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
National University of Singapore
rajjoshi@comp.nus.edu.sg, guoqi0912@gmail.com, {nishant, ayush, chanmc, benleong}@comp.nus.edu.sg

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 naive 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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

APNet 2022, July 1–2, 2022, Fuzhou, China © 2022 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.

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 for wired networks have 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 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 naively retransmits 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.

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.

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 10Gbps DPDK-capable NIC. Both the hosts run Linux kernel 5.4.0-91-lowlatency on Ubuntu 20.04.3. The network’s MTU is 1,518 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 transmitted 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 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 \( \sim 1.23 \mu s \), the TCP’s “reordering window” of 3 packets is \( \sim 3.69 \mu s \). From Table 1, we see that the average estimated link-local recovery delay is \( \sim 2.59 \mu s \) which is less than \( \sim 3.69 \mu s \). 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 \( \sim 10^{-6} \) [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. 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.

<table>
<thead>
<tr>
<th>End-host based (kernel)</th>
<th>End-host based (DPDK)</th>
<th>Link-local Retransmission</th>
</tr>
</thead>
<tbody>
<tr>
<td>mean</td>
<td>32.73</td>
<td>33.44</td>
</tr>
<tr>
<td>50%</td>
<td>32.50</td>
<td>33.52</td>
</tr>
<tr>
<td>99%</td>
<td>44.00</td>
<td>46.25</td>
</tr>
</tbody>
</table>

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 carry 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 & ReTx

As described in §3.2, the receiver switch keeps the copy of latestRxSeqNo on the sender switch updated and 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 $>\times10^{-3}$ such that the effective loss rate is $\leq10^{-6}$.

### 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 of a fixed width, it will periodically wrap around and restarts at zero. Therefore, to correctly compare sequence numbers, we need an additional bit that 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 sw3 and sw4 are two independent physical switches functioning as LinkGuardian sender and receiver switches, respectively. In all experiments, h1 uses iperf to send CUBIC or DCTCP 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 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 inures a proportional degradation of ~2% compared to the baseline.

<table>
<thead>
<tr>
<th>Loss Rate → 0 (Baseline)</th>
<th>10^{-3}</th>
<th>10^{-4}</th>
<th>10^{-5}</th>
<th>10^{-6}</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUBIC</td>
<td>9.49</td>
<td>9.48</td>
<td>7.28</td>
<td>3.43</td>
</tr>
<tr>
<td>DCTCP</td>
<td>9.49</td>
<td>9.46</td>
<td>7.88</td>
<td>3.32</td>
</tr>
</tbody>
</table>
Table 3: FCT distribution (in $\mu$s) for 45 KB “affected” DCTCP flows with and without LinkGuardian (LG)

<table>
<thead>
<tr>
<th>Loss rate</th>
<th>0</th>
<th>$10^{-4}$</th>
<th>$10^{-3}$</th>
<th>$10^{-2}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(Base)</td>
<td>Lossy</td>
<td>LG</td>
<td>Lossy</td>
</tr>
<tr>
<td>min</td>
<td>113</td>
<td>198</td>
<td>174</td>
<td>193</td>
</tr>
<tr>
<td>mean</td>
<td>197</td>
<td>701</td>
<td>389</td>
<td>707</td>
</tr>
<tr>
<td>50%</td>
<td>180</td>
<td>423</td>
<td>268</td>
<td>419</td>
</tr>
<tr>
<td>90%</td>
<td>311</td>
<td>2016</td>
<td>442</td>
<td>2421</td>
</tr>
<tr>
<td>95%</td>
<td>315</td>
<td>3182</td>
<td>500</td>
<td>3216</td>
</tr>
<tr>
<td>99%</td>
<td>329</td>
<td>4114</td>
<td>3412</td>
<td>4192</td>
</tr>
</tbody>
</table>

Next, for latency-sensitive ($\leq 100$ 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 $\mu$s) 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 $\sim 35\%$ and $\sim 85\%$ respectively, for all the loss rates. However, for the same percentiles, FCT after LinkGuardian’s protection is still higher compared to the baseline. On investigating, we found that when a TCP sender is in the slow start state, it does not exhibit “reordering tolerance” and exits slow start immediately on detecting loss. Consequently, the $cwnd$ grows more slowly leading to an increase in 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 $\sim 2\%$ of the total SRAM memory when provisioned for 256 ports per switch. In the direction of corruption, LinkGuardian incurs a throughput overhead of $\sim 0.2\%$ due to the custom header (see Table 2 baseline). In the reverse direction, the 64 byte ACK packets sent every 5 $\mu$s add a maximum throughput overhead of $\sim 100$ 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 $\sim 90\%$ of the time [30]. In 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 protected by duplicating 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. Since TCP does not exhibit ”reordering tolerance” during the slow start, LinkGuardian can only partially mitigate the impact of corruption for latency-sensitive flows (see §4.1). 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. While challenging to do so, we believe that the next-generation programmable switches provide features such as packet buffering without recirculation [22] that make this plausible. Since link-local recovery delays are small (see §2), considering other overheads, we estimate the receiver switch to require $\sim 12.5$ KB ($\sim 8$ 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 [23], 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 recover from a corruption loss. For example, our current results suggest that our current out-of-order retransmission mechanism is able to detect recover corruption losses for link rates up to 400G, as long as the sending rates for the individual TCP flows are lower than 10 Gbps. We plan to determine the maximum TCP sending rate that our current out-of-order transmission 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 future, this overhead could be avoided if we could programatically 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 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 decreases when 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 (for a single 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 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 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 towards self-driving/self-patching networks [12, 13] where network faults patch themselves allowing the network to run seamlessly with minimal impact on the SLAs.

REFERENCES


