows still benefit from LinkGuardian's protection.

6 RELATED WORK

In this section, we place LinkGuardian in the context of the most relevant prior work as categorized below.

Avoiding the faulty links. A common strategy is to disable a corrupting link when the loss rate is above a certain threshold [27, 30]. CorrOpt [30] uses a global view of the network to selectively disable corrupting links while meeting capacity constraints. Besides the loss of capacity, disabling a link causes disruption to the rest of the network [27] as routing and load-balancing protocols need to move traffic away from the disabled link. RAIL [31] avoids lossy links for latency-sensitive applications through virtual network topologies but requires end-host modifications.

End-host based recovery. End-to-end redundancy via FEC [14, 31] or packet duplication [26] can help, but it adds redundant bytes for *all* the packets across the *entire* path and risks worsening congestion in the network. Latency-sensitive flows can make faster loss recovery using NIC-offloaded [6] and/or multipath [10] transport stacks, but the recovery delay is still lower-bounded by 1 RTT.

Link-local recovery. Wharf [15] mitigates corruption by using link-local frame-level block-based FEC. However, compared to LinkGuardian, it incurs significant overhead (see Table 2) as redundancy is added for *all* the packets while corruption loss rates remain small. In addition, it requires FPGA support on switches for FEC encoding/decoding. Newer Ethernet standards provide FEC options at the PHY layer [19]. However, our preliminary measurements suggest that the links can still suffer from packet corruption loss. SQR [25] is another algorithm that implements link-local retransmission, but it is designed to recover packet loss during fail-stop link failures and does not work for corrupting links.

Unlike prior work, LinkGuardian's overheads are low and proportional to the corruption loss rate, while its impact is localized to the corrupting link. It requires no support from the end-hosts and the recovery delay is less than 1 RTT.

7 CONCLUSION

In this paper, we argue that link-local retransmission is a promising approach to mitigate packet corruption losses in datacenter networks. We show that a simple implementation that retransmits out-of-order is able to mitigate the impact of packet corruption loss on both throughput-sensitive flows and latency-sensitive flows. LinkGuardian is currently a work in progress, but it represents a step toward self-driving/self-patching networks [12, 13] where network faults patch by themselves allowing the network to run seamlessly with minimal impact on the SLAs.

ACKNOWLEDGMENTS

This work was supported by the Singapore Ministry of Education Academic Research Fund Tier 1 (T1 251RES1917) and Tier 2 (MOE2019-T2-2-134).

REFERENCES

[1] 3GPP. 2007. TS 36.321: E-UTRA; Medium Access Protocol Specification (Release 8). (2007).
[2] 3GPP. 2020. TS 36.321: LTE; E-UTRA; Medium Access Protocol Specification (Release 16). (2020).
[3] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Francis Matus, Rong Pan, Navindra Yadav, George Varghese, et al. 2014. CONGA: Distributed Congestion-aware Load Balancing for Datacenters. In *Proceedings of SIGCOMM*.
[4] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data Center TCP (DCTCP). In *Proceedings of SIGCOMM*.
[5] Mark Allman, Vern Paxson, and Ethan Blanton. 2009. TCP Congestion Control. *RFC 5681* (2009).
[6] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. 2020. Enabling Programmable Transport Protocols in High-SpeedNICs. In *Proceedings of NSDI*.

[7] Hari Balakrishnan, Srinivasan Seshan, Elan Amir, and Randy H. Katz. 1995. Improving TCP/IP Performance over Wireless Networks. In *Proceedings of MOBICOM*.

[8] Ethan Blanton, Mark Allman, Lili Wang, Ilpo Jarvinen, Markku Kojo, and Yoshifumi Nishida. 2012. A conservative loss recovery algorithm based on selective acknowledgment (SACK) for TCP. *RFC 6675* (2012).

[9] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2016. BBR: Congestion-based congestion control: Measuring bottleneck bandwidth and round-trip propagation time. *Queue* 14, 5 (2016), 20–53.

[10] Guo Chen, Yuanwei Lu, Yuan Meng, Bojie Li, Kun Tan, Dan Pei, Peng Cheng, Layong Larry Luo, Yongqiang Xiong, Xiaoliang Wang, et al. 2016. Fast and cautious: Leveraging multi-path diversity for transport loss recovery in data centers. In *Proceedings of NSDI*.

[11] Linux Networking Documentation. 2022. *DCTCP (DataCenter TCP)*. https://www.kernel.org/doc/html/latest/networking/dctcp.html

[12] Nick Feamster, Arpit Gupta, Jennifer Rexford, and Walter Willinger. 2019. NSF workshop on measurements for self-driving networks. In *Proceedings of Workshop on Measurements for Self-Driving Networks*.

[13] Nick Feamster and Jennifer Rexford. 2017. Why (and how) networks should run themselves. *arXiv preprint arXiv:1710.11583* (2017).

[14] Zeng Gaoxiong, Chen Li, Yi Bairen, and Chen Kai. 2021. Optimizing Tail Latency in Commodity Datacenters using Forward Error Correction. *arXiv preprint arXiv:2110.15157* (2021).

[15] Hans Giesen, Lei Shi, John Sonchack, Anirudh Chelluri, Nishanth Prabhu, Nik Sultana, Latha Kant, Anthony J McAuley, Alexander Poylisher, André DeHon, et al. 2018. In-network computing to the rescue of faulty links. In *Proceedings of the NetCompute Workshop*.

[16] P4.org Architecture Working Group. 2018. *Portable Switch Architecture*. https://p4.org/p4-spec/docs/PSA-v1.1.0.pdf

[17] IEEE. 2009. *802.11n-2009 Standard*. https://standards.ieee.org/ieee/802.11n/3952/

[18] IEEE. 2013. *802.11ac-2013 Standard*. https://ieeexplore.ieee.org/document/6687187

[19] IEEE. 2018. *802.3-2018 Standard*. https://ieeexplore.ieee.org/document/8457469

[20] Raj Joshi, Ben Leong, and Mun Choon Chan. 2019. Timertasks: Towards time-driven execution in programmable dataplanes. In *Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos*.

[21] Glenn Judd. 2015. Attaining the Promise and Avoiding the Pitfalls of TCP in the Datacenter. In *Proceedings of NSDI*.

[22] Jeongkeun Lee. 2020. Advanced Congestion & Flow Control with Programmable Switches. In *P4 Expert Roundtable Series*. https://bit.ly/3J8x7fw

[23] Hwijoon Lim, Wei Bai, Yibo Zhu, Youngmok Jung, and Dongsu Han. 2021. Towards timeout-less transport in commodity datacenter networks. In *Proceedings EuroSys*.

[24] Christina Parsa and JJ Garcia-Luna-Aceves. 1999. TULIP: A Link-Level Protocol for Improving TCP over Wireless Links. In *Proceedings of WCNC*.

[25] Ting Qu, Raj Joshi, Mun Choon Chan, Ben Leong, Deke Guo, and Zhong Liu. 2019. SQR: In-network packet loss recovery from link failures for highly reliable datacenter networks. In *Proceedings of ICNP*.

[26] Ashish Vulimiri, Oliver Michel, P Brighten Godfrey, and Scott Shenker. 2012. More is less: Reducing latency via redundancy. In *Proceedings of HotNets*.

[27] Xin Wu, Daniel Turner, Chao-Chih Chen, David A Maltz, Xiaowei Yang, Lihua Yuan, and Ming Zhang. 2012. NetPilot: Automating datacenter network failure mitigation. In *Proceedings of SIGCOMM*.

[28] Yu Zhou, Chen Sun, Hongqiang Harry Liu, Rui Miao, Shi Bai, Bo Li, Zhilong Zheng, Lingjun Zhu, Zhen Shen, Yongqing Xi, et al. 2020. Flow event telemetry on programmable data plane. In *Proceedings of SIGCOMM*.

[29] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y Zhao, et al. 2015. Packet-level telemetry in large datacenter networks. In *Proceedings of SIGCOMM*.

[30] Danyang Zhuo, Monia Ghobadi, Ratul Mahajan, Klaus-Tycho Förster, Arvind Krishnamurthy, and Thomas Anderson. 2017. Understanding and mitigating packet corruption in data center networks. In *Proceedings of SIGCOMM*.

[31] Danyang Zhuo, Monia Ghobadi, Ratul Mahajan, Amar Phanishayee, Xuan Kelvin Zou, Hang Guan, Arvind Krishnamurthy, and Thomas Anderson. 2017. RAIL: A Case for Redundant Arrays of Inexpensive Links in Data Center Networks. In *Proceedings of NSDI*.