Understanding the Micro-Behaviors of Hardware Offloaded Network Stacks with Lumina

Paper #144, 12 pages body + 3 pages references

Abstract

Hardware offloaded network stacks are widely adopted in modern datacenters to meet the demand for high throughput, ultra-low latency and low CPU overhead. To best utilize the superb performance, network developers need to have in-depth understandings of their behaviors. Recent years, there have been various of testing tools helping users to test and understand software network stacks. However, hardware network stacks are left behind as its kernel bypass nature and high performance make testing challenging. In this paper, we present Lumina, a tool to test the correctness and performance of hardware network stacks. The key idea of Lumina is exploiting network programmability to emulate various network scenarios and dumping the complete packet trace for offline analysis. Lumina supports injecting deterministic events with user-friendly interfaces, thus enabling developers to write precise reproducible tests. Due to the limited resource and flexibility of programmable network devices, we mirror and dump traffic to a pool of dedicated servers for offline analysis. We start with RDMA NIC as the testing target and prototype Lumina with our testbed. We evaluate Lumina with microbenchmark experiments and share our findings on three widely used RDMA NICs using Lumina.

1 Introduction

Modern cloud applications demand high throughput and ultra-low processing latency (several μs) with low CPU overhead. Traditional TCP/IP stacks in operating system (OS) kernel are ill-suited to these requirements. Therefore, cloud providers tend to offload their network stacks into network interface card (NIC) hardware to achieve better performance and free-up CPU cycles. Hardware offloaded networking technics (e.g., RDMA, SRD [1]) have been widely deployed in various of areas including data storage [2, 3], deep learning [4, 5], etc.

To best utilize the great performance brought by hardware offloading, network developers should be familiar about the behavior of the hardware network stacks. While reading specifications provided by vendors is helpful, the specifications are normally hundreds of pages long and some are tedious to read. Most importantly, the specifications normally does not touch the details of the actual implementation. In the past decades, there have been many tools [6–9] to test the software network stack implementation. These tools enable users to test the correctness of the network stack implementation and understand the bias between specification and actual implementation. For example, packetdrill [6] uses libpcap or TUN device as a "shim layer" to inject and consume packets. Users can customize test cases based on the scripting language provided by packetdrill to test their TCP/UDP/IP network stacks. However, testing tools for hardware network stacks are in a very initial state. For example, to test RDMA, network operators normally use synthetic workloads (e.g., `perftest` [10]) to measure overall performance. This approach suffers from noises of different applications and network conditions and cannot capture micro-behaviors such as congestion control and retransmission.

Enabling fine-grained testing for hardware network stacks has two challenges. First, the network protocols are implemented in the hardware which is almost a blackbox and we cannot touch or modify the code inside. Because the data path bypasses the kernel, we cannot inject packets/events or measure behaviors at the end host. Thus the testing tools [6] that use a shim layer (e.g., libpcap, TUN) to inject packets cannot work for hardware network stacks. Second, as hardware network stacks provide high throughput and ultra-low latency, testing them is very time-sensitive: we cannot add extra substantial latency to the under-test traffic.

In this paper, we design and implement Lumina, a tool to test the correctness and performance of hardware network stack implementation. To overcome the challenge that we cannot interact with the hardware implementaion, Lumina adopts an in-network solution: we connect two under-test hardwares with a programmble middlebox and use the programmable middlebox to inject network events and test the behaviors. As mentioned above, the programmable middlebox should not introduce substantial latency. With the rapid development of programmable network devices, we now have various of choices for such high-performance programmable middleboxes: programmable switches, smartNICs, or software switches. Each of them can support programmability to some extent and high-speed packet processing.

However, due to the limited resource (e.g., on-chip memory,

processing cycles) on commodity in-network devices, it is hard to realize complex measurement and testing tasks fully in the network. To this end, we utilize the mirror function to generate copies of network traffic and dump them to dedicated servers. The dumped packet traces are used for further analysis.

The design goal of Lumina is to reliably enable developers to write *reproducible* tests and inject *deterministic* events with *user-friendly* interfaces. To achieve this goal, we co-design three components: event injector (on middlebox), traffic generator (on traffic servers with under-test hardware), and packet dumper (on mirror servers). Lumina creates a dedicated connection between traffic generator and event injector to share the traffic metadata so that event injector can take user-friendly configurations as input and generates deterministic entries for the event injector. When doing mirroring, Lumina tags extra information (sequence number, timestamp, event type) to the mirror traffic to enable reliable and reproducible tests. Besides, Lumina supports per-packet load balanced packet dumping to dump packets to the packet dumper, which guarantees the reliability and efficiency of Lumina itself.

In summary, we make the following contributions.

- We propose Lumina to enable testing the correctness and performance of hardware network stack implementation.
- We co-design event injector, traffic generator and packet dumper to provide reliable and user-friendly interfaces for network developers to write reproducible tests and inject deterministic events.
- We prototype Lumina to test RDMA NICs. Our microbenchmark experiments demonstrate the efficiency and negligible-overhead of Lumina. Besides, we present our findings about testing RDMA NICs using Lumina, related to fast retransmission, timeout, and congestion notification.

2 Background and Motivation

In this section, we first introduce the background of hardware offloaded network stack. Then we present the motivation and challenges of hardware network stack testing.

2.1 Hardware offloaded network stacks

We list some representative hardware offloaded network stack techniques as follows.

TCP offload engine (TOE): Unlike techniques [11–13] that only offload some operations, TOE offloads processing of the *entire* TCP/IP stack to the NIC. This technology is supported by some vendors (e.g., Chelsio Terminator 5 [14] and Broadcom NetXtreme II 5708 [15]). TOE not only frees up CPU cycles, but also reduces PCIe traffic.

RDMA: RDMA enables NIC to transfer data from the pre-registered memory to the wire or from the wire to the memory. The networking protocol is implemented on the NIC, thus bypassing OS kernel. Unlike TCP, RDMA provides message semantics rather than stream semantics. The high performance computing (HPC) community has long deployed RDMA in special purpose clusters [16] using InfiniBand (IB) [17]. In recent years, cloud providers also deploy RDMA in datacenters over Ethernet and IP networks [2, 18] to free CPU cores and accelerate communication-intensive workloads, e.g., storage and machine learning (ML). To this end, most of cloud providers adopt RDMA over Converged Ethernet (RoCE) v2 given its large vendor ecosystem [19–23]. Another alternative is Internet Wide Area RDMA Protocol (iWARP) [24, 25] which runs RDMA over TCP/IP on the NIC.

Scalable reliable datagram (SRD): SRD is a customized transport protocol implemented in Amazon Web Services Nitro networking card [1]. SRD spays packets over multiple paths to minimize the chance of hotspots and use delay information for congestion control. Unlike TCP and RDMA reliable connection (RC), SRD provides reliable but out-of-order delivery and leaves order restoration to applications.

2.2 Motivation

With wide adoptions of hardware offloaded network stacks, it is important for network operators to have in-depth understandings of their behaviors. Recent years have witnessed the progress [6, 7, 26–28] in testing software network stacks. However, testing hardware offloaded network stacks is left behind. For example, to safely enable RDMA, network operators run synthetic workloads (using traffic generators like `perftest` [10]) and application load tests in lab testbed and test clusters before production deployments [2, 18]. While this approach can measure overall performance and reveal significant functional bugs, it cannot accurately capture micro-behaviors and suffers from noises due to variations in application and network conditions, let alone its high resource consumption. Hence, performance anomalies and bugs in these areas are likely to remain unnoticed in tests, and then hit production networks. These problems will be magnified as the link speed keeps increasing and hardware network stacks are becoming more and more complex.

To illustrate this problem, we use NVIDIA Mellanox ConnectX-4 RDMA NIC as an example. Shpiner et al. evaluated its performance over a lossy testbed network, and found it could preserve high goodput under synthetic incast workloads (Figure 4 and 5 in [29]). However, we find that the retransmission delay of this NIC is actually around 200μs (Figure 8 and 9), which is about 100 base round-trip times (RTTs). Given these limitations, we need a tool that can enable developers to easily write precise and reproducible tests for hardware offloaded network stacks. To this end, the tool should be able to interact with hardware network stacks (e.g.,

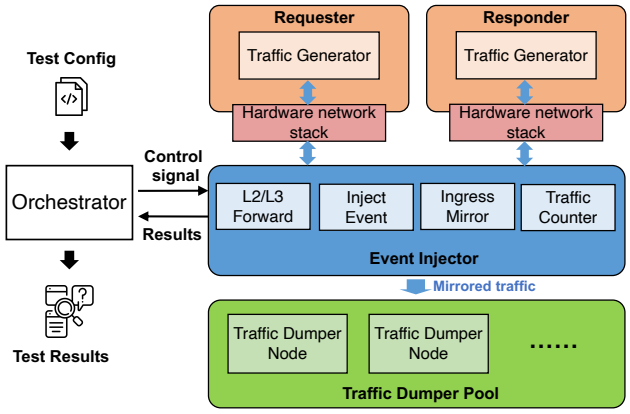


Figure 1: Lumina Overview.

```
requester:
  workspace: /home/foo/bar/
  username: test
  control-ip: cx4-testing-traffic-requester
  nic:
    type: cx4
    if-name: enp4s0
    switch-port: 144
    ip-list:
      - 10.0.0.2/24
      - 10.0.0.12/24
  roce-parameters:
    dcqcn-rp-enable: False
    dcqcn-np-enable: True
    min-time-between-cnps: 0
    adaptive-retrans: False
    slow-restart: True
```

Listing 1: Traffic Generation Host Configuration Snippet

inject packet losses) in a flexible and deterministic manner, and accurately capture their micro-behaviors (e.g., per-packet transmission time).

Compared to software network stacks, hardware network stacks impose some unique challenges for testing tools. First, since hardware network stacks bypass kernel, we cannot interact with network stacks and measure their behaviors through shim layers, e.g., libpcap and TUN device [6] at the host as before. Second, hardware runs at high throughput and ultra-low latency. This translates to stringent performance requirements for test event injections and data plane measurement.

3 The Lumina Design

3.1 Overview

Motivated by above observations, we build Lumina, a tool that enables testing the correctness and performance of RDMA NIC implementations. In this paper, we focus on RoCEv2[1] since it is the de facto hardware offloaded network stack technology in the cloud environment with wide support [19–21] and adoptions [2, 18, 30]. RoCEv2 encapsulates an IB transport packet using UDP and IP headers to enable routing over Ethernet/IP-based networks. We believe Lumina can be extended to support other hardware network stack technologies.

In this section, we first present the design rationale of Lumina. Then we give an overview of Lumina. After that, we introduce the design of each mechanism in detail.

3.2 Design Rationale

The kernel bypass nature of hardware offloaded network stacks prevents us from directly injecting events and measuring behaviors at the end host. To meet this challenge, we embrace an in-network solution. We connect two hosts with hardware network stack under test to a *high-performance programmable middlebox*, which is used to emulate realistic and worst-case network scenarios. The middlebox should be able to forward traffic and be programmed to inject various events at the line rate with ultra-low extra processing delay. Recent industry progress on reconfigurable network hardware provides many options [31–34] for the middlebox. In current prototype, we adopt the programmable switch.

However, the middlebox hardware may not have enough resource and flexibility (e.g., limited stateful memory and arithmetic operations) to realize complex measurement. Instead of in-device measurement, we dump all the packets by mirroring them from the middlebox to dedicated servers. After that, we reconstruct the complete packet trace from dumped packets for further processing and analysis.

As shown in Figure 1, Lumina has four components: *Orchestrator*, *Traffic Generator*, *Event Injector*, and *Traffic Dumper*. To run a test, the orchestrator takes a user configuration file as input, sets up the environment, and sends Remote Procedure Calls (RPCs) to each component to coordinate their executions.

Lumina uses two hosts with the same bandwidth capacity to generate traffic. Each host is equipped with the under-test hardware offloaded network stack and runs a traffic generator instance. We use one host as the requester and the other one as the responder. They generate traffic based on the configurations conveyed from the orchestrator (§3.3).

The event injector forwards the traffic and injects configured events, e.g., ECN marks, packet losses and corruptions (§3.4). In the meantime, the event injector also mirrors all the RoCE packets to the traffic dumper pool, which consists of multiple servers, for offline analysis in the future (§3.5).

Once the traffic finishes, the orchestrator collects results from the other components, e.g., dumped packets, NIC counters and log files. It reconstructs the complete packet trace from dumped packets collected by traffic dumper servers. After that, users can parse the packet trace and other results to analyze behaviors of the hardware network stack (§3.6).

[1] In the following sections, we use RDMA, RoCE, and RoCEv2 interchangeably.

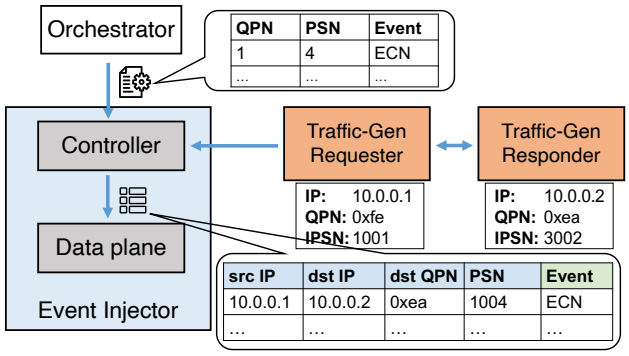


Figure 2: Lumina combines the runtime traffic metadata and intent-based traffic configuration to populate the match-action table for event injection.

3.3 Traffic Generation

Before starting traffic generators, the orchestrator first configures IP addresses and network stack settings, e.g., congestion control and loss recovery parameter, of traffic generation hosts. Listing 1 gives an example.

After the configuration, the orchestrator starts traffic generator instances on both hosts. Our traffic generator adopts Reliable Connected (RC) transport, and supports RDMA send, receive, write, and read verbs. In this paper, we use SEND, RECV, WRITE and READ to denote them, respectively. RDMA traffic generators communicate using one or multiple queue pairs (QPs). As shown in Listing 2, the user can configure many parameters of traffic generators, e.g., the number of QPs, retransmission timeout, and MTU.

After two traffic generators initialize objects such as QPs and memory regions (MRs), they exchange necessary metadata, e.g., QP number (QPN), packet sequence number (PSN), global identifier (GID), memory address and key, through a TCP connection. Since QPNs and PSNs are randomly generated at runtime and critical for the event injector to locate right packets, the traffic generator sends metadata information to the event injector as well (more details in §3.4).

After exchanging metadata and establishing QP connections, the requester posts work requests to generate RDMA traffic. The requester controls the total number of requests/messages and the maximum number of outstanding requests on each QP. In the case of SEND/RECV, the responder keeps posting RECV requests correspondingly. The requester can support barrier synchronization among QPs: the requester posts the next round of requests only after it gets the completions of the current round of requests across all the QPs.

Finally, when the requester gets completions of all the requests, it calculates metrics such as request/message completion times and total goodput, and sends a completion notification to the responder through the TCP connection.

```
traffic:
  num-connections: 2
  rdma-verb: write
  num-msgs-per-qp: 10
  mtu: 1024
  message-size: 10240
  multi-gid: true
  barrier-sync: true
  tx-depth: 1
  min-retransmit-timeout: 14
  max-retransmit-retry: 7
  data-pkt-events:
  # Mark the 4th pkt on the 1st QP conn
  - qpn: 1
    psn: 4
    type: ecn
    iter: 1
  # Drop the 5th pkt on the 2nd QP conn
  - qpn: 2
    psn: 5
    type: drop
    iter: 1
  # Drop the retrans 5th pkt on the 2nd QP conn
  - qpn: 2
    psn: 5
    type: drop
    iter: 2
```

Listing 2: Traffic and Event Injection Configuration Snippet

3.4 Event Injection

Lumina connects two traffic generation hosts to a high-performance programmable middlebox (event injector). To emulate realistic network scenarios like congestion and failures. the event injector can be programmed to inject packet corruptions, packet drops and ECN marks to RDMA data packets[2]. It is worthwhile to notice that the responder generates data packets for READ while the requester generates data packets for the other verbs.

While above events are not difficult to realize on reconfigurable network hardware, we find that the biggest challenge is to provide *user-friendly* interfaces for developers to express a series of *deterministic* injection events. This challenge is actually translated into two concrete requirements as follows.

Deterministic: Since Lumina aims at precise and reproducible tests to understand micro-behaviors, it only accepts descriptions of deterministic injection events from the user. A description like "randomly drop 10% packets" is not deterministic as different rounds can drop different sequences of packets. In contrast, a description like "drop the first packet of the first QP" can generate deterministic injection behaviors.

User-friendly: Users should be able to express their high-level testing intents without the need to understand low-level details of Lumina. For example, the user should be able to tell the Lumina to drop the first packet of the second QP, then drop the retransmission of this packet. The user does not need to specify QPN and PSN for each QP, and understand how the event injector identifies the retransmitted packet.

[2]Currently Lumina does not support injecting events to control packets, e.g., ACK and NACK.

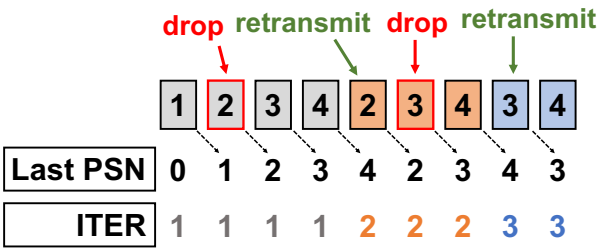


Figure 3: Lumina maintains ITER to differentiate packets in fine-grained. ITER denotes the rounds of (re)transmissions for a connection. If PSN of the current packet is no larger than that of the previous packet, ITER is increased by 1.

Listing 2 gives a configuration example of event injections. There are three events across two QP connections. It is worthwhile to notice that Lumina only preserves the order of events on the same QP connection. The events of different QP connections are independent. On the first QP, we mark its fourth packet. On the second QP, we drop its fifth packet. When the sender retransmits this lost packet, we drop it again. Users just need to specify *relative* QPNs and PSNs, and can use *iter* field to express retransmission behaviors, thus to locate retransmitted packets which have same QPN and PSN as original packets.

Next, we will describe how Lumina translates high-level test intents (e.g., relative QPN and PSN) to the low-level configuration of the event injector, and uses an iteration number (ITER) to express per-connection retransmission behaviors.

Translate user intents to configurations: Since users only provide high-level intent information such as relative PSN and QPN, Lumina needs to translate this to the low-level configuration for the event injector. One straightforward solution is using the event injector to detect new QPs and parse their QPNs and initial PSNs on the data plane. While promising, this approach significantly complicates the data plane. This is because, for every RDMA packet, the event injector first checks if it belongs to a new QP. If yes, the event injector further needs to initialize states for this new QP.

Instead of the above stateful approach, we take a stateless approach by leveraging traffic generators to provide runtime traffic metadata. As mentioned above (§3.3), traffic metadata like QPN and initial PSN (IPSN) is randomly generated at runtime. Once traffic generator instances finish exchanging metadata through TCP, the traffic requester sends the complete traffic metadata to the event injector through the control plane. The metadata is organized as a list of tuples. Each tuple contains the information for a certain QP connection: requester IP/QPN/IPSN and responder IP/QPN/IPSN. After that, the event injector combines the runtime traffic metadata from traffic generators and traffic configuration intents from the orchestrator to populate the match-action table for event injections. Only after the event injector populates the table, traffic generators can start RDMA traffic.

Figure 2 gives an example. The IP address, QPN, and IPSN of the requester's QP are 10.0.0.1, 0xfe, and 1001, respectively. The IP address, QPN, and IPSN of the responder's QP are 10.0.0.2, 0xea, and 3002, respectively. Data packets are sent from the requester to the responder. The user intends to drop the fourth packet of the first QP connection. By combining above information, the event injector computes and inserts the following entry: if the source IP, destination IP, destination QPN, and PSN fields of a RoCEv2 packet are 10.0.0.1, 10.0.0.2, 0xea, and 1004 (1001 + 4 - 1), respectively, we should mark ECN for this packet.

Express retransmission behaviors: In many tests, the user needs to inject events to retransmitted packets to understand behaviors like retransmission timeout backoff. However, we cannot differentiate the retransmitted packet from the original packet by looking into the RoCEv2 packet header since they have the same IP addresses, UDP ports, QPN and PSN.

To realize this flexibility, we introduce an iteration number *ITER*, which denotes the rounds of (re)transmissions for a connection. ITER starts from 1 and is maintained by the event injector. For every arriving RDMA packet, the event injector compares its PSN with PSN of the last packet of its connection. If PSN of the current packet is *not larger than* that of the previous packet, the event injector identifies this as a new round of transmissions and increases ITER of this connection by 1. Regardless of the comparison result, the event injector always updates PSN of the last packet of the connection using PSN of the current packet. Lumina can use (PSN, ITER) to uniquely identify every packet in a connection. It is worthwhile to note that checking per-packet PSN and updating per-connection last PSN are done before event injections.

Figure 3 gives an example of how Lumina tracks ITER. In this example, there is only a single connection and the user intends to drop the second packet in the first round (PSN=2, ITER=1), and the third packet in the second round (PSN=3, ITER=2). The sender transmits four packets. ITER is initialized as 1 and the last PSN is set to IPSN-1, which is 0 in this case. In the first iteration, we drop packet 2. When packet 2 is retransmitted, current PSN (2) is smaller than the last PSN (4), thus triggering a new round of transmissions (ITER=2). Likewise, after we drop packet 3 in the second round, the retransmission of packet 3 triggers a new round (ITER=3).

3.5 Traffic Dumping

Lumina aims to dump *all* the RDMA packets between traffic generators for offline analysis. A straightforward solution is using tools like `ibdump` to dump packets at the end host. However, it is unclear if traffic dumping at the end host will impact behaviors of network stacks. In addition, if we want to reconstruct the complete packet trace from packets dumped at

both traffic generation hosts, we need to realize nanosecond-level clock synchronization, which is non-trivial [35].

Realizing these challenges, we adopt the event injector to mirror all the packets to a group of dedicated servers, which forms a traffic dumper pool. Packet mirroring essentially clones packets of specified interfaces and forward them to other interfaces for examination. It has been widely used for measurement and diagnosis purposes [36, 37]. We mirror all the RoCE packets at the ingress pipeline before actually dropping any packets in Memory Management Unit (more details in §5). We choose ingress mirroring instead of egress mirroring because we want to capture original behaviors of hardware network stacks.

For the ease of integrity check and traffic analysis, we leverage the event injector to embed some important metadata in mirrored packets. To avoid losses during packet dumping, we develop a per-packet load balancing mechanism to evenly distribute mirrored packets across CPU cores of traffic dumpers. We will describe them in detail.

Embedding metadata in mirrored packets: For the ease and efficiency of testing, the event injector embeds three types of metadata, *mirror sequence number*, *event type* and *mirror timestamp*, in mirrored packets for the following purposes.

1. **Integrity check.** As we use mirrored packets to understand behaviors of the hardware network stack, the first and most important step is to make sure we mirror and dump all the packets. To this end, the event injector maintains a global variable, *mirror sequence number*, which is incremented for every arriving RDMA packet and embedded in each mirrored packet. Together with switch port counters, we can easily verify if there is any packet loss. If we dump all the packets, we should see consecutive mirror sequence numbers, and the largest mirror sequence number matches the total number of RX packets.
2. **Indicating events.** For the ease of analyzing mirrored packets, we embed an *event type* in each packet to indicate the injected event, currently including ECN marking, drop, and corruption, and none. Note that we mirror all the packets at the ingress pipeline before Memory Management Unit (MMU) executes dropping actions.
3. **Fine-grained measurement.** To accurately measure behaviors of the hardware offloaded network stack, we embed a *mirror timestamp* in each mirrored packet, which carries the nanosecond-level time when the original packet enters the ingress pipeline. Since the event injector adds timestamps to all the packets, it does not require clock synchronization.

To embed above metadata in mirrored packets, a straightforward solution is expanding packets with new fields storing these metadata. However, this may overload the bandwidth capacity of mirroring ports if original traffic's throughput is close to line rate. To avoid this, we rewrite existing header fields that are not involved in traffic analysis to store above metadata. We use the Time to Live (TTL) field, the source MAC address field, and the destination MAC address field, to store *event type*, *mirror sequence number*, and *mirror timestamp*, respectively.

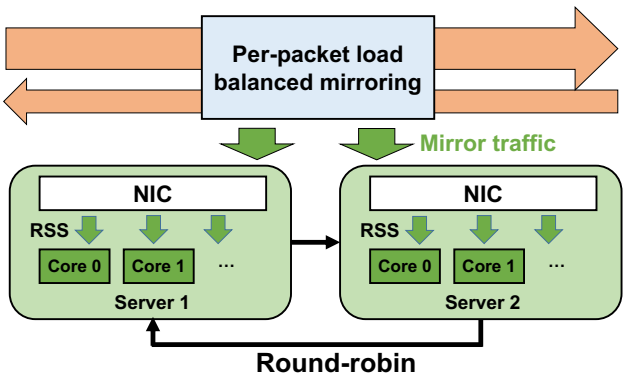


Figure 4: Per-packet load balancing to distribute mirrored packets across all the CPU cores of traffic dumpers.

Per-packet load balancing. In our initial design, we used two hosts to dump mirrored packets generated by the requester and the responder, respectively. Traffic dumping hosts had the same bandwidth capacity as traffic generation hosts. Despite our optimization efforts on the traffic dumping program, we still occasionally observed few packets discards (`rx_discards_phy` in our testbed) on the NIC when receiving line-rate mirrored packets. Although we can identify such *invalid* tests through integrity checks (§3.6), this degrades the efficiency of Lumina. Beyond that, this design requires powerful traffic dumpers which have enough capacity, e.g., network bandwidth, memory bandwidth, and CPU, to dump packets sent by traffic generators at line rate. This degrades the flexibility of hardware choices of Lumina.

Realizing above limitations, we develop a per-packet load balancing mechanism to evenly distribute mirrored packets across CPU cores of all the traffic dumpers. Instead of using two powerful hosts, we organize several hosts as a traffic dumper pool. The user can flexibly set up hosts as long as the total capacity of the traffic dumper pool is enough, e.g., process bi-directional line-rate traffic with minimum-sized packets. As shown in Figure 4, we use a weighted round-robin scheduler at the event injector to forward mirrored packets to different traffic dumpers based on their processing capacity. Though the requester and responder generate traffic at heterogeneous rates, the event injector can evenly distribute mirrored packets to homogeneous traffic dumpers.

At each traffic dumping host, we leverage Receive Side Scaling (RSS) to distribute packets across multiple CPU cores. However, RSS preserves flow to CPU affinity by hashing certain packet fields to select a CPU core. As a result, the CPU processing capacity depends on the number of flows in the test. To fully exploit CPU cores, we use the event injector to rewrite the UDP destination port to a random number. Note that UDP destination port number 4791 is reserved for

Name	Content Description
Dumped packets	Packets collected by all the hosts of the traffic dumper pool
Network stack counters	Link/Network/Transport layer counters
Traffic generator log	Application level metrics, e.g., goodput and message completion time.
Switch counters	TX/RX/mirrored packet counters for each switch port

Table 1: Results collected by the orchestrator

RoCEv2. By rewriting this well-known port number, we can create an illusion of many concurrent flows to RSS, thus maximizing CPU processing capacity.

After the traffic finishes, the orchestrator sends a TERM message to stop all the traffic dumpers. The traffic dumper catches the message, *recovers* the UDP destination port of all the packets, and writes packets to a disk file. We show the benefits of our traffic dumping design in §6.1.

3.6 Result Collection and Integrity Check

Once traffic generators stop, the orchestrator terminates all the other components and collects various result files given in Table 1. The orchestrator collects dumped packets from all the traffic dumpers, hardware network stack counters and traffic generator log files from traffic generation hosts, and switch counters from the event injector.

Upon collecting all the result files, the orchestrator reconstructs the packet trace from packets collected by all the hosts of the traffic dumper pool. Since the event injector maintains the mirror sequence number and stores this on the source MAC address field of every mirrored packet (§3.5), the orchestrator simply sorts all the packets based on their mirror sequence numbers.

After the packet trace reconstruction, the orchestrator runs an *integrity check* using the following four conditions to determine if the packet trace is complete without losses of any packets during traffic mirroring and dumping:

1. Mirror sequence numbers in the trace are consecutive.
2. Mirror timestamps in the trace keep increasing unless the timestamp wraps around.
3. The number of packets in the trace equals the total number of packets mirrored by the event injector.
4. The number of packets in the trace equals the total number of RDMA packets received by the event injector.

Only if all the conditions hold simultaneously, we can ensure that Lumina reconstructs a *complete* packet trace. Otherwise, Lumina reports an "invalid test" error to stop users from running any analysis on this test.

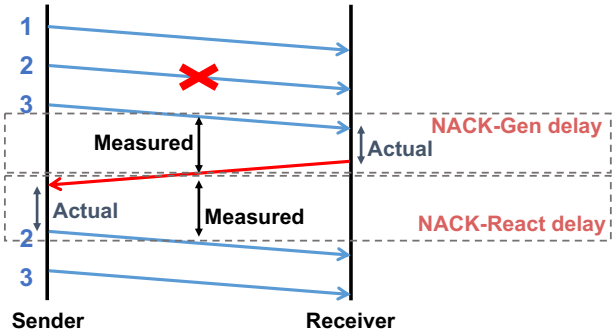


Figure 5: Retransmission latency breakdown.

4 Built-in Analyzers

Once tests pass integrity checks, users can parse reconstructed packet traces, log files, and counters to realize their customized requirements. For the ease of analysis in common cases, we provide a set of built-in analyzers for some widely used features in RDMA, e.g., Go-back-N retransmission [18] and ECN-based congestion control [38]. We use these analyzers in our later experiments (§6).

Retransmission logic. Retransmission is crucial for reliable delivery. Even in *lossless* networks, RDMA NICs (RNICs) still need effective retransmission *mechanisms* to handle non-congestion losses [18], no to mention lossy networks without Priority-based Flow Control (PFC).

To this end, we develop a retransmission logic analyzer to check if the RNIC's behaviors under packet losses follow the specifications, e.g., if the Go-back-N receiver generates a NACK packet correctly when it observes out of order arriving packets. To realize this, we translate the specification of Go-back-N, the de facto retransmission algorithm of RNICs [18], into a finite-state machine (FSM) and feed the reconstructed packet trace into this FSM. If the packet trace leads to a wrong state, we can determine the RNIC's retransmission implementation does not fully comply with the specification.

Retransmission performance. Many efforts have been made to enable RDMA over lossy networks [29, 39–41]. Lossy RDMA technologies heavily rely on efficient retransmission *implementations*. For example, when the RNIC receives a NACK/SACK packet, it should start the retransmission *immediately*, rather than wait for a long time.

To help users understand the retransmission performance of RNICs, we develop a retransmission performance analyzer. Note that this tool should be used in combination with the above retransmission logic analyzer to determine if the RNIC under test has a *correct* and *efficient* retransmission implementation. The retransmission performance analyzer can deal with both fast retransmissions (triggered by NACK/SACK) and timeout retransmissions (due to tail losses), and provide the performance breakdown to help users to identify the bot-

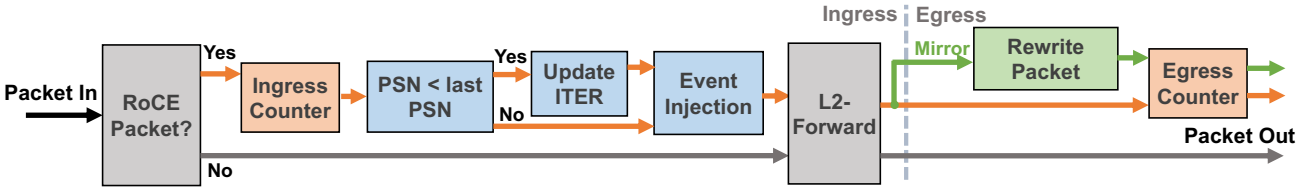


Figure 6: Pipeline layout of Lumina switch data plane.

tleneck.

Figure 5 shows how we break a NACK-triggered retransmission into two phases: NACK generation phase (receiver side) and NACK reaction phase (sender side). The NACK generation phase is the time between the receiver detects an out-of-order packet and it transmits a NACK. The NACK reaction phase is the time between the sender gets a NACK and it starts retransmission. We note that there is a half-RTT deviation since the timestamp is added by the middlebox rather than the end host. This deviation can be reduced by pre-measuring the RTT of the testbed.

Congestion notification. DCQCN [38] is the de facto RoCEv2 congestion control protocol implemented in Mellanox ConnectX-3 Pro and later RNICs. Once the DCQCN notification point (NP, receiver) receives ECN-marked packets, it notifies the reaction point (RP, sender) to reduce the rate using Congestion Notification Packets (CNPs). Recent Mellanox RNICs extend DCQCN to lossy networks. When the NP receives out-of-order packets, it generates *both* NACKs and CNPs to notify the RP to start the retransmission and lower the sending rate. In addition, to reduce the volume of CNP traffic and the CNP processing overhead, Mellanox RNICs also have a CNP pacer at the NP side, which determines the minimum interval between two consecutive generated CNPs [38].

In summary, the generation of CNPs depends on ECN-marked packets, packet losses and the CNP pacer configuration. We develop a CNP analyzer to check if CNPs are generated as expected under various network conditions and CNP pacer configurations.

Hardware network stack counter. We also develop a counter analyzer to check if counters of the hardware network stack are updated correctly. Currently, we support counters related to retransmission, timeout, congestion and packet corruption: counters of sent/received packets, sequence errors, out-of-sequences, timeouts (and retry), packets with iCRC errors, discarded packets, CNPs sent/handled.

5 Implementation

We have built a prototype of Lumina using Tofino-based programmable switches and commodity servers equipped with NVIDIA Mellanox RNICs.

The data plane of the event injector is implemented with 668 lines of code (LoC) in P4-16 [42] and is compiled to Intel Tofino ASIC [31] using BF SDE 9.4.0. Figure 6 shows the data plane pipeline layout of the event injector. The event injection module modifies packets and sets the drop flag at the ingress pipeline (§3.4). The events are injected by manipulating the packet field or intrinsic metadata: packet drop is enabled by setting The RoCE packets are mirrored from ingress to egress. The egress pipeline includes a module to rewrite packet fields of mirrored packets (§3.5). We track both incoming and outgoing RoCE packets, including mirrored packets, on each port for integrity check (§3.6). The switch control plane is implemented with 815 LoC in Python, which translates RPC calls to configure the data plane modules and dumps port counters after the experiment finishes.

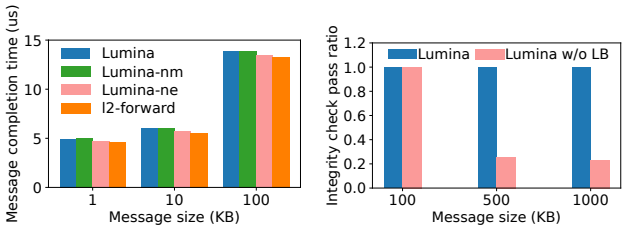
The traffic generator is implemented with 3116 LoC in C. It uses `Libibverbs` to generate RDMA traffic over RC transport. The traffic generator controls the GID (IPv4 address) associated with each QP to emulate traffic from multiple hosts. It reports total goodput and average request/message completion times for each QP.

The packet dumper is implemented with 584 LoC in C. To maximize the performance and efficiency with multi-core processing, it uses DPDK [43] and Receive Side Scaling (RSS) to dispatch the packets among the RX queues and cores. It buffers packets in the pre-allocated memory during the experiment and writes them to the disk upon receiving the TERM message from the orchestrator.

The orchestrator is implemented with 1198 LoC in Python, and the built-in analyzers are implemented on top of it with 1726 LoC in Python.

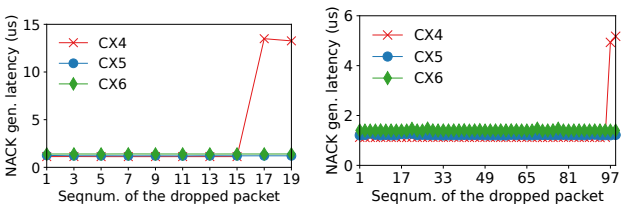
6 Evaluation and Case Studies

We test three NVIDIA Mellanox RDMA NICs: ConnectX-4 Lx MCX4131A 40GbE, ConnectX-5 MCX515A 100GbE, and ConnectX-6 Dx MCX623105AN 100GbE. In the rest of this section, we refer them as CX4 CX5 and CX6, respectively. The experiments are conducted on three testbeds: namely *cx4-testbed*, *cx5-testbed*, and *cx6-testbed*. Each testbed has four servers connected to an Edgecore Wedge100BF-65X switch with Intel Tofino ASIC, which works as the event injector. Each server in *cx4-testbed* has an 8-core CPU (Intel Xeon E5-2450) and a CX4 NIC. Each server in *cx5-testbed* has a 16-core CPU (Intel Xeon Silver 4216) and a CX5 NIC. Each server in *cx6-testbed* has a 16-core CPU (AMD EPYC 7302) and a CX6 NIC. All the servers run Ubuntu 20.04.3 LTS and MLNX_OFED_LINUX-5.4-3.0.3.0. For each testbed, we



(a) Overhead. (b) Load balance.

Figure 7: Microbenchmark results of Lumina's overhead (left) and load balance scheme (right).



(a) Message size = 20KB. (b) Message size = 100KB.

Figure 8: NACK generation latency v.s. sequence number of the dropped packet.

use two hosts to generate traffic and the rest hosts to dump packets. The MTU is set to 1024B for all the experiments. We use WRITE verb by default.

In the rest of this section, we first use microbenchmark experiments to evaluate the basic performance of Lumina. (§6.1). Then we present some interesting results we find on the target devices using Lumina (§6.2, §6.3, and §6.4).

6.1 Microbenchmark

We conduct microbenchmark experiments for two purposes: (1) measure the overhead of event injections and mirroring on the data path, and (2) evaluate the benefit of per-packet load balancing in traffic dumping.

Overhead of event injection and mirroring. We find that Lumina adds little overhead to under-test traffic on the data path. In this experiment, the traffic generator keeps sending 1000 messages with fixed size over a single QP, and we measure the average message completion time (MCT). The messages are sent back-to-back and we run the experiment with different message sizes: 1KB, 10KB and 100KB.

We use a simple L2-Forwarding program as a baseline. For Lumina, we keep all the match-action tables but disable the exact "drop" behavior to avoid retransmissions. We also implement two variants of Lumina: Lumina without event injection (Lumina-ne) and Lumina without mirroring (Lumina-nm) for comparison. As shown in Figure 7(a), event injection introduces negligible overhead. The MCT of Lumina is only 4.1–7.2% higher than that of Lumina-ne and l2-forward. In the meantime, mirroring actually adds almost no overhead to the under-test traffic as Lumina delivers almost the same message completion time with or without mirroring.

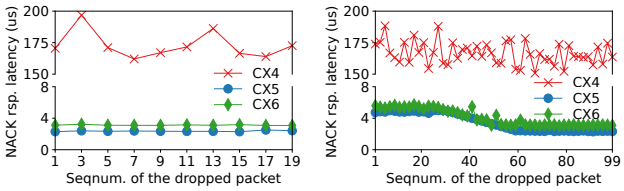
Benefit of per-packet load balancing. The efficiency of Lumina relies on the reliability of packet dumping. In the experiment, we show how our per-packet load balancing can guarantee efficiency by minimizing during traffic mirroring and dumping phase. We send messages with different sizes (100KB, 500KB and 1MB respectively) at line rate and let the switch mirror all the RoCE packets. A tweaked version of Lumina without per-packet load balancing (mirror traffic from an ingress port to a specific server) is implemented as a comparison. We run the experiment for 100 rounds, and measure the ratio of rounds that pass the integrity check. Figure 7(b) shows that as the message gets larger, the pass ratio of the tweaked version is as low as 23%. However, with per-packet load balancing, Lumina can achieve 100% pass ratio by capturing all the packets.

6.2 Fast retransmission

When packets in the middle are dropped, the receiver can observe out-of-order packets and generate NACK or SACK to trigger *fast* retransmissions. CX4, CX5 and CX6 adopt Go-back-N as the default fast retransmission algorithm. Here we use Lumina to evaluate RNICs' fast retransmission behaviors by deliberately dropping packets in the middle. First, we would like to note that all the RNICs pass our FSM-based retransmission logic check (§4) in a set of cunning and aggressive test cases. This indicates that their retransmission implementations strictly follow the specification. Then we present our findings about their fast retransmission performance.

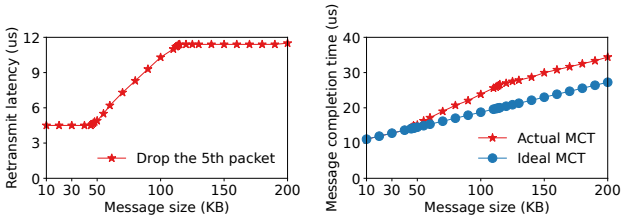
Setting. In this experiment, the traffic generator uses one connection to generate WRITE traffic with only a single outstanding request. For each message, we drop one packet with a different (relative) PSN. We fix the message size as 20KB and 100KB respectively. The experiment runs 1000 iterations and we compute the average latency. As shown in Figure 5, we break the Go-back-N retransmission latency into two parts: the NACK generation latency, and the NACK reaction latency. We show NACK generation latency results in Figure 8 and NACK reaction latency results in Figure 9.

Performance improvement from CX4 to CX5 and CX6. From the results, we can clearly observe *significant* improvement on retransmission performance from CX4 to CX5 and CX6. As shown in Figure 8, if we drop one of the last several packets (e.g., the 19-th packet in Figure 8(a)) for CX4, the NACK generation latency (13–1μs) is much larger than the latency when we drop other packets (about 1.1μs). While for CX5 and CX6, the NACK generation latency statically stays around 1.1μs. We can find things more interesting in



(a) Message size = 20KB. (b) Message size = 100KB.

Figure 9: NACK reaction latency v.s. sequence number of the dropped packet.



(a) Retransmission latency. (b) Message completion time.

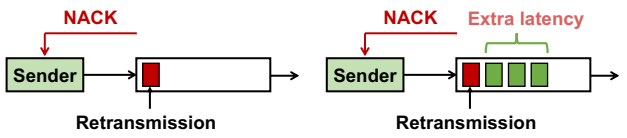
Figure 10: Effect of dropping the fifth packet for different message size on Mellanox CX6.

Figure 9: the NACK reaction latency of CX4 (150–200μs) is much larger than that of CX5 and CX6 (3–6μs). Combine the two parts together, CX5 and CX6 have a huge improvement ($\sim$ 200μs v.s. $\sim$ 4.5μs) over CX4 in terms of retransmission performance.

Remark 1. Above observations confirm Mellanaox's significant efforts to enable lossy RDMA, e.g., move retransmission from firmware (CX4) to hardware (CX5 and later) [44].

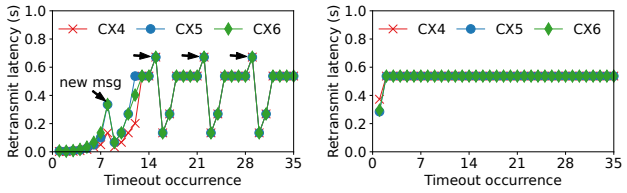
Retransmission might be blocked. While CX5 and CX6 deliver low NACK reaction latency, the NACK reaction latency still varies. As shown in Figure 9(b), when the message size is 100KB, the NACK reaction latency is about 6μs if we drop one of the first 20 packets, while it is about 3μs if we drop a latter packet.

We conduct another set of experiments to further investigate this behavior. This time, we would like to analyze the effect of message size. To do this, we keep dropping the fifth packet of a message and vary the message size from 10KB to 200KB. The packets are sent back-to-back. While CX5 and CX6 have very similar behavior in this experiment, here we only show the results for CX6. Figure 10(a) plots the retransmission latency for different message sizes when we drop the fifth packet. The retransmission latency starts growing when the message size is around 50KB, and becomes constant after the message size is larger than 120KB. This trend reflects in the message completion time (MCT). In Figure 10(b), we plot the "Actual MCT" based on the output from traffic generator, and the "Ideal MCT" which is the sum of MCT without retransmission and a fixed "ideal" latency 4.5μs. When message size is between 50KB to 120KB, the "Actual MCT" grows faster than "Ideal MCT", which is consistent with Figure 10(a).



(a) Pipeline is empty when retransmission happens. (b) Pipeline is not empty when retransmission happens.

Figure 11: Our hypothesis for NACK reaction behavior. The retransmitteed packets and the original packets share the same pipeline.



(a) Adaptive. (b) Non-adaptive.

Figure 12: Retransmission latency v.s. Timeout occurrence. *timeout* is set as 14, *retry_cnt* is set as 7. We send 5 WRITE messages and keep dropping the last packet of each message for 7 times.

We have a hypothesis for this anomaly: retransmitted packets and original packets share the same transmission pipeline on the NIC, and retransmitted packets cannot *preempt* original packets that are already in the pipeline. As a result, retransmitted packets may be delayed. We illustrate this hypothesis with Figure 11. When the sender transmits a packet, it pushes the packet to the tail of the pipeline no matter whether it's a retransmitted packet or a normal packet. If the message is short (e.g., 10KB), the pipeline is already empty when the retransmission happens (Figure 11(a)) because all the packets have been transmitted. Otherwise, the pipeline might still retain a few packets that haven't been sent when a packet is going to be retransmitted (Figure 11(b)), if the message is relatively large (e.g., 100KB). Note that this guess may or may not be correct, but we believe Lumina helps users build a clearer profile of what is going on inside the blackbox/hardware.

6.3 Timeout retransmission

When tail packets or retransmitted packets are dropped, the sender can only use retransmission timer to recover them. Inappropriate timeout values can lead to either spurious retransmissions or poor tail performance. In this section, we report our findings related to timeout retransmissions.

Setting. When creating QPs, `Libibverbs` provides an interface to configure the *timeout* and *retry_cnt* value. We use the default values. *timeout* is set to 14, meaning that the *mini-*

Setting		Max. Retry of Msg1–Msg5				
Mode	NIC	1st	2nd	3rd	4th	5th
Adaptive	CX4	13	8	7	7	7
Adaptive	CX5	13	8	8	8	8
Adaptive	CX6	13	8	8	8	8
Non-adaptive	All	7	7	7	7	7

Table 2: Maximum retry times. *retry_cnt* is 7. We send 5 WRITE messages and keep dropping the last packet of each message until it reports an error.

mum timeout is $4.096\mu s * 2^{timeout} = 0.0671s$ [45]. *retry_cnt* indicates the maximum number of times that the QP will try to resend the packets before reporting an error. We set it to 7.

In the experiment, we keep dropping the tail packet to trigger timeouts. Specifically, we use one connection to send 5 WRITE messages. For each message, the size is 10KB, and we drop the tenth (tail) packet for 7 (*retry_cnt*) times. We run the experiments in both adaptive [3] and non-adaptive mode (default) respectively. The results are shown in Figure 12.

Unexpected timeout value. When adaptive retransmission is enabled, the actual timeout value changes according to the packet loss frequency. It is worth noting that except for the first message, the first timeout of a message jumps to a high level unexpectedly (e.g., 0.267s for the second message and 0.671s for the latter messages, as pointed with black arrows in Figure 12(a)). Besides, the timeout value is not bounded by the pre-configured 0.0671s: for the first message, some of the retransmission timeouts are smaller than 0.0671s: 0.0056s, 0.0041s, 0.0084s, 0.0167s, 0.0251s, 0.0671s, and 0.1342s. For non-adaptive retransmission (Figure 12(b)), the timeout value obeys the specification: the first timeout is around 0.2–0.4s, then the following timeouts are static at about 0.537s. All these values are larger than the *minimum* timeout 0.0671s.

Retry times. We also find that the maximum retry times is correctly enforced on non-adaptive mode, but not on adaptive mode. We send 5 WRITE messages and keep dropping the last packet of each message until it reports an error. Table 2 shows the results for every message and every NIC in both adaptive mode and non-adaptive mode. All the NICs work correctly under non-adaptive mode as it can retry for 7 times exactly as *retry_cnt* specifies. However, under adaptive mode, the first message can retry for 13 times and the later message can retry for 7 or 8 times. The maximum retry attempts also vary between different NIC models. For example, for the third message, CX4 can retry 7 times, while CX5 and CX6 can retry 8 times. The result might make sense, since the timeout value is static in non-adaptive mode, so it is easier to ensure QoS by enforcing a fixed total retry count. While for adaptive mode, as the timeout is adaptively changing, it might be better to adjust the retry count according to other patterns (e.g., time spent for retransmission).

[3] Adaptive retransmission is a new feature in recent Mellanox RNICs to improve RDMA's resiliency over lossy networks.

NIC Model	Time of 1st ECN (μs)			Time of 2st ECN (μs)		
	QP1	QP2	QP3	QP1	QP2	QP3
CX4	10	31	45	616	630	643
CX5	8	17	18	246	255	258
CX6	9	16	17	246	254	259

Table 3: Timestamp of ECN-marked packets.

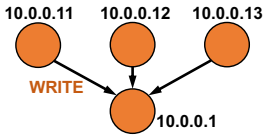


Figure 13: Simulated setting

NIC Model	Number of CNPs		
	QP1	QP2	QP3
CX4	2	2	2
CX5	2	2	2
CX6	2	0	0

Table 4: CNP interval.

Remark 2. Adaptive retransmission is a new feature that has been rarely explored in the research community. We cannot find a public available specification that accurately explains its behaviors. Despite this, Lumina can still give us some viability into its micro-behaviors.

6.4 CNP generation

In this section, we focus on CNP generation which is crucial for congestion control [38]. More specifically, we try to resolve the following doubt: "Will the CNP generation of one connection be affected by ECNs or losses from another connection".

Setting. We conduct this experiment using three connections, each sending a 1MB WRITE message. All the three connections have the same responder address (10.0.0.1) but different requester addresses (10.0.0.11, 10.0.0.12, 10.0.0.13) to simulate a scenario that three requesters are sending WRITE traffic to one responder (as shown in Figure 13). We denote the QPs at the responder side as QP1, QP2 and QP3 respectively. Mellanox NICs use a parameter named *min_time_between_cnps* to control the CNP generation interval. In this experiment, we set *min_time_between_cnps* to 50μs. The three connections are sending traffic simultaneously, and we mark ECN for the 50-th packet and the 950-th packet of each connection. We disable the DCQCN reaction functionality in the requester so that it won't adjust the sending rate upon receiving CNPs. After the experiment, we check how many CNPs does each (sender) QP receives. Table 3 and Table 4 present the results.

Per-port interval or per-dstIP interval. As shown in the Table 3, the first ECN-marked packet of each connection arrives at a relatively close time frame (within 50μs). Then after a long time period (about 600μs for *cx4-testbed*, 200μs for *cx5-testbed* and *cx6-testbed*), the second ECN-marked packet arrives. Table 4 shows how many CNPs each (receiver) QP

sends out on each testbed. It's surprising to see that these NICs deliver different results. For CX6, only QP1 sends two CNPs while QP2 and QP3 do not reply to the ECN-marked packets. However, for CX4 and CX5, each of the three QPs sends two CNPs. According to our conversation with the vendor, there are at least two modes to enforce CNP intervals: per-port and per-destination IP. With per-port mode, all connections share a same CNP timer. While with per-destination IP mode, only the connections with the same destination IP share a same CNP timer. Different NICs may use different modes. It explains what we observe: CX6 enforces CNP interval with per-port mode, so that after a CNP is generated for an ECN-marked packet at time 9μs, the next CNP should be generated at least after 50+9=59μs. As a result, only QP1 sends two CNPs. While CX4 and CX5 use per-destination IP mode, the three QPs use separate timers and each can send two CNPs. While this is not necessarily a bug, it causes bias when users try to understand the NICs' behaviors. We hope Lumina could mitigate such bias for users.

7 Discussion

Lossy RoCE. RoCEv2 originally relied on PFC to provide a lossless fabric. However, there has been a lot of discussion on lossy RoCE network [2, 29, 39] and lossy RDMA is already supported on recent RDMA NICs. While both industry and academia are taking the temperature of deploying lossy RDMA [2, 39], we believe that having a deep understanding of RDMA NICs' retransmission behavior and performance is essential. By anatomizing the retransmission process, Lumina can help us gain deep understanding about the micro-behaviors behind it.

Flexibility. We choose programmable switch as our middlebox implementation solution for Lumina because it provides the set of easy-to-use functionalities we need and it is accessible. However, our middlebox design is not restricted to programmable switch. The middlebox can be any programmable high-performance hardware or software. One of our future visions is to deploy Lumina with a FPGA board so that it is more light-weight and users can directly plug-and-test.

Extensibility. We do not intent to implement our traffic generator to cove all the traffic patterns for various application scenarios. Instead, we design Lumina in the extensible way. Users can customize their own traffic generators to test their own application-specific traffic patterns while no changes for other components are needed. Lumina is also extensible in terms of different transport protocols. While we start with RDMA as the target, Lumina can be extended with reasonable effort to test and measure other hardware network stacks.

Fuzzing and auto testcase generation. Lumina is suitable for integration with fuzzing or auto testcase generation modules as it provides well-structured input and output. It is also possible to leverage reinforcement learning to find potential issues by defining anomaly as rewards. We leave these as future work.

8 Related Work

Network protocol testing. There are many tools and research works focusing on testing network protocol implementation [6–9, 28]. Among them, packetdrill [6] is most related. Packetdrill is a scripting tool that enables tests for entire TCP/IP network stack. To interact with the local and remote network stack, packetdrill uses libpcap and TUN device as a "shim layer" to inject or consume packets. Packetdrill has also been utilized to test QUIC [7] and for educational purposes [46]. Similarly, Packet Shell [8] and Orchestra [9] also test the conformance of TCP implementation to its specification by injecting packets or events. DETER [28] focuses on deterministic TCP replay to reproduce performance problems and support tracing of TCP executions. Compared to them, Lumina focuses on hardware offloaded network stacks.

Performance anomaly and security of RDMA. With the wide adoption of RDMA in datacenters, the performance anomaly and security of RDMA have drawn a lot of attention. Collie [47] is a tool to systematically find RDMA performance anomalies caused by NIC resource contention. Kalia et al. [48] studied the scalability limits of RDMA: RDMA caches connection states in NICs which leads to scalability bottlenecks. Rothenberger et al. [49] demonstrated that the design and implementation of IB-capable NICs contain vulnerabilities and design flaws. Compared to them, Lumina focuses on transport protocol behaviors.

Network testing with programmable networks. Reconfigurable and programmable networks are becoming more relavant than ever before. In recent years, there are many works using smartNICs or programmable switches for applications acceleration [50, 51], network telemetry [52, 53], and achieve novel applications [54, 55]. Among them, Hypertester [54] and IMap [55] test the network environment by injecting packets into the network using switches. Hypertester applies programmable switches as network testers to generate and capture test traffic at line rate, and then use switch CPU to analyze the statistics. IMap implements a network scanner with programmable switches. It uses switch CPU to generate probe packets. The switch data plane is responsible for replicating the probe packets by recirculation. Lumina is the first work that leverages network programmability to test hardware network stacks, to the best of our knowledge.

9 Conclusion

We present Lumina, a tool to test the correctness and performance of hardware network stack implementation. We find several interesting micro-behaviors on RDMA NICs using Lumina. We believe Lumina can help network developers understand the micro-behaviors of complex hardware network stacks.

References

[1] L. Shalev, H. Ayoub, N. Bshara, and E. Sabbag, "A cloud-optimized transport protocol for elastic and scalable hpc," *IEEE/ACM MICRO*, 2020.

[2] Y. Gao, Q. Li, L. Tang, Y. Xi, P. Zhang, W. Peng, B. Li, Y. Wu, S. Liu, L. Yan, *et al.*, "When cloud storage meets {RDMA}," in *USENIX NSDI*, 2016.

[3] A. Kalia, M. Kaminsky, and D. G. Andersen, "Using RDMA efficiently for key-value services," in *ACM SIGCOMM*, August 2014.

[4] Y. Jiang, Y. Zhu, C. Lan, B. Yi, Y. Cui, and C. Guo, "A unified architecture for accelerating distributed {DNN} training in heterogeneous {GPU/CPU} clusters," in *USENIX OSDI*, November 2020.

[5] J. Xue, Y. Miao, C. Chen, M. Wu, L. Zhang, and L. Zhou, "Fast distributed deep learning over rdma," in *EuroSys*, March 2019.

[6] N. Cardwell, Y. Cheng, L. Brakmo, M. Mathis, B. Raghavan, N. Dukkipati, H.-k. J. Chu, A. Terzis, and T. Herbert, "packetdrill: Scriptable network stack testing, from sockets to packets," in *USENIX ATC*, 2013.

[7] V. Goel, R. Paulo, and C. Paasch, "Testing quic with packetdrill," in *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*, 2020.

[8] S. Parker and C. Schmechel, "The packet shell protocol testing tool," *Software distribution at http://playground.sun.com/psh*, 1996.

[9] S. Dawson, F. Jahanian, and T. Mitton, "Experiments on six commercial tcp implementations using a software fault injection tool," *Software: Practice and Experience*, 1997.

[10] "OFED perftest," 2022. https://github.com/linux-rdma/perftest.

[11] J. S. Chase, A. J. Gallatin, and K. G. Yocum, "End system optimizations for high-speed tcp," *IEEE Communications Magazine*, vol. 39, no. 4, pp. 68–74, 2001.

[12] A. Menon and W. Zwaenepoel, "Optimizing {TCP} receive performance," in *2008 USENIX Annual Technical Conference (USENIX ATC 08)*, 2008.

[13] "Segmentation Offloads," 2022. https://www.kernel.org/doc/html/latest/networking/segmentation-offloads.html.

[14] "Chelsio Terminator 5 ASIC," 2022. https://www.chelsio.com/terminator-5-asic/.

[15] "Broadcom NetXtreme Ethernet Adapters," 2022. https://www.broadcom.com/how-to-buy/hardware-partners/ethernet-network-adapters/broadcom.

[16] T. S. Woodall, G. M. Shipman, G. Bosilca, R. L. Graham, and A. B. Maccabe, "High performance rdma protocols in hpc," in *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, Springer, 2006.

[17] G. F. Pfister, "An introduction to the infiniband architecture," *High performance mass storage and parallel I/O*, 2001.

[18] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn, "Rdma over commodity ethernet at scale," in *ACM SIGCOMM*, 2016.

[19] "NVIDIA ConnectX-6 Dx," 2022. https://www.nvidia.com/en-us/networking/ethernet/connectx-6-dx/.

[20] "Intel Ethernet Network Adapter E810," 2022. https://www.intel.com/content/www/us/en/products/details/ethernet/800-network-adapters/e810-network-adapters.html.

[21] "Broadcom M1100G16 100GbE OCP 2.0 Adapter," 2022. https://www.broadcom.com/products/ethernet-connectivity/network-adapters/100gb-nic-ocp/m1100g.

[22] "Marvell FastLinQ 41000 Series Ethernet NICs," 2022. https://www.marvell.com/products/ethernet-adapters-and-controllers/41000-ethernet-adapters.html.

[23] "OneConnect OCe14000-Series Adapters," 2022. https://docs.broadcom.com/doc/12356182.

[24] R. Recio, B. Metzler, P. Culley, J. Hilland, and D. Garcia, "A remote direct memory access protocol specification," tech. rep., RFC 5040, October 2007.

[25] V. Cerf, S. Burleigh, A. Hooke, L. Torgerson, R. Durst, K. Scott, K. Fall, and H. Weiss, "Delay-tolerant networking architecture," 2007.

[26] V. Paxson, "Automated packet trace analysis of tcp implementations," in *ACM SIGCOMM*, 1997.

[27] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough, "Rigorous specification and conformance testing techniques for network protocols, as applied to tcp, udp, and sockets," in *Proceedings of the Conference on Applications, technologies, architectures, and protocols for computer communications*, 2005.

[28] Y. Li, R. Miao, M. Alizadeh, and M. Yu, "{DETER}: Deterministic {TCP} replay for performance diagnosis," in *USENIX NSDI*, February 2019.

[29] A. Shpiner, E. Zahavi, O. Dahley, A. Barnea, R. Damsker, G. Yekelis, M. Zus, E. Kuta, and D. Baram, "Roce rocks without pfc: Detailed evaluation," in *Proceedings of the Workshop on Kernel-Bypass Networks*, 2017.

[30] D. Mudigere, Y. Hao, J. Huang, Z. Jia, A. Tulloch, S. Sridharan, X. Liu, M. Ozdal, J. Nie, J. Park, *et al.*, "Software-hardware co-design for fast and scalable training of deep learning recommendation models," *arXiv preprint arXiv:2104.05158*, 2021.

[31] "Intel Tofino," 2022. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html.

[32] "Intel Tofino 2," 2022. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html.

[33] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, *et al.*, "Azure accelerated networking:{SmartNICs} in the public cloud," in *USENIX NSDI*, 2018.

[34] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore, "Netfpga sume: Toward 100 gbps as research commodity," *IEEE/ACM MICRO*, 2014.

[35] Y. Li, G. Kumar, H. Hariharan, H. Wassel, P. Hochschild, D. Platt, S. Sabato, M. Yu, N. Dukkipati, P. Chandra, *et al.*, "Sundial: Fault-tolerant clock synchronization for datacenters," in *USENIX OSDI*, November 2020.

[36] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, *et al.*, "Packet-level telemetry in large datacenter networks," in *ACM SIGCOMM*, August 2015.

[37] J. Rasley, B. Stephens, C. Dixon, E. Rozner, W. Felter, K. Agarwal, J. Carter, and R. Fonseca, "Planck: Millisecond-scale monitoring and control for commodity networks," *SIGCOMM CCR*, August 2014.

[38] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang, "Congestion control for large-scale rdma deployments," *SIGCOMM CCR*, August 2015.

[39] R. Mittal, A. Shpiner, A. Panda, E. Zahavi, A. Krishnamurthy, S. Ratnasamy, and S. Shenker, "Revisiting network support for rdma," in *ACM SIGCOMM*, 2018.

[40] Y. Lu, G. Chen, Z. Ruan, W. Xiao, B. Li, J. Zhang, Y. Xiong, P. Cheng, and E. Chen, "Memory efficient loss recovery for hardware-based transport in datacenter," in *Asia-Pacific Workshop on Networking*, August 2017.

[41] "NVIDIA Zero Touch RoCE (ZTR)," 2021. https://tinyurl.com/yc6tnv7h/.

[42] "P4-16 Language Specification," 2021. https://p4.org/p4-spec/docs/P4-16-v1.2.2.html.

[43] "Intel data plane development kit (dpdk)," 2018. http://dpdk.org/.

[44] "Hardware Offloading To RDMA and Beyond." https://conferences.sigcomm.org/sigcomm/2018/files/slides/kbnet/keynote_2.pdf.

[45] "InfiniBand Architecture Specification Volume 1 Release 1.5," 2021. https://www.infinibandta.org/ibta-specification/.

[46] O. Bonaventure, Q. De Coninck, F. Duchêne, A. Gego, M. Jadin, F. Michel, M. Piraux, C. Poncin, and O. Tilmans, "Open educational resources for computer networking," *SIGCOMM CCR*, August 2020.

[47] X. Kong, Y. Zhu, H. Zhou, Z. Jiang, J. Ye, C. Guo, and D. Zhuo, "Collie: Finding performance anomalies in {RDMA} subsystems," in *USENIX NSDI*, April 2022.

[48] A. Kalia, M. Kaminsky, and D. Andersen, "Datacenter {RPCs} can be general and fast," in *USENIX NSDI*, February 2019.

[49] B. Rothenberger, K. Taranov, A. Perrig, and T. Hoefler, "{ReDMArk}: Bypassing {RDMA} security mechanisms," in *USENIX Security*.

[50] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "NetCache: Balancing key-value stores with fast in-network caching," in *ACM SOSP*, October 2017.

[51] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang, "KV-Direct: High-performance in-memory key-value store with programmable NIC," in *ACM SOSP*, October 2017.

[52] C. Kim, A. Sivaraman, N. Katta, A. Bas, A. Dixit, and L. J. Wobker, "In-band network telemetry via programmable dataplanes," in *ACM SIGCOMM*, August 2015.

[53] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger, "Sonata: Query-driven streaming network telemetry," in *ACM SIGCOMM*, August 2018.

[54] D. Zhang, Y. Zhou, Z. Xi, Y. Wang, M. Xu, and J. Wu, "Hypertester: high-performance network testing driven by programmable switches," *IEEE/ACM Transactions on Networking*, 2021.

[55] G. Li, M. Zhang, C. Guo, H. Bao, M. Xu, H. Hu, and F. Li, "{IMap}: Fast and scalable {In-Network} scanning with programmable switches," in *USENIX NSDI*, April 2022.