Anytest: Localizing the Root Cause of Hardware Transport Performance Anomalies

Zhaochen Zhang[1], Jiaqi Gao[2], Sheng Cheng[2], Peiwen Yu[2], Feiyang Xue[2], Chang Liu[2], Boliang Liu[3], Kun Liu[2], Yubin Li[2], Rui Li[1], Li Wang[1], Peirui Cao[1], Qingkai Meng[1], Guihai Chen[1], Gang Lu[2], Bo Shi[2], Shuguang Cheng[2], Shu Ma[2], Yongqing Xi[2], Binzhang Fu[2], Dennis Cai[2], Chen Tian[1]
[1]State Key Laboratory of Novel Software Technology, Nanjing University [2]Alibaba Cloud [3]Fudan University

ABSTRACT

RoCEv2-based RDMA fabrics are the backbone of modern high-performance data-center workloads, yet large-scale deployments increasingly suffer transport-layer network performance anomalies (NPAs) such as throughput collapse, persistent unfairness, and latency inflation even without link failures. Localizing root causes of NPAs is one of today's hardest operational challenges: RoCEv2 transport logic is offloaded into proprietary NIC/switch hardware with limited observability; available counters miss μs-scale dynamics; and similar symptoms can originate from sender, receiver, or switch behaviors in the tightly coupled RoCEv2 system.

We present Anytest, an *in-situ* black-box testing tool that localizes root causes of transport-layer NPAs on commodity RoCEv2 RNICs and Ethernet switches without re-cabling or hardware modification. Anytest decomposes RoCEv2 network system into logical roles and isolates the hardware under test by emulating the other roles with protocol-correct DPDK endpoints. This enables deterministic injection of transport events and μs-resolution measurements. We overcome non-trivial technical challenges to implement Anytest's DPDK-based endpoints, which realize protocol correctness while enforcing μs-level packet timing at the hardware line rate. Integrated into a trace–reproduce–localize workflow, Anytest has been deployed in production for ~1 year, reducing mean localization effort to 3.1 person-hours.

CCS CONCEPTS

• **Networks** → **Network performance modeling**; **Network experimentation**; **Network performance analysis**;

KEYWORDS

RDMA, RoCEv2, network performance anomaly, DPDK

ACM Reference Format:
Zhaochen Zhang[1], Jiaqi Gao[2], Sheng Cheng[2], Peiwen Yu[2], Feiyang Xue[2], Chang Liu[2], Boliang Liu[3], Kun Liu[2], Yubin Li[2], Rui Li[1], Li Wang[1], Peirui Cao[1], Qingkai Meng[1], Guihai Chen[1], Gang Lu[2], Bo Shi[2], Shuguang Cheng[2], Shu Ma[2], Yongqing Xi[2], Binzhang Fu[2], Dennis Cai[2], Chen Tian[1]. 2026. Anytest: Localizing the Root Cause of Hardware Transport Performance Anomalies. In *ACM SIGCOMM 2026 Conference (SIGCOMM '26), August 17–21, 2026, Denver, CO, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3789240.3829125

SIGCOMM '26, August 17–21, 2026, Denver, CO, USA

ACM ISBN 979-8-4007-2467-1/26/08.
https://doi.org/10.1145/3789240.3829125

1 INTRODUCTION

Remote Direct Memory Access (RDMA) and hardware-offloaded implementations such as RoCEv2 have become the foundational interconnect for modern high-performance data center workloads [5, 13, 14]. By bypassing the kernel and offloading the transport stack to the Network Interface Card (NIC), RDMA delivers sub-microsecond latency and near-zero CPU overhead—critical for distributed storage and Large Language Model (LLM) training [12, 16, 27] and serving [20, 28, 29]. However, as RDMA deployments scale, they increasingly encounter transport-layer Network Performance Anomalies (NPAs): unexpected throughput collapse, persistent unfairness, and latency inflation, even without link failures or hardware errors.

Diagnosing NPAs in production is one of today's hardest operational challenges. Unlike TCP, whose behavior can be inspected via kernel tracing (e.g., eBPF), RoCEv2 transport logic resides in proprietary "black-box" hardware/firmware, creating a severe observability gap. Operators can often detect that an NPA is happening using coarse telemetry, but localizing the root cause remains a manual process that can take weeks.

Localization is difficult for three reasons. First, monitoring operates at 10 ms–1 s granularity, while hardware state machines react at sub-microsecond timescales; micro-bursts and timing deviations are averaged out. Second, RoCEv2 is a tightly coupled distributed system spanning sender, receiver, and switches, so the same symptom (e.g., low throughput) may originate from an aggressive sender, a slow-reacting receiver, or a misconfigured switch—yet counters rarely provide causality. Third, many NPAs are in situ, tied to specific production topologies and firmware versions, making them hard to reproduce in offline testbeds.

Existing approaches do not meet operational needs. Counter-based inference is lightweight but lacks packet-level causality and µs-scale visibility. Commercial and programmable-hardware testers [6, 7, 31] can inject traffic and capture packets, but are largely offline and impractical for live production due to physical access and re-cabling. Recent test-suite generation and fuzzing [17, 18] can detect broad anomaly classes, but typically localize only coarsely and rely on second-level metrics (e.g., throughput or FCT), which miss transport behaviors driven by microsecond timing and hidden device state.

This paper presents Anytest, an in-situ black-box testing tool for localizing transport-layer NPAs in Commercial Off-The-Shelf (COTS) RoCEv2 RNICs and Ethernet switches. Anytest is based on a simple principle: when transport logic is a black box, localization

must rely on externally controlled experiments that (i) generate protocol-correct stimuli, (ii) deterministically inject transport-layer events, and (iii) measure on-wire responses at packet granularity and µs resolution—without hardware modification or disruptive re-cabling. The core insight is a role-based decomposition of the protocol. We decompose RoCEv2 into three logical roles: the Reaction Point (RP) at the sender, the Notification Point (NP) at the receiver, and the Congestion Point (CP) at the switch. By emulating two roles in software while treating the third as the device under test (DUT), Anytest isolates participants and enables systematic attribution of anomalies.

Implementing Anytest requires DPDK-based software endpoints that remain protocol-correct while enforcing microsecond timing at hardware line rate. We address this with three techniques: (1) an $O(1)$ fast-path Invariant CRC (ICRC) update enabling line-rate packet generation on a single CPU core; (2) padding-based gap control that provides µs-level inter-packet interval precision without sacrificing throughput; and (3) a high-precision measurement pipeline anchored by hardware timestamps to capture transient transport dynamics.

In deployment, Anytest is integrated into an trace–reproduce–localize workflow. We use an RDMA call-tracing library to capture real application behavior without code changes, replay/mutate traces in place, and then localize root causes via role-isolated tests that "open" the offloaded transport through controlled experiments. Over roughly one year of production use, Anytest has localized diverse NPAs, including congestion collapse, incast unfairness due to unexpected CNP timer aggregation, firmware-dependent congestion-control regressions, latency inflation under ETS scheduling, and switch scheduling imbalance in Virtual Output Queue (VOQ) designs. Empirically, Anytest reduces mean localization cost to 3.1 person-hours, and a regression test suite resolves a majority of incidents directly.

This paper makes the following contributions:

- We characterize the practical difficulty of localizing transport-layer NPAs in hardware-offloaded RoCEv2 deployments and distill the key capabilities required by a localization tool: *in-situ* testing, μs-level resolution, packet-level event injection, and role-based decomposition.
- We design Anytest, a DPDK-based black-box testing framework that surrounds a DUT with protocol-correct emulators and performs role-isolated tests for RP/NP/CP, enabling causal and systematic NPA root cause localization.
- We present the techniques that enable Anytest to operate at line rate while supporting precise event injection and high-fidelity measurement, including an $O(1)$ ICRC fast path and padding-based gap control.
- We report production deployment experience, including a regression-test-driven localization workflow and case studies that demonstrate how Anytest pinpoints real-world root causes that are opaque to counter-based telemetry and offline testing.

Anytest is open-sourced at [3]. *This work does not raise any ethical issues.*

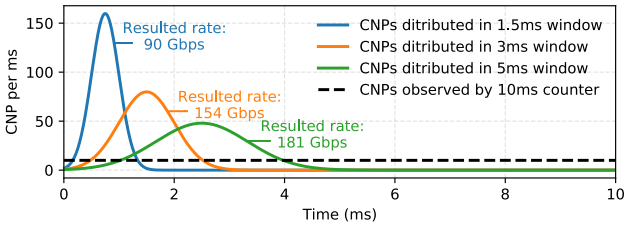


Figure 1: μs-level CNP distribution can significantly affect the sending rate, yet the telemetry counter at 10 ms granularity cannot distinguish.

2 BACKGROUND

Remote Direct Memory Access (RDMA), along with hardware-offloaded implementations such as RoCEv2, SOLAR [22], and SRD [30], is the backbone of modern high-performance applications, including HPC, storage, and especially AI workloads such as LLM training and serving. Compared with kernel-based transports (TCP/UDP), RDMA achieves superior performance with near-zero CPU overhead.

Hardware-offloaded RDMA requires tight coordination among sender/receiver NICs and switches along the path. NICs implement hierarchical QP and packet scheduling to balance fairness and throughput; switches route packets and mark ECN under congestion; NICs then adjust sending rates using congestion control to limit congestion and queuing delay. As RDMA adoption grows, production traffic becomes more complex and triggers transport-layer NPAs—throughput/latency degradation, unfairness, and congestion collapse—without link or switch failures.

Diagnosing NPAs is among the hardest problems in day-to-day cluster operations. An incident often begins with an alarm (e.g., elevated loss) or customer complaints about degraded application performance. For example, one cluster running a storage service experienced long queuing delay, while monitoring detected packet drops at the Top-of-Rack (ToR) switch. Even approaching this case is challenging:

- **Understand traffic behavior.** RDMA libraries commonly run in user space to avoid kernel context switching. We cannot capture the application's behavior using common tracing tools such as eBPF. Customer applications may be closed-source, requiring extensive coordination to infer traffic behavior. In our example, the loss spike occurred when multiple clients issued requests to one storage backend.
- **Hard to reproduce offline.** NPAs often depend on specific cluster settings—topology, switch configuration, and NIC firmware—making faithful offline reproduction difficult. Diagnosis must often occur in production, with limited flexibility and time windows.
- **Coarse-grained monitor.** Hardware transport runs fast and is black-boxed/underdocumented; telemetry counters are coarse (up to 10 ms granularity), while congestion control can react at sub-microsecond timescales. Bridging this four-order-of-magnitude gap is inherently difficult. As we show in Figure 1, the same coarse-grained CNP arrival rate (1,000 packets per 10 ms) can cause up to a 2× throughput difference for a single RDMA connection, depending on the µs-scale arrival pattern.

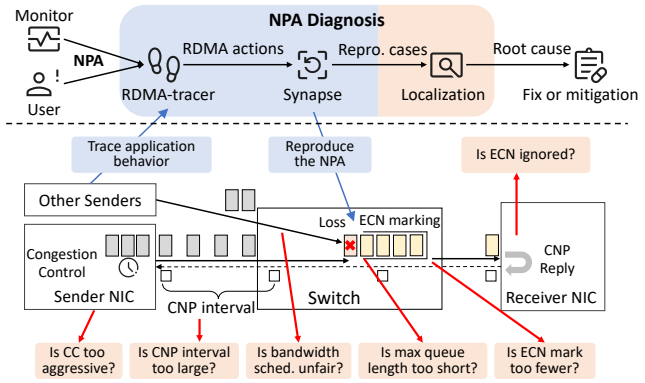


Figure 2: (up) An overview of the NPA diagnosis workflow. (down) Hardware-offloaded RDMA is a tightly coupled system, where an NPA can be triggered by any component.

- **Tightly coupled distributed system.** NPAs reflect interactions among application behavior, congestion control, and switch queuing. Multiple root causes can produce the same symptom (*e.g.,* low throughput due to aggressive ECN marking thresholds or conservative congestion control parameters), and limited visibility into both hardware and applications makes attribution especially challenging.

Without proper analysis tools, each NPA can easily cost an experienced engineer weeks to resolve. However, to the best of our knowledge, none of the existing approaches can help alleviate this situation. To fight against the NPA problems, we have built and deployed an RDMA tracing, reproducing, and troubleshooting workflow. As shown in Figure 2, the workflow involves three individual tools.

- *RDMA-tracer* is an RDMA hijacking library that leverages the `LD_PRELOAD` to create an interception layer between the user application and the RDMA library upon user application launches. The *RDMA-tracer* can record each and every RDMA calls in to a trace file, including the QP metadata, timestamp, and data sizes. It requires zero application code modification or integration effort.
- *Synapse* is an RDMA traffic generator that replays the collected or manually generated traces in the production environment. It allows us to remove complicated application logic and focus on the traffic behavior. We can also manipulate the user traffic to create different variants to find other potential NPAs.
- *Anytest* is a DPDK-based NPA localization tool that assists in opening up the hardware-offloaded transport's black box and locating the NPA root cause. It creates an emulated environment surrounding the DUT and injects protocol-correct packets, controlled transport events, and records packet-level measurements at μs granularity. Fine-grained DUT behavior measurement enables operators to efficiently narrow down the suspected root cause.

Together, this workflow helps us capture application behavior, reproduce NPAs in place, and measure hardware behavior at µs resolution, reducing diagnosis time from weeks to days or hours. Since *RDMA-tracer* and *Synapse* are more engineering-focused, we present them in Appendix B and C; **the paper primarily focuses on Anytest's design and deployment experience**.

Method	In-situ localization	Resolution Tput	Resolution Event	Inject	Decompose
Inference from counters	●	sec	Flow	○	○
Hardware testers	○	μs	Pkt	●	◐
Test-suite generators	●	sec	Flow	◐	○
Anytest (ours)	●	μs	Pkt	●	●

●: Fully supported, ◐: Partially supported, ○: Not supported.

Table 1: Comparison of NPA localization approaches.

2.1 NPA Root Cause Localization

In practical network operations, we observe that even when an application's network behavior can be captured and test cases can be constructed to reproduce NPAs, identifying the root cause remains difficult. This is because hardware-offloaded networking consists of multiple tightly coupled black-box components. Consequently, NPAs with identical symptoms and reproducible test cases may stem from diverse underlying causes.

Take the switch congestion packet loss, one of the most common NPAs, as an example (Figure 2), we can usually reproduce the problem by creating an N-to-1 incast, but the actual root cause may reside in many places. Typical examples include: (1) an aggressive congestion control setting, (2) a misconfigured switch queue that has a shorter maximum queue length, (3) a long CNP interval configuration that replies too few CNPs, and (4) packet losses cause out-of-order packet arrival, and the receiver NIC ignores ECN after that, worsening the congestion.

We believe the location tool should provide four capabilities to localize the NPAs effectively.

- **In-situ testing.** Many NPAs are tied to the specific cluster setting; the tool has to be portable to deploy in any setting.
- **μs-level resolution.** Hardware-offloaded RDMA transports react to network events at μs granularity. Meanwhile, a single critical control packet (NACK/CNP) can substantially affect transport behavior. The localization tool has to catch up with such dynamics.
- **Packet-level event injection.** Some tricky NPAs are only reproducible under specific packet orderings or timing. Active injection of the corresponding events is necessary for observing the problem and finding the NPA's trigger.
- **Decompose the interaction between hardware.** The transport's overall behavior is jointly decided by all hardware on the path. Root cause localization is insurmountable without decomposition.

Such complex requirements rule out all existing approaches (Table 1): **(i) Inference from coarse-grained counters.** In current practice, engineers heavily rely on RNIC/switch counters. However, these counters are typically *coarse* (often 10 ms to 1 s level) and *aggregated* (e.g., per-port/per-queue), which averages out microbursts, transient queues, and control packet timing that largely determine transport dynamics. More importantly, counter correlations rarely establish causality. For example, in an unfairness NPA we observe that one sender's `np_cnp_received` counter indicates

a higher number of processed CNPs. However, we cannot explain whether this stems from the switch marking more ECNs or the receiver generating more CNPs for this sender, as both metrics (ECN marked at switch and CNP sent at receiver) are available only at port granularity rather than flow granularity.

(ii) Hardware testers. Commercial testers (e.g., Viavi, Keysight) and programmable-hardware-based testers [6, 7, 31] provide packet-level measurement and controlled traffic patterns. However, they are fundamentally *offline*: they require physical access to the problematic cluster and re-cabling, which is operationally prohibitive for live production clusters.

(iii) Test-suite generators. Recent work proposes generating test suites via heuristics or fuzzing to expose device issues [17, 18]. These approaches are valuable for *detecting* classes of anomalies online, but they face two limitations for *localization*. First, their *scope* is often constrained by the modeled failure space (e.g., targeting QoS [17] or sender-rate anomalies [18]). Second, their *observability* is typically based on coarse metrics (e.g., second-level throughput or flow completion times), which can miss micro-behaviors rooted in hardware timing and state machines; as a result, they often indicate that "something is wrong" without pinpointing which participant and which transport logic is responsible.

3 DESIGN

This section presents the design of Anytest, a black-box testing tool for localizing root causes of NPAs in COTS hardware RoCEv2 transports. We first introduce Anytest's core idea, architecture, and capability model (§3.1), then describe three role-isolated test scenarios that decouple the DUT from cross-component interactions (§3.2).

3.1 Anytest Overview

Key idea. To make the *black-box* logic of COTS hardware observable to operators, Anytest adopts an approach that *surrounds* the device under test (DUT) with white-box software endpoints implemented in DPDK. These DPDK endpoints generate protocol-correct packets, inject controlled transport events, and record packet-level measurements at μs granularity, enabling externally controlled experiments without any hardware modification.

Role-isolated test. Anytest structures localization around three RoCEv2 transport roles: a *reaction point* (RP, sender-side congestion reaction), a *notification point* (NP, receiver-side congestion notification), and a *congestion point* (CP, switch-side congestion signaling). For each test, the DUT implements exactly one role, while DPDK emulates the other two roles. This role-based decomposition isolates the DUT's contribution from cross-party interactions and turns black-box behaviors into testable hypotheses, enabling systematic localization (§3.2).

Note that we intentionally describe participants using RoCEv2 roles (RP/NP/CP) rather than generic terms such as *sender*, *receiver*, and *switch* to avoid ambiguous. For example, when testing DUT as RP (*sender* in RoCE2 network), the DUT becomes the *receiver* of the packets generated by Anytest. Using RP/NP/CP makes the role in the network explicit and keeps scenario descriptions unambiguous.

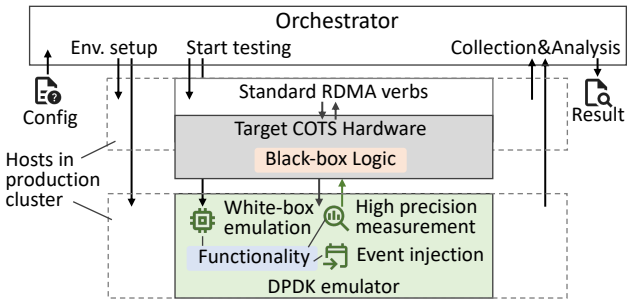


Figure 3: Overview of a single Anytest test.

Architecture and workflow. Figure 3 illustrates the high-level procedure of one Anytest testing. Each testing consists of three components: (1) an *orchestrator* that configures experiments, deploys binaries, coordinates execution, and aggregates results; (2) *DPDK emulator(s)* running on a small set of hosts within the production cluster; and (3) the *target COTS device* (NIC or switch) being tested.

A typical test proceeds as follows. (1) The orchestrator read a configuration that specifies the involved hosts, the DUT, the role to test (RP/NP/CP), and the events that need to be injected (an example configuration is in Appendix A.2). (2) The orchestrator performs environment setup on selected hosts in the affected production cluster (no re-cabling or special hardware). (3) Anytest executes the controlled test case, which will be detailed in § 3.2. (4) The orchestrator collects traces and summary statistics and performs offline analysis to produce a human-readable report.

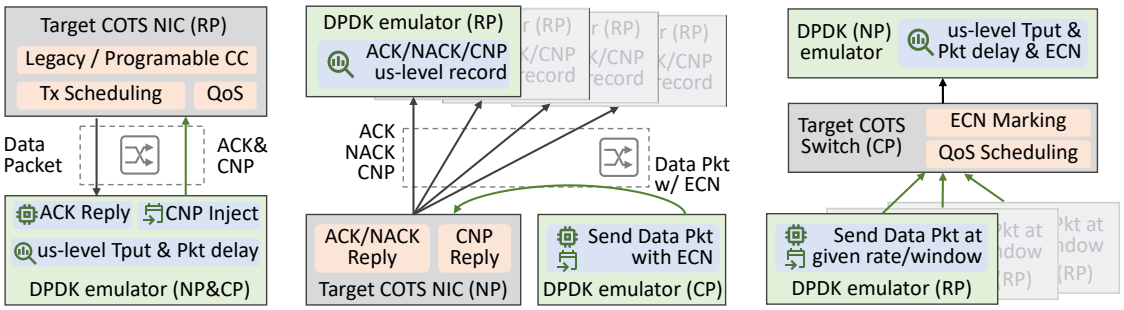
DPDK capability model. To support role-based testing and high-precision localization, Anytest needs three types of capabilities. (i) *White-box emulation*, i.e., protocol-correct packet generation/processing to emulate missing roles in a controlled and reproducible manner. (ii) *Event injection*, i.e., actively injecting transport layer events (e.g., control packets or specific data packet sequences). and (iii) *High-precision measurement*, i.e., measuring μs-level metrics, including throughput, delay, ECN marking ratio, and control-packet sequences, exposing transient transport behaviors invisible to coarse counters. We next show how Anytest composes these capabilities into test scenarios.

3.2 Role-isolated Test Scenarios

Figure 4 summarizes the three test scenarios. Each scenario isolates one role's transport logic under test by holding the other two roles under Anytest control.

(a) RP test: sender NIC as DUT. Anytest tests sender-side transport logic (e.g., congestion control algorithm, scheduling) by running the target NIC as the RP while a DPDK emulator plays both NP and CP. The NIC generates real RoCE traffic via standard RDMA verbs. The DPDK NP logic receives data packets and records data packet arrival timing and rate dynamics; the DPDK CP logic injects controlled CNP patterns to emulate congestion feedback. This scenario could attribute observed rate/behavioral deviations to the sender logic, decoupled from receiver and switch behaviors.

(b) NP test: receiver NIC as DUT. Anytest tests receiver-side transport logic (e.g., ACK/NACK generation and CNP reply) by running the target NIC as the NP while multiple DPDK endpoints



(a) RP test scenario (b) NP test scenario (c) CP test scenario

Figure 4: Three role-based test scenarios supported by Anytest. In each scenario, the DUT acts one role (RP/NP/CP), and DPDK emulates the other two roles to enable controlled stimulus and high-precision observation.

emulate RP and CP. A single DPDK CP crafts data packets with controlled congestion signals (e.g., ECN marks) and sends them to the NIC. To explore how the receiver replies to different senders while keeping the packet sequence deterministic, DPDK CP emulates multiple logical RPs by embedding RPs' IP addresses as the source IP in the generated packets. Given that DPDK bypasses kernel source-IP validation and switches forward packets by destination IP, this requires no routing changes. The DPDK RP records the NIC's outputs (ACK/NACK/CNP sequences) at packet granularity. This scenario isolates how the receiver maps congestion, loss, and reordering conditions to notifications.

(c) CP test: switch as DUT. Anytest tests switch-side congestion signaling and scheduling (e.g., ECN marking behaviors and QoS scheduling) by running the target switch as the CP while DPDK endpoints as RP and NP. The DPDK RPs inject traffic at a controlled rate/window to the switch; the DPDK NP records packet delays and the fraction/timing of ECN markings. Because both traffic stimulus and measurement endpoints are software-controlled, this scenario isolates switch behaviors from NIC-side transport logic.

Coverage. Together, the three scenarios provide a practical localization path for most transport-layer NPAs in production. The coverage limitation is detailed in § 7.

Deployability. Anytest is designed to run *in situ* on a small number of servers within the affected production cluster without re-cabling, packet taps, or specialized tester hardware. Its only deployment requirement is to suspend the service traffic of hosts involved in the test to reduce the noise. For example, background packets sharing the same switch port or queue may introduce additional queuing delays, affecting measurement or disrupting the timing of event injection. Note that it does not require halting all traffic in the cluster; its interaction with normal production traffic is discussed in § 7.

4 CATCHING THE LINE RATE

To realize the observable and controllable test on COTS hardware, Anytest must emulate protocol-correct packets, precisely control packet timing, and record packet-level responses at μs granularity, all while operating at the hardware line rate. Implementing these capabilities in a DPDK-based software dataplane is non-trivial. This section summarizes the key challenges we address: (1) accelerating RoCE packet construction and full-packet ICRC computation to keep up with line rate (§ 4.1), (2) enabling high-precision inter-packet interval control over arbitrary gaps (§ 4.2), and (3) performing accurate measurements with minimal interference to the line-rate datapath (§ 4.3). The engineering details of implementing Anytest are placed in Appendix A.2.

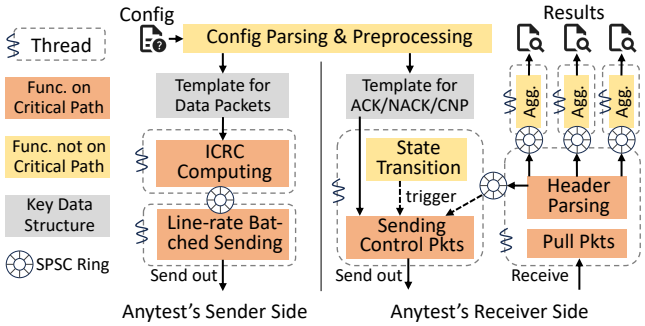


Figure 5: Anytest architecture and thread model.

Anytest is implemented on top of DPDK using ~20,000 lines of code [3]. Figure 5 summarizes the architecture and the thread model. Each DPDK thread is pinned to an isolated CPU core. All threads communicate via lock-free single-producer/single-consumer (SPSC) rings. This section presents experimental results obtained on hosts running Alibaba Cloud Linux 3 (kernel 5.10) and DPDK 24.11, equipped with Intel Xeon Platinum 8469C CPU (48 cores), 1,TB DDR5-4800 memory (16×64 GB), and a BlueField-3 DPU (connected via PCIe Gen5 x16).

4.1 Line-rate Packet Transmit

Packet model. Anytest emulates RoCEv2 roles by generating and processing protocol-correct packets on the wire. Figure 6 illustrates the RoCEv2 packet model used by Anytest. Each packet includes Ethernet/IPv4/UDP headers, RoCEv2 BTH, optional extended headers (e.g., RETH for first packets), payload bytes (often zeros for controlled experiments), and the ICRC field. The ICRC is calculated using the standard CRC-32 (poly `0x04C11DB7`) on the IP, UDP, and IB transport headers along with *the whole payload*, while masking all fields that can vary during transit (e.g., ECN and TTL) to all 1s.

Line-rate Transmit Challenges A straightforward "construct-and-send" implementation cannot reach line rate because of three tightly coupled bottlenecks. (i) *Memory* overhead from per-packet

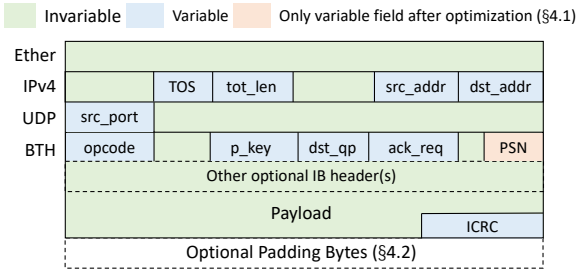


Figure 6: On-wire packet model used by Anytest.

buffer allocation and zero-filling for payload, (ii) *Compute* overhead from per-packet ICRC computation over MTU-sized inputs, and (iii) *Limited parallelism*: Experiments require strict sequencing (continuous PSNs or specific packet sequences), which makes it non-trivial to parallelize packet construction across cores without introducing packet gaps or reordering. We address these bottlenecks with three corresponding techniques below.

Solving the memory bottleneck: packet template pool. To eliminate per-packet allocation and bulk memory writes, Anytest pre-allocates and pre-initializes a packet template pool according to the packet model in Figure 6. Each template contains all invariant bytes and reserves fixed offsets for the few runtime-updated fields. At send time, the transmit path performs only constant-size writes and reuses buffers via DPDK's mbuf clone mechanisms, avoiding per-packet `memset` on the critical path.

Solving the compute bottleneck: PSN-only ICRC fast path. Even with the template pool, recomputing ICRC per packet is prohibitive. A straightforward implementation recomputes the full RoCEv2 ICRC per packet over the entire covered region, which runs in $O(pkt\_len)$ time. However, as shown in Figure 7, using zlib's CRC computation [11] on a single core processes about 10^6 packets per second (pps), which further decreases as the MTU increases. This is far below the pps required to sustain line rate.

We also attempted a `crc_combine` method that leverages the compatibility of CRC[1]. We precompute CRC(*payload*) when materializing packet templates. At runtime, we compute CRC(*header*) after filling dynamic header fields, and then combine the two. However, as shown in Figure 7, it is even slower in practice. This is because the `crc_combine` operation cannot benefit from the CRC accelerator in the CPU, so its overhead outweighs the saved payload CRC work.

To further reduce ICRC computation overhead, we propose $O(1)$ *ICRC runtime computation*. This technique leverages the fact that Anytest only needs to emulate a small set of WQEs with different lengths without actual payloads and send WQEs repeatedly. Therefore, we can pre-compute one template for each packet in WQEs used in the experiment. And at runtime, the only mutable field is `BTH.PSN`, which is located in a 4-byte word. Using CRC composability, the contribution of a 4-byte word at a known position could be precomputed and depends only on the fixed suffix length after this word. To make the correct ICRC, we just need to precompute a *baseline* ICRC for the template with the mutable 4-byte word cleared and update the ICRC with the contribution of the actual word, which requires only $O(1)$ operations. The technical details of this optimization is placed in Appendix A.1. With this optimization, as shown in Figure 7, the pps of single-core runtime ICRC calculation could sustain 200 Gbps when MTU is 1024 bytes and sustain 800 Gbps when MTU is 4096 bytes.

[1]CRC is composable: given CRC(A), CRC(B), and $|B|$ (the length of B), CRC($A \parallel B$) can be derived via a `crc_combine` operation that runs in $O(\log |B|)$ time.

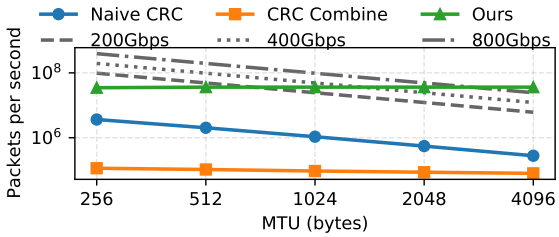


Figure 7: Single core PPS of different ICRC computation methods.

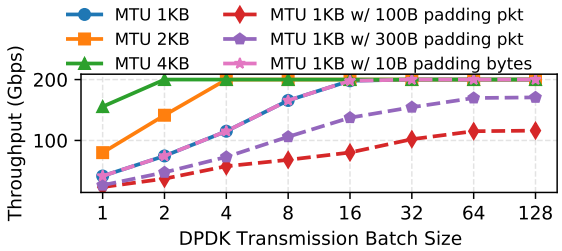


Figure 8: Sender-side DPDK throughput with different transmission batch.

Solving limited parallelism: two-thread transmission pipeline. RoCEv2 tests require strict packet ordering for continuous PSNs or specific packet sequences. Anytest meets this requirement via a two-thread pipeline: a *prepare* thread updates runtime fields and applies the optimized ICRC computation, while a dedicated *send* thread performs batched transmission calls to saturate the bandwidth. An SPSC ring preserves ordering deterministically between the two threads.

As shown in Figure 8, with 1024 B MTU, Anytest easily saturates 200 Gbps, which is the highest single-port line rate currently deployed in our production network. As the transmit datapath of Anytest is largely packet-length agnostic: ICRC is computed in $O(1)$ time per packet, and packet sending and receiving are implemented with zero-copy I/O, the dominant overhead scales with pps rather than bytes. Therefore, we are optimistic that Anytest can extend to 400 Gbps or even 800 Gbps with 4096 B MTU.

4.2 Control μs-level Packet Timing

For the NP and CP tests, Anytest needs to send packets at the given μs-level interval as the stimuli of the hardware, which is challenging for DPDK's throughput-oriented, batch-based transmission model used to sustain line rate.

Padding-based inter-packet interval control. The naïve software approach for injecting packet intervals (busy-wait between packets) is both inaccurate and ineffective: it forces batched transmission calls to degenerate into per-packet calls, which make the sending thread's CPU a bottleneck. In our setup, breaking the

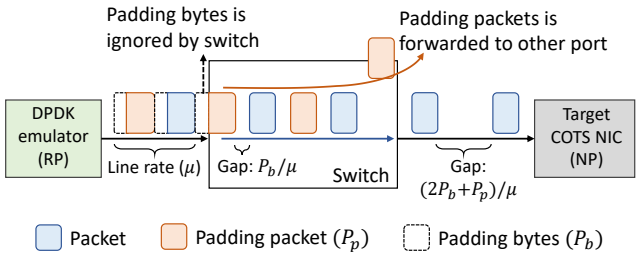


Figure 9: Padding-based packet gap control.

batching could collapse throughput from line rate to just 45 Gbps (batch size 1 in Figure 8).

Anytest achieves μs-level interval control without sacrificing DPDK's batching by using two complementary padding mechanisms (Figure 9) that exploit the NIC's constant-rate hardware serialization to deterministically extend per-packet transmission time and thus shape inter-packet gaps.

(1) Padding packets (coarse-grained control). To create controllable intervals, Anytest inserts dummy packets destined to non-DUT hosts in the batched sending packet train. This creates controlled "holes" in the DUT-observed sequence without breaking the batched transmission, keeping line-rate sending. In this case, the sender's line rate becomes the "pacer" of the packets. The interval of packets observed by DUT is determined by the padding packets' length divided by the line rate. However, this approach cannot provide fine-grained interval control: when padding packets are too small, the throughput bottleneck shifts to PCIe. As shown by Figure 8, with an MTU of 1024 B and a 100 B or 300 B padding packet inserted between every two data packets, the sender cannot sustain line rate even with very large transmission batches. In this regime, the inter-packet interval becomes uncontrollable.

(2) Padding bytes (fine-grained control). To solve the above fine-grained interval control problem, Anytest appends a configurable number of zero bytes *after* the packet (Figure 6). Since these tail bytes are excluded from the ICRC and never interpreted by hardware parsers, they only add transmit-time cost on the DPDK sender's wire serialization, while the DUT RNIC effectively ignore them during parsing. (The padding sits before the Ethernet FCS. So switches forward the frame at its full length (packet + padding) as a normal frame with no extra pipeline overhead.) As shown in Figure 8, padding bytes enable fine-grained interval control while still sustaining line-rate throughput. Note that the padding length is limited: it must not exceed the maximum `mbuf` length in DPDK, otherwise throughput will also downgrade.

As shown in Figure 9, the two complementary padding mechanisms combine to achieve μs-level packet-timing control with a wide tuning range.

4.3 Minimal-interference High Precision Measurement

Hardware timestamps and logging pipeline All packet-level measurements in Anytest are grounded on receive-side hardware timestamps, which provide ns-level timing and are widely supported across firmware versions in our deployment. Upon receiving a packet, the RX thread extracts the timestamp and the minimal metadata needed by each metric (e.g., PSN, ECN bit, packet type) and enqueues them into per-metric SPSC rings. Dedicated logger threads consume these rings, aggregate events into μs-granularity time series, and stream results into in-memory files to avoid disk I/O interference. This design keeps measurement overheads off the critical RX/control path.

This pipeline itself supports the most measurement metrics required by Anytest: μs-level throughput, ECN marking ratio and timing, control-packet sequences and timing, and packet reordering via PSN gaps. For latency-critical tests, receiver-side timestamps alone are insufficient because the injected event (e.g., WQE post or gap insertion) occurs in the sender's software clock domain. The latency measurement method is detailed in Appendix A.2

5 REAL-WORLD DEPLOYMENT

Anytest is a black-box testing tool that *enables* transport-layer localization in production RoCEv2 networks. In practice, we distill an end-to-end workflow using Anytest to accelerate localization. At its core is a *regression test suite*, which could rapidly narrow down the fault domain and often directly pinpoint the root cause by systematically testing each candidate transport role (RP, NP, or CP) in isolation (§ 3.2).

End-to-end workflow. The localization follows the path *symptom → track → role-isolated regression test → trigger*. Using the regression test suite, the localization is performed in three stages. (1) Operators first assign an NPA to one or more localization tracks based on observable symptoms. (2) For each activated track, the operator runs a small set of role-isolated regression tests. These regression tests typically narrow down the failing role and function class and frequently pinpoint the root cause directly. (3) After the regression suite implicates a role and function class, the operator starts a targeted sweep over a small set of knobs to obtain a *sufficient* trigger that reliably reproduces the deviation.

This workflow is largely half-automated. When successful, the end-to-end workflow typically completes within about one hour. If the suite cannot pinpoint the failing role and function class in step (2), on-call engineers are engaged for joint localization, which may involve developing new Anytest test cases and iterating on the test suite. A complete end-to-end example is walked through in § 6.1.

Test track selection. The regression test suite is organized by the symptoms. The *congestion-control* track targets persistent unfairness, rate collapse/oscillation under congestion signals, or abnormal ECN/CNP dynamics. The *loss-recovery* track targets packet drops or elevated retransmissions/timeouts. The *scheduling* track targets multi-priority traffic, priority inversion, or strong sensitivity to host pairing.

Regression tests. The regression test suite comprises a set of representative tests. In the congestion-control track, we use several specific CNP patterns to test the sender RNIC's (RP's) CC performance, ECN-marked data packet sequences to test the receiver RNIC's (NP's) CNP generation, and window-based packet injection to induce target switch queue length and examine the switch's (CP's) ECN marking behavior. In the loss-recovery track, we test the sender RNIC's (RP's) retransmission responses to typical ACK/-NACK sequences and the receiver RNIC's (NP's) NACK generation under out-of-order packet sequences. In the scheduling track, we

Case	Symptom	Type	Role	Root cause	Effort	Why existing tools fail
#1.1	*Congestion Collapse:* After the *cliff* point, performance *collapses*, goodput drops to zero, and latency extremely increases. The network cannot recover to its normal state by itself.	CC	NP	Firmware bug causes CNP reply mode to deviate from configuration	Solved by regression test	Cannot inject line-rate ECN-marked packets
#1.2		CC	RP	A hidden token bucket hinders rate reduction, keeping CC aggressive	2 person-days	Cannot inject specific CNP patterns; no μs-level throughput measurement
#1.3		LR CC	NP	NP stops generating CNPs after out-of-order delivery	1 person-day	Cannot inject specific packet sequences
#2	Unfairness of incast initiated by a host with multiple NICs	CC	NP	Firmware bug indexes `PER_QP` CNP timer by source QP number	5 person-days	Cannot inject line-rate ECN-marked packets; cannot isolate components' effects
#3	Large performance gap under identical incast patterns	CC	RP	The NIC's DCQCN implementation resets α after reaching line rate	2 person-days	Cannot inject specific CNP patterns; no μs-level throughput measurement
#4	Performance downgrades in specific firmware	CC	RP NP	Firmware bug makes CC performance inconsistent across versions	Solved by regression test	Cannot inject specific CNP patterns; no μs-level throughput measurement
#5	Small-message latency spikes with background traffic	Sche.	RP	Unexpected sub-optimal transmission scheduling behavior	Solved by regression test	No μs-level per-packet latency measurement
#6	Bandwidth scheduling imbalance in VOQ-based switch	Sche.	CP	Imbalanced slice scheduling in VOQ-based switch	Solved by regression test	Cannot inject line-rate packets w/o CC; cannot isolate cross-component effects

Table 2: Representative root cause localization cases solved by Anytest and limitations of existing tools.

run workloads spanning multiple traffic classes and measure the resulting scheduling throughput and latency on the sender RNIC (RP) or the switch (CP).

Expected result of regression tests We check results using two types of explicit baselines. A *theory baseline* that is enforced by protocol- and configuration-implied rules (e.g., correspondence between configuration and response, ACK/NACK/CNP triggering rules). A *comparison baseline* contrasts the DUT against a reference device/configuration that does not reproduce the anomaly under the same regression test, which is particularly useful when theory does not prescribe expected results.

Empirical Data. Anytest has been deployed on-demand across Alibaba Cloud RoCE clusters for 12 months, covering 4 NIC models and 6 switch models. During this period, Anytest successfully localized 58 NPAs in total. Among them, 39 (67.2%) were resolved directly by the regression test suite with zero manual effort. For the remaining 19 NPAs that required bespoke investigation, the mean localization cost was 9.47 person-hours, with the most complex case (Case #2 in § 6) requiring 5 person-days. Averaged over all 58 cases (counting zero-effort ones as 0), the overall mean is 3.1 person-hours. Beyond efficiency, Anytest enables previously infeasible diagnoses. For example, case #2's prior workaround consumed ~20 person-days before Anytest pinpointed the root cause.

6 CASE STUDY AND LESSON LEARNED

In this section we present and discuss several representative root cause localization cases solved by Anytest, summarized in Table 2.

6.1 Root Cause Localization Walkthrough

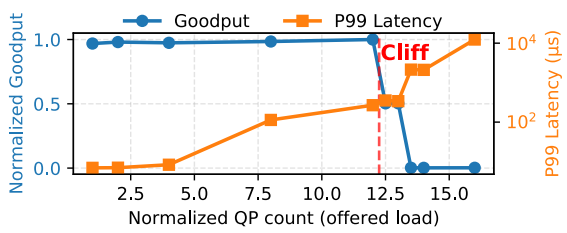
We use Case #1 to illustrate the whole workflow of a root-cause localization. Case #1 is a real production incident where the network exhibits a *congestion collapse*—a cliff-like goodput drop once the offered load exceeded the network's sustainable capacity. Congestion collapse is historically associated with early Internet deployments [1, 15] and is widely believed to be mitigated by modern CC [4, 19]. Surprisingly, we observed it in a production RoCEv2 environment with COTS RNICs and Ethernet switches. The deployment used our in-house CC based on Mellanox's PCC mechanism [25], with ECN/CNP as the congestion signals. This case highlights how black-box transport logic and subtle cross-component interactions can introduce severe performance anomalies.

Anomaly detection. The incident occurred during a scale-across expansion that enlarged the communication domain and increased the probability of transient many-to-one bursts. In an extreme case (up to 400 senders incasting to one receiver), the online monitor system observed uncontrollable packet loss and a rapid goodput drop to near zero.

Minimal reproduction test case. While the large-scale workload triggered the incident, engineers quickly distilled it into a minimal, network-only reproduction test case using *RDMA-tracer and Synapse* (§ 2). The collapse could be reproduced using two senders that simultaneously incast to one receiver, increasing the number of QPs per sender to raise the offered load. The result of this reproduction is shown in Figure 10. As the offered load grows, the system exhibits a clear phase transition. (i) In the *pre-collapse* regime, aggregate goodput remains at line rate while 99.9-percentile latency and ECN/CNP volume increase, without loss (indicated by `out_of_order` counter). (ii) After the *cliff* point, it *collapses*, goodput drops, latency and ECN extremely increase, and loss becomes pervasive. Once the network transits to the collapse state, it can not self-recover over time.

Track selection and regression tests. Following §5, we first assign this case to the *congestion control* track and run the set of regression tests. The sender-side CC regression (RP test) and the switch-side marking/scheduling regression (CP test) both *pass*: under the same controlled stimuli, the DUT behaviors match those of the reference NIC/switch. In contrast, the receiver-side CNP regression (NP test) *fails*, which is detailed below.

Root cause #1.1: "`PER_QP`" configuration but "`PER_PORT`" CNP reply. On currently shipping Mellanox RNICs, the CC algorithm can be configured in either a coarse-grained or a fine-grained mode



(a) Network performance.

(b) Congestion signal and loss count.

Figure 10: Congestion collapse observed in our RoCEv2 deployment.

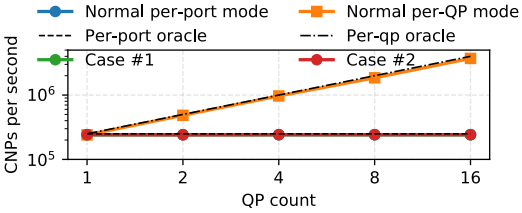


Figure 11: CNPs per second under different configurations and cases.

In-order transmission: CNP is generated for every packet.	PSN	0	1	2	3	4	5
	CNP	●	●	●	●	●	●
After an PSN gap: out-of-order packets donot trigger CNP.	PSN	0	2	3	4	5	6
	CNP	●	●	○	○	○	○
After retransmission completes: CNP generation resumes.	PSN	0	2	3	4	1	2
	CNP	●	●	○	○	●	●

Table 3: Receiver CNP behavior under various packet order. All the packets are ECN-marked. ● means a corresponding CNP generated.

[26]. In coarse-grained mode, the sender maintains one CC state per destination IP (`PER_IP`), while the receiver uses a single CNP interval timer shared by all senders (per-port, i.e., one timer per receiving port). In fine-grained mode, the sender maintains one CC state per QP (`PER_QP`), and the receiver maintains one CNP timer per QP (`PER_QP`). In our experience, coarse-grained CC can cause congested QPs to throttle unrelated non-congested QPs. Thus, most of our deployments have migrated (or are migrating) to `PER_QP` CC mode.

Under this setting, we configured both the sender and receiver to `PER_QP` mode but observed that the receiver still generates CNPs in a per-port manner. Figure 11 shows the results using the NP test scenario: we set the receiver CNP timer to 4 µs and transmit ECN-marked packets at 200 Gbps line rate (~23 Mpps). However, regardless of increasing the number of sending hosts or the number of QPs, the receiver emits only ~0.25 Mpps CNPs, which matches the maximum rate allowed by a single 4 µs timer and is far below that is expected under a correct `PER_QP` mode. We reported this issue to the vendor and upgraded the driver to a widely validated release, which resolved the problem.

Root cause #1.2: hidden token bucket hinders PCC rate throttling. After fixing the CNP response, collapse still occurred at larger QP counts. In the meantime, engineers working on CC noted that, under the observed CNP intensity, PCC should be able to bound the sending rate even in the theoretical worst case. This suggests there is an unknown mechanism that hinders the sender (RP) from throttling the rate. We use the RP test scenario to test the CC response of a single QP. For this QP, we inject CNPs at the per-QP CNP rate observed just before the cliff and sweep the injected CNP rate around that point. Surprisingly, when the injected CNP rate is at around ~100 CNPs/s, we observe no measurable rate reduction at the sender. After detailed experiments and discussions with RNIC architecture experts, we identify an unconfigurable token bucket between the PCC rate-control module and the packet transmission module. When the CC reduction is small and the fast recovery is aggressive, the token bucket never drains enough to throttle transmission, and thus no slowdown is observed. To mitigate this, we applied conservative parameter tuning to reduce CC aggressiveness.

Root cause #1.3: packet loss suppresses CNP generation. However, resolving the two issues above only shifts the offered-load threshold for congestion collapse. Once collapsed, the network cannot self-recover over time. We hypothesize that the anomalous reduction of CNP count after collapse contributes to this lack of self-recovery, and thus seek the root cause of the reduction. Notably, the CNP drop coincides with packet loss. We therefore manually construct an NP-side test that injects ECN-marked packets together with controlled PSN gaps to emulate packet loss. As shown in Table 3, after a PSN gap occurs, the RNIC will stop generating CNPs even when subsequent packets are ECN-marked. This behavior is confirmed by the vendor and explains the collapse's positive feedback loop.

Putting the pieces together: why the collapse occurs. The collapse is not caused by any single bug, but by a compounding control-loop failure. With root cause #1.1, the intensity of per-QP CNPs weakens as QP increases. Root cause #1.2 adds an actuation lag: small CC backoffs do not translate into immediate rate reduction due to the hidden token bucket. These two effects increase the chance of packet loss. Once that happens, root cause #1.3 suppresses CNP generation despite continued ECN marking, effectively reducing the congestion signal and creating positive feedback: congestion → loss → less CNPs → more congestion.

Mitigation and longer-term fix. As an immediate mitigation, we upgraded firmware to fix Case #1.1 and applied conservative CC parameter tuning to mitigate Case #1.2. These two fixes reduce the offered-load threshold for triggering collapse to below the worst-case load that could occur in our production environment. Long term, we are extending our in-house PCC-based CC to incorporate

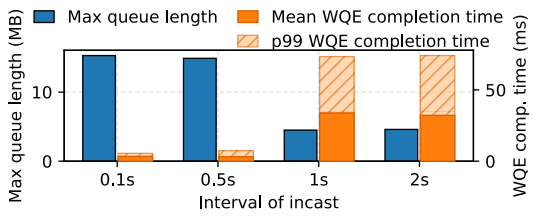


Figure 12: Performance gap of identical incast traffic with different interval.

NACK and RTT as signals so that the control loop remains robust even if CNP feedback is impaired.

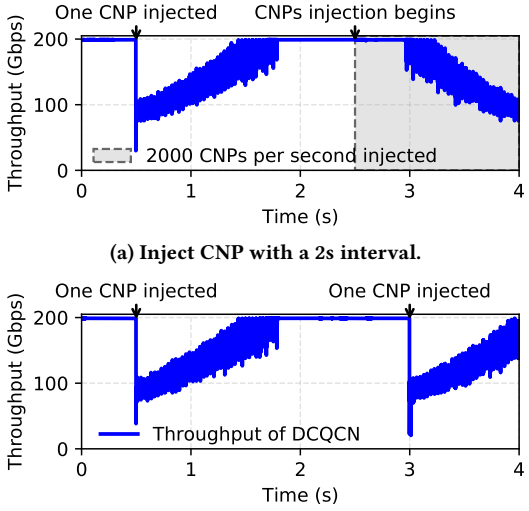
6.2 Cases localized by Anytest

Case #2: Unfairness caused by problematic CNP timer aggregation. In multi-NIC hosts (e.g., training or inference servers with 4/8 RNICs per node), incast unfairness frequently occurs under the PER_QP configuration. A minimal setup is multiple RNICs on one host incasting to a single RNIC on another host. However, this setup does not reliably reproduce the unfairness. We first ran the full CC-track regression test and found no issues. After several days of debugging, we noticed a consistent pattern: whenever the anomaly occurred, the higher-rate QPs shared the same *source* QP number. In a production RDMA network, arbitrarily modifying the source QP number is generally infeasible because it is managed by the RDMA stack and NIC driver. In our NP test scenario, however, the sender is DPDK-based and can bypass the driver, allowing us to freely populate QP fields and thus customize the source QP number in Anytest.

We then developed a feature in Anytest to customize the source QP number in the NP test scenario. Note that arbitrarily modifying the source QP number is generally infeasible as it is allocated by NIC driver. However, in NP test scenario, the sender is a DPDK-based emulator, allowing direct population of QP fields. Therefore Anytest enables a capability that conventional approaches cannot provide and quickly confirm the root cause. Under PER_QP CNP generation, the receiver aggregates QPs with the same *source* QP number—rather than the same destination QP—to share a CNP timer. As a result, these QPs receive fewer CNPs and maintain higher sending rates, leading to unfairness. The vendor has confirmed this unexpected behavior.

This issue is more likely on multi-NIC hosts because different RNICs are assigned the same initial QP number after the host's boot, making it likely that concurrent QPs are using identical QP numbers. We modify software to avoid QP-number collisions as a short-term mitigation and wait for the vendor fix as a long-term solution.

Case #3: Poor network performance caused by interval-sensitive DCQCN behavior. In a production application with a recurring incast traffic pattern, we observed that the DCQCN exhibited bimodal performance: one is overly aggressive, while another is overly conservative. Using our tracing tools, we found that the bimodal performance of a recurring incast workload is tightly correlated with the *incast interval*. Figure 12 shows a 2-to-1 incast with 16 QPs per sender, where each QP repeatedly sends 16 WQEs at 1 MB burstily under different inter-burst intervals. When the interval is small, DCQCN behaves aggressively, leading to high switch queue length. When the interval is large, DCQCN becomes overly conservative and significantly inflates WQE completion times. We then used Anytest to sweep the interval under the *same* traffic pattern and found that the RP tests reproduce the bimodal performance, which localizes the anomaly to DCQCN implementation.



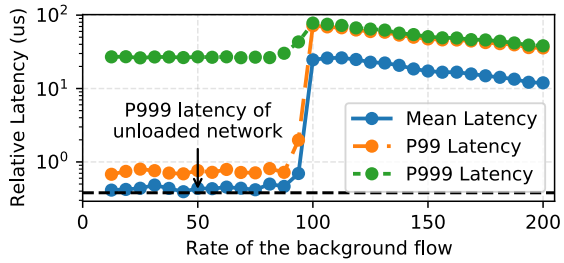
(a) Inject CNP with a 2s interval.

(b) Inject CNP with a 2.5s interval.

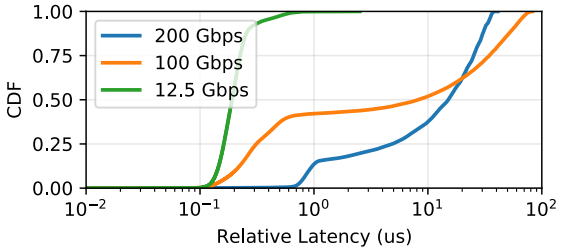
Figure 13: Detailed analysis of DCQCN under different CNP interval.

To further highlight its sensitivity, Figure 12 shows a single-QP RP test with controlled CNP injection. Upon the first CNP, DCQCN halves the rate and then gradually recovers, consistent with the algorithm (i.e., the flow starts with α=1 and decelerates by $\alpha/2$ once it receives CNP [33]). After 2 seconds, injecting multiple CNPs results in only a slow decrease in the rate, which is expected as α decays over time, leading to a moderate backoff for each CNP (Figure 13a; Listing 1 shows the full configuration). However, when the interval between two CNP is longer (e.g., 2.5 s in Figure 13b), DCQCN again reacts as if α=1, producing a much larger rate reduction. This is counter-intuitive: *later and less frequent* CNPs trigger *stronger* throttling. Discussions with the vendor reveal the cause: the DCQCN implementation in RNIC resets α to 1 after the rate recovers to the line rate. As a mitigation, we deploy our PCC-based CC whenever possible to ensure stable performance.

Case #4: Firmware-dependent CC behavior divergence. Beyond the CC anomalies above, we repeatedly observed that CC behavior can diverge across RNIC firmware versions. Such regressions are straightforward to localize with Anytest's CC-track regression tests: Anytest replays identical, controlled stimuli and compares μs-level metrics against a previously validated firmware, enabling direct attribution of the anomaly to the firmware change. In contrast, without Anytest, this comparison is often impractical in production. Clusters are typically deployed from a set of configuration combinations. As a result, firmware differences often co-occur with changes in other components (e.g., host and network hardware models, kernel and driver versions), preventing confident attribution to any single component.



(a) Latency delta w/ various background rate.



(b) CDFs of latency delta.

Figure 14: Latency anomaly observed under ETS mechanism.

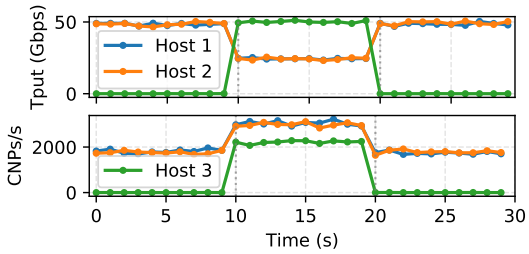


Figure 15: Telemetry counters with unbalanced scheduling switch.

Case #5: Sharp latency inflation under ETS scheduling. RNIC's ETS mechanism [24] is used to perform scheduling across multiple traffic classes (TC). In principle, we place latency-sensitive, low-rate traffic in a dedicated TC and allocate it a weight exceeding its required bandwidth to protect its latency. However, we find that when another ETS TC carries high-throughput traffic, the latency-sensitive traffic degrades even when the bandwidth is not saturated. We therefore use Anytest to stress-test ETS under the RP scenario on a 200 Gbps RNIC. We configure two ETS TCs with a weight ratio of 1:2. A QP for latency-sensitive traffic is mapped to the TC with weight 1 and transmits 4096 B WQEs every 5 µs (about 6.5 Gbps, far below its allocated share), while the background traffic QP continuously sends large WQEs at another TC. Anytest regulates the background QP rate by injecting CNPs to it. Figure 14a plots the latency-sensitive flow's delay relative to the unloaded-network baseline under different background traffic rates. When the background rate is below 90 Gbps, the latency inflation is modest and stable. As the background rate increases further, we observe an sharp latency jump: the mean inflation rises to ~30 µs, and p99/p99.9 increase to ~70 µs. However, with further increases in background rate, the latency inflation decreases. Figure 14b further illustrates this behavior: there is a heavier tail at 100 Gbps than at 200 Gbps despite smaller low percentiles. This pattern suggests non-trivial internal ETS scheduling behavior in the RNIC that is opaque without Anytest.

Case #6: Unfair port bandwidth scheduling in a VOQ-based switch. We observed severe throughput unfairness on a VOQ-based switch (Figure 15). When Host1 and Host2 simultaneously send to Host4, the two flows share the 100 Gbps bottleneck fairly. However, after an additional Host3→Host4 flow starts, the throughput allocation shifts to roughly 25-25-50 Gbps. The lower subplot shows that Host3 receives fewer CNPs than Host1 and Host2, leaving the root cause of Host3 getting more bandwidth ambiguous (ECN marking, NP-side CNP generation, or switch scheduling). Anytest resolves the ambiguity using the regression test: we drive identical offered rates without CC from Host1/2/3 to Host4 and still observe a 25/25/50 Gbps split. This rules out ECN/CNP effects and attributes the anomaly to switch scheduling. Follow-up investigation reveals that this VOQ-based switch schedules bandwidth at the granularity of *slices* [8]. And an unfair slice partitioning caused the bias. The issue was fixed in a subsequent switch configuration update.

6.3 Lessons learned

Root cause localization is challenging without isolating roles and observing µs-scale dynamics. As shown by cases above, similar symptoms (loss/unfairness/latency spikes) can originate from any participant or cross-party coupling. Therefore, effective localization must isolate one role at a time (RP/NP/CP) and measure packet-level responses at µs granularity under controlled stimulus.

Regression test suite turns much of localization into automated workflows. A large fraction of incidents (e.g., #1.1 and #4-#6) can be localized directly by the regression test suite, while others (e.g., #1.2 and #3) benefit from the suite's rapid triage that narrows the search to a specific role/function class. Besides, as Case #4 shows, production heterogeneity makes ad-hoc A/B comparisons heavily confounded, and many NPAs recur after firmware or configuration updates. Overall, maintaining such a suite is effective for (i) quickly attributing an anomaly to a responsible role/function and (ii) gating firmware/driver/config rollouts with automated checks.

Configuration does not imply semantics—validate config-to-behavior mappings. Cases #1.1 and #4 show that vendor-exposed knobs are not reliable semantics in offloaded devices. The key takeaway is to treat configuration as an interface contract that must be empirically validated via controlled stimulus and packet-level responses.

Congestion feedback can fail—design and test for degraded feedback. Case #1.3 highlights that the congestion signal itself can degrade when it is most needed, therefore creating positive feedback leading to congestion collapse. The lesson is that a robust CC must tolerate impaired CNP feedback (e.g., incorporating NACK/RTT) and be validated under adverse conditions.

Hidden modules can nullify CC action—measure on-wire execution, not just algorithmic analysis. Case #1.2 reveals that undocumented token buckets in RNIC can mask small backoffs and introduce action lag. The lesson is that debugging and tuning must

be grounded in execution-level evidence rather than in algorithmic expectations alone.

7 DISCUSSION

Scope and observability. Anytest localizes anomalies using externally observable on-wire behaviors under controlled stimulus, focusing on transport-visible RoCEv2 dynamics (ECN/CNP/ACK-/NACK and packet-level timing/ordering). As a black-box testing tool, it cannot directly validate internal firmware/ASIC states or micro-architectural mechanisms beyond what can be inferred from packet-level responses. Instead, it typically narrows issues to the responsible role, function class, and a compact trigger specification, which is often sufficient for mitigation and vendor fixes. This external-observability constraint also limits coverage at lower layers: PFC anomalies operate at the link layer and are not directly observable in our DPDK-based pipeline, so Anytest cannot localize PFC anomalies. Extending Anytest to observe and localize PFC-related anomalies is committed as future work.

In-situ interference between tests and production traffic. Because Anytest runs *in situ*, its tests may perturb nearby production traffic by consuming shared switch resources (e.g., buffers). Conversely, background traffic can inject noise into Anytest experiments by contending for shared resources or sending unrelated packets to the tested hosts. We mitigate these effects by scoping tests to explicitly selected hosts/ports, bounding test intensity and duration, and repeating runs to distinguish systematic deviations from noise.

Support higher line-rate. Our current performance evaluation demonstrates Anytest sustaining today's 200 Gbps production network with 1024 B MTU. Note that our datapath is largely packet-length agnostic: ICRC is computed in $O(1)$ time per packet, and both packet sending and receiving are implemented with zero-copy I/O. Therefore, we are optimistic that Anytest can extend to 400 Gbps or even 800 Gbps with 4096 B MTU.

Human-in-the-loop localization. The regression test suite (§ 5) is fully automated, resolving 67% of incidents with zero manual effort. Human expertise is required only when the existing suite cannot cover the anomaly: engineers must design new stimuli, select sweep dimensions, and interpret results for previously unseen NPAs (e.g., Case #1.2, #2). This limitation is inherent to black-box testing: it cannot guarantee fully automated coverage of all boundary conditions. Once such an NPA is localized with human involvement, the test case that exposes the root cause—including the sweep dimensions, sweep range, and expected result—is codified into the regression suite. This process continuously expands the suite's coverage, progressively automating triage for future incidents and rollout gating.

Comparison with other open-source tools. MoonGen [9] is a scriptable DPDK-based packet generator designed for Ethernet testing; however, it lacks the RoCE-specific optimizations described in § 4 (e.g., $O(1)$ ICRC update, patterned injection FSM), and therefore cannot sustain line-rate RoCE traffic with correct transport semantics. Verbsmarks [10] and similar ibverbs-based microbenchmark suites can generate real RDMA traffic through the NIC hardware and thus reproduce NPAs, but they operate as black-box workload generators with no mechanism to isolate individual transport roles, sweep protocol parameters, or compare DUT behavior against expected baselines. Both tools, together with our *Synapse*, belong to the traffic-generator category—they produce or replay traffic but do not localize root causes. In contrast, Anytest goes beyond traffic generation: it combines protocol-correct packet injection with role-isolated regression testing, enabling systematic NPA localization that no traffic generator alone can achieve.

Why simpler substitutes are insufficient. Simpler alternatives may appear capable of reproducing some NPAs that Anytest localizes, but they cannot cover all scenarios or fully control the test stimulus. For example, constructing incast traffic with fixed congestion windows or disabled congestion control can produce ECN-marked sequences sufficient to reproduce Case #1.1 (§ 6.1); however, when a switch is configured with probabilistic ECN marking, such setups cannot generate a fully ECN-marked sequence on demand, losing the ability to deterministically probe specific CC state transitions. Similarly, low-rate packet generators can construct arbitrary DATA/ACK/NACK sequences to attempt localization of cases like Case #1.3, but their slow response loop may trigger spurious retransmission timeouts in RP tests, confounding the measurement with artifacts absent from real traffic.

Generalizability. Although our current implementation targets DCQCN/RoCEv2, the RP/NP/CP role decomposition maps naturally to the broader pattern of sender-side reaction, receiver-side feedback generation, and network-side signaling, which is shared by other congestion control schemes (e.g., INT-based [2, 21] or delay-based [19, 23, 32] ones). Extending Anytest to these protocols requires updating the packet templates and state-machine definitions in the configuration, while the core infrastructure—line-rate injection, role isolation, and μs-resolution measurement—remains directly reusable.

8 CONCLUSION

We present Anytest, an in-situ tool for localizing transport-layer NPAs in RoCEv2 COTS RNICs and switches. Anytest isolates each transport role (RP/NP/CP) with DPDK-based protocol-correct emulation, enabling controlled event injection and μs-granularity packet-level measurement to attribute anomalies to a responsible participant, function class, and trigger. In production deployments, Anytest localizes diverse real-world incidents and efficiently reduces the average NPA root cause localization time to 3.1 person-hours.

ACKNOWLEDGE

We would like to thank the anonymous shepherd and the anonymous reviewers for all their constructive feedback and comments. This research is supported by the National Natural Science Foundation of China under Grant Numbers 62325205, U25B2035, 62502197, 62441236, and 62372296, the Key Program of Natural Science Foundation of Jiangsu under grant No. BK20243053, the Basic Research Program of Jiangsu under Grant Number BK20251206, the Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China (No. JYB2025XDXM118), and the "111 Center" (No. B26023). Jiaqi Gao and Qingkai Meng are corresponding authors.

REFERENCES

[1] 1984. Congestion Control in IP/TCP Internetworks. RFC 896. (Jan. 1984). https://doi.org/10.17487/RFC0896

[2] Vamsi Addanki, Oliver Michel, and Stefan Schmid. 2022. {PowerTCP}: Pushing the performance limits of datacenter networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 51–70.

[3] Alibaba Group. 2025. AnyTest. https://github.com/AlibabaResearch/Anytest. (2025).

[4] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*. 63–74.

[5] Wei Bai, Shanim Sainul Abdeen, Ankit Agrawal, Krishan Kumar Attre, Paramvir Bahl, Ameya Bhagat, Gowri Bhaskara, Tanya Brokhman, Lei Cao, Ahmad Cheema, et al. 2023. Empowering azure storage with {RDMA}. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 49–67.

[6] Yanqing Chen, Bingchuan Tian, Chen Tian, Li Dai, Yu Zhou, Mengjing Ma, Ming Tang, Hao Zheng, Zhewen Yang, Guihai Chen, et al. 2023. Norma: towards practical network load testing. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 1733–1749.

[7] Yanqing Chen, Li Wang, Jingzhi Wang, Songyue Liu, Keqiang He, Jian Wang, Xiaoliang Wang, Wanchun Dou, Guihai Chen, and Chen Tian. 2025. Marlin: Enabling High-Throughput Congestion Control Testing in Large-Scale Networks. In *Proceedings of the Twentieth European Conference on Computer Systems*. 460–474.

[8] Cisco Systems, Inc. 2025. *Traffic Management with Virtual Output Queues (VOQs)*. Cisco Systems, Inc. https://www.cisco.com/c/en/us/td/docs/iosxr/cisco8000/qos/b-modular-qos-config-cisco8000/m-traffic-management-with-voq.pdf Cisco IOS XR documentation chapter (updated Dec. 17, 2025). Describes VOQ replication across ingress pipelines/slices and credit-based scheduling.

[9] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. 2015. MoonGen: A Scriptable High-Speed Packet Generator. In *Proceedings of the 2015 Internet Measurement Conference*. ACM, 275–287.

[10] Falcon Transport. 2024. verbsmarks. https://github.com/falcon-transport/verbsmarks. (2024).

[11] Jean-loup Gailly and Mark Adler. 2024. zlib Home Site. (2024). https://zlib.net/ Accessed: 2026-01-31.

[12] Adithya Gangidi, Rui Miao, Shengbao Zheng, Sai Jayesh Bondu, Guilherme Goes, Hany Morsy, Rohit Puri, Mohammad Riftadi, Ashmitha Jeevaraj Shetty, Jingyi Yang, et al. 2024. RDMA over Ethernet for Distributed Training at Meta Scale. In *Proceedings of the ACM SIGCOMM 2024 Conference*. 57–70.

[13] Peter X Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Network requirements for resource disaggregation. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 249–264.

[14] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, et al. 2021. When cloud storage meets {rdma}. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 519–533.

[15] Van Jacobson. 1988. Congestion avoidance and control. *ACM SIGCOMM computer communication review* 18, 4 (1988), 314–329.

[16] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, et al. 2024. {MegaScale}: Scaling large language model training to more than 10,000 {GPUs}. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 745–760.

[17] Xinhao Kong, Jingrong Chen, Wei Bai, Yechen Xu, Mahmoud Elhaddad, Shachar Raindel, Jitendra Padhye, Alvin R Lebeck, and Danyang Zhuo. 2023. Understanding {RDMA} microarchitecture resources for performance isolation. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 31–48.

[18] Xinhao Kong, Yibo Zhu, Huaping Zhou, Zhuo Jiang, Jianxi Ye, Chuanxiong Guo, and Danyang Zhuo. 2022. Collie: Finding performance anomalies in {RDMA} subsystems. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 287–305.

[19] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan MG Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, et al. 2020. Swift: Delay is simple and effective for congestion control in the datacenter. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 514–528.

[20] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.

[21] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, et al. 2019. HPCC: High precision congestion control. In *Proceedings of the ACM Special Interest Group on Data Communication*. 44–58.

[22] Rui Miao, Lingjun Zhu, Shu Ma, Kun Qian, Shujun Zhuang, Bo Li, Shuguang Cheng, Jiaqi Gao, Yan Zhuang, Pengcheng Zhang, et al. 2022. From luna to solar: the evolutions of the compute-to-storage networks in Alibaba cloud. In *Proceedings of the ACM SIGCOMM 2022 Conference*. 753–766.

[23] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. 2015. TIMELY: RTT-based congestion control for the datacenter. *ACM SIGCOMM Computer Communication Review* 45, 4 (2015), 537–550.

[24] NVIDIA. 2023. Quality of Service (QoS). MLNX_OFED Documentation (Rev 4.6-1.0.1.1). (Oct. 2023). https://docs.nvidia.com/networking/display/mlnxofedv461000/quality+of+service+(qos) Includes the section "Enhanced Transmission Selection (ETS)" and its configuration via `mlnx_qos`; last updated Oct. 23, 2023.

[25] NVIDIA 2025. *DOCA PCC Application Guide*. NVIDIA. https://docs.nvidia.com/doca/sdk/doca-pcc-application-guide/index.html Last updated Dec. 18, 2025. Accessed: Feb. 1, 2026.

[26] NVIDIA 2025. *NVIDIA ConnectX-8 SuperNIC Firmware Release Notes v40.47.1026.1026*. NVIDIA. https://docs.nvidia.com/networking/display/nvidia-connectx-8-supernic-firmware-release-notes-v40-47-1026.1026.pdf Customer affecting change: the default CNP moderation behavior changed from per-flow to per-port; configurable via mlxconfig parameter ROCE_CC_CNP_MODERATION. Accessed: 2026-02-01.

[27] Kun Qian, Yongqing Xi, Jiamin Cao, Jiaqi Gao, Yichi Xu, Yu Guan, Binzhang Fu, Xuemei Shi, Fangbo Zhu, Rui Miao, et al. 2024. Alibaba hpn: A data center network for large language model training. In *Proceedings of the ACM SIGCOMM 2024 Conference*. 691–706.

[28] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. 2025. Mooncake: Trading more storage for less computation—a {KVCache-centric} architecture for serving {LLM} chatbot. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*. 155–170.

[29] Zhenghang Ren, Yuxuan Li, Zilong Wang, Xinyang Huang, Wenxue Li, Kaiqiang Xu, Xudong Liao, Yijun Sun, Bowen Liu, Han Tian, Junxue Zhang, Mingfei Wang, Zhizhen Zhong, Guyue Liu, Ying Zhang, and Kai Chen. 2025. Enabling Efficient GPU Communication over Multiple NICs with FuseLink. In *19th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2025, Boston, MA, USA, July 7-9, 2025*, Lidong Zhou and Yuanyuan Zhou (Eds.). USENIX Association, 91–108. https://www.usenix.org/conference/osdi25/presentation/ren

[30] Leah Shalev, Hani Ayoub, Nafea Bshara, and Erez Sabbag. 2020. A cloud-optimized transport protocol for elastic and scalable hpc. *IEEE micro* 40, 6 (2020), 67–73.

[31] Zhuolong Yu, Bowen Su, Wei Bai, Shachar Raindel, Vladimir Braverman, and Xin Jin. 2023. Understanding the micro-behaviors of hardware offloaded network stacks with lumina. In *Proceedings of the ACM SIGCOMM 2023 Conference*. 1074–1087.

[32] Zhaochen Zhang, Feiyang Xue, Rui Ning, Keqiang He, Gianni Antichi, Jiaqi Gao, Zhimeng Yin, Kexin Liu, Rui Li, Zhengqi Cui, Zhehao Lin, Peirui Cao, Guihai Chen, and Chen Tian. 2026. OSCAR: O(1)-Step Convergence and Readily-deployable Congestion Control. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*. USENIX Association, Renton, WA, 543–570. https://www.usenix.org/conference/nsdi26/presentation/zhang-zhaochen

[33] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. 2015. Congestion control for large-scale RDMA deployments. *ACM SIGCOMM Computer Communication Review* 45, 4 (2015), 523–536.

Appendices

Appendices are supporting material that has not been peer-reviewed.

A IMPLEMENTATION

A.1 Mathematics in ICRC Optimization

Mathematical principle of the PSN-only ICRC fast path. Let $\mathrm{CRC}(\cdot)$ denote the 32-bit CRC operator used to form the RoCEv2 ICRC over the covered byte sequence (after applying the RoCEv2-specified header masking), and let $\oplus$ denote bitwise XOR. The covered region ends at the 4-byte ICRC field; any tail padding beyond that field is excluded from the CRC input and therefore must not be counted in any length term below.

CRC composability. CRC is composable over concatenation: for any byte strings A and B,

$$\mathrm{CRC}(A \,\|\, B) = \mathrm{Shift}(|B|)\big(\mathrm{CRC}(A)\big) \;\oplus\; \mathrm{CRC}(B), \tag{1}$$

where $\|$ denotes concatenation, $|B|$ is the length of B in bytes, and $\mathrm{Shift}(|B|)$ is a fixed 32×32 linear transform over GF(2) that advances a CRC state by $|B|$ bytes. This is the algebraic meaning behind `crc_combine`-style operations.

Decomposing the ICRC input. In our setting, at runtime the only mutable header field is `BTH.PSN`, which resides in a known 4-byte word together with the `ack_req` bit. Let W denote this 4-byte word (in network byte order). For each packet template, the covered byte sequence M can be written as

$$M = P \,\|\, W \,\|\, S, \tag{2}$$

where P is a fixed prefix, S is a fixed suffix, and only W changes across packets. Applying (1) twice yields

$$\begin{aligned}\mathrm{CRC}(P \,\|\, W \,\|\, S) =& \mathrm{Shift}(4 + |S|)\big(\mathrm{CRC}(P)\big) \\ &\oplus \mathrm{Shift}(|S|)\big(\mathrm{CRC}(W)\big) \\ &\oplus \mathrm{CRC}(S),\end{aligned} \tag{3}$$

where $|S|$ is the suffix length in bytes (within the covered region).

Baseline-plus-contribution form. Let $Z = 0^4$ be a 4-byte all-zero word. We define the *baseline* CRC of a template as the CRC of the packet with the mutable word cleared:

$$\mathrm{CRC}_{base} \;\triangleq\; \mathrm{CRC}(P \,\|\, Z \,\|\, S). \tag{4}$$

Then, for any runtime value of W,

$$\mathrm{CRC}(P \,\|\, W \,\|\, S) = \mathrm{CRC}_{base} \;\oplus\; \Delta(W), \tag{5}$$

where the *contribution* of the actual 4-byte word is

$$\Delta(W) \;\triangleq\; \mathrm{Shift}(|S|)\big(\mathrm{CRC}(W)\big) \;\oplus\; \mathrm{Shift}(|S|)\big(\mathrm{CRC}(Z)\big). \tag{6}$$

Crucially, $\mathrm{CRC}(Z)$ must not be omitted: even an all-zero 4-byte word advances the CRC state machine by 4 bytes, so its effect is captured by the $\mathrm{Shift}(|S|)\big(\mathrm{CRC}(Z)\big)$ term in (6). With this definition, $\Delta(Z) = 0$ holds by construction.

$O(1)$ update via per-bit precomputation. Since CRC and $\mathrm{Shift}(\cdot)$ are linear over GF(2), $\Delta(\cdot)$ is linear in the 32 bits of the word W. Let e_b denote the 32-bit basis word whose b-th bit is 1 and all other bits are 0. Precomputing

$$C_b \;\triangleq\; \Delta(e_b), \qquad b \in \{0, \ldots, 31\}, \tag{7}$$

we can compute the runtime CRC as

$$\mathrm{CRC}(P \,\|\, W \,\|\, S) = \mathrm{CRC}_{base} \;\oplus \bigoplus_{b:\, W[b]=1} C_b, \tag{8}$$

which requires at most 32 XOR operations (i.e., $O(1)$ time). Here, $\bigoplus$ denotes the cumulative bitwise XOR over all terms, i.e., $\bigoplus_{b:\, W[b]=1} C_b$ means XORing together all C_b for which $W[b] = 1$. The constants $\{C_b\}$ depend only on the fixed suffix length $|S|$ and the fixed placement/byte order of the 4-byte word, and CRC_{base} is precomputed per packet template. If packet layouts differ (e.g., the first packet in a WQE includes an additional header such as RETH), then $|S|$ changes; consequently, CRC_{base} and $\{C_b\}$ must be computed separately for each distinct template.

A.2 Implementation Details

Fast control responses: ACK/NACK/CNP. Anytest prepares control packets according to the same packet model as detailed in § 4.1 and updates only the minimal runtime header fields and ICRC using the same mechanisms. Upon detecting trigger conditions (e.g., `AckReq`, ECN marks, or injected loss/reordering patterns), Anytest immediately emits the corresponding control packet without constructing packets or recomputing full ICRC.

Patterned injection via a configurable FSM. Beyond gap control, RP tests require structured stimulus patterns (e.g., scripted CNP injections). Anytest encodes these behaviors in a config-driven finite state machine (FSM) with time- and counter-based transitions. Each state specifies actions (e.g., inject n CNPs) and transition predicates (e.g., after t μs, after n packets, upon observing an RX condition). The FSM is out of the critical path of RoCEv2 emulation (5) enables high expressiveness without sacrificing packet rate.

Listing 1 shows the simplified configuration used to produce Figure 13a: an RP test that probes DCQCN rate-recovery behavior by injecting CNPs at varying intervals.

Listing 1: Configuration for the CNP-interval RP test in Figure 13a. The `cnp_state_machine` section defines a four-state timeline that (1) waits 2.5 s for the flow to reach steady state, (2) injects a single CNP, (3) waits 2 s to observe recovery, and (4) injects periodic CNPs at 500 μs intervals for 2 s.

```
# Peer hosts and DCQCN role assignment
servers:
  peer_ip: ["10.0.0.1", "10.0.0.2"]
  np: 0 # Notification Point
  rp: [1] # Reaction Point(s)

# Experiment parameters (traffic shape, mode)
experiment:
  exp_mode: "SENDER"
  qp_cnt: 1
  customed_mtu: 5 # 4096 B MTU
  auto_stop_seconds: 7 # the experiment duration

# Time-sequenced CNP injection timeline
cnp_state_machine:
  enabled: true
  states:
    # State 0: silent, let flow reach steady state
    - {duration_us: 2500000, send_cnp: "NONE"}
    # State 1: inject exactly one CNP
    - {duration_us: 1, send_cnp: "ONCE"}
    # State 2: observe single-CNP recovery
    - {duration_us: 2000000, send_cnp: "NONE"}
    # State 3: periodic CNPs at 600 us interval
    - {duration_us: 2000000, send_cnp: "MANY",
       cnp_interval_us: 500}

# Shell commands run before the experiment
# (e.g., configure NIC QoS / DCQCN parameters)
pre_launch:
  enable: true
  host: "10.0.0.2"
  pre_launch_cmd: [...] # omitted

# Shell commands run after the experiment
# (e.g., restore default NIC settings)
post_launch:
  enable: true
  host: "10.0.0.2"
  post_launch_cmd: [...] # omitted
```

The FSM executes its state list sequentially in a loop. Each state specifies three parameters: (i) `duration_us`—how long the state lasts; (ii) an execution mode—`NONE` (no action), `ONCE` (fire the task once on entry), or `MANY` (fire repeatedly at a specified interval); and (iii) `cnp_interval_us` (or `rdma_interval_us` for

QP-side state machines)—the repetition interval in `MANY` mode. Transitions are purely time-driven: once `duration_us` elapses, the FSM advances to the next state; after the last state it wraps to the first, repeating until the experiment ends. A second FSM variant (`qp_state_machines`) applies independently to each QP on the sender/RP side, additionally accepting per-state `rdma_len` to control the RDMA write size and per-group `max_depth` to bound outstanding work requests.

Latency measurement and cross-domain alignment. For latency-critical tests, receiver-side timestamps alone are insufficient because the injected event (e.g., WQE post or gap insertion) occurs in the sender's software clock domain. Anytest therefore records sender-side software timestamps for injected events using a lightweight side thread. To align sender software time with receiver hardware time, Anytest runs short calibration phases on the unloaded network before and after the test (1 s each). During calibration, the sender continuously transmits small packets at a rate below 1 Gbps to make sure the network is uncongested. And Anytest records per-packet sender and receiver timestamps. From the timestamp pairs, Anytest estimates the clock offset and relative drift between the two domains. For each latency measurement in the test, Anytest then subtracts the calibrated baseline latency, yielding a *delay delta* relative to the unloaded-network baseline. This delay delta is sufficient for our localization by comparing the delta deviations between a failing DUT and a reference under the same stimuli.

B RDMA-TRACER

RDMA-tracer is an RDMA tracing tool that logs the RDMA applications' behavior without any application modification. It leverages the fact that applications commonly call RDMA's dynamically linked library during application boot up and hijacks the linking procedure. It provides the same API as a normal RDMA dynamically linked library, but rewrites `ibv_open_device`, the function that all RDMA applications call before performing any RDMA operations. The rewritten `ibv_open_device` allows us to intercept all subsequent RDMA operations, such as `ibv_post_send`, `ibv_post_recv`, *etc.* The intercepted functions generate function call metadata, such as timestamp, arguments, and QP number, and push the metadata to a circular buffer. A separate logging thread/process (depending on the user's configuration) reads from the buffer and logs to a file system to minimize the logging overhead.

For the customers whose applications are statically compiled, we offer *RDMA-tracer*'s source code and can intercept their application's RDMA trace by recompiling their application. Note that we will ask for the customer's consent before tracing.

RDMA-tracer's overhead highly depends on the applications' behavior. For each RDMA API call, we measure less than 10% average logging overhead. Since RDMA applications commonly issue large RDMA messages to improve network throughput, the end-to-end overhead is often negligible. For example, we intercepted all NCCL's RDMA verbs on a 16-GPU GPT-3 training job, and the overall throughput only drops by 0.7%. We want to highlight that it's more important to capture the applications' RDMA pattern rather than getting microsecond-accurate timestamps. Because we can always adjust the trace manually afterward, but the communication relationship and data volume are more critical to resolve the NPAs.

Handling application dependencies during trace replay. When replaying a captured trace, *RDMA-tracer* should respect the original application's communication dependencies. *Cross-node* dependencies (e.g., a send that waits for a remote completion) are recovered via WQE-to-CQE matching: each WQE is paired with its corresponding CQE across the two endpoints, restoring the causal order. *Intra-node* dependencies (e.g., chained posts on the same QP) are preserved by replaying WQE in their recorded local order. Even when dependency recovery is incomplete, the recorded `ibv_post_send` timestamps have already "baked in" the delays introduced by application-level dependencies; replaying at those timestamps therefore reproduces the wire-level packet pattern that originally triggered the transport-layer NPA.

C SYNAPSE

Synapse is an RDMA traffic generator written in Go. Similar to the *perftest*, but it provides richer customization capability. It employs a single-master, multi-worker architecture. By feeding in a trace, the master distributes the workload to all workers and together, they start replaying the trace. *Synapse* offers great customizability, such as the sending and receiving NIC, RDMA verbs, data sizes, MTU, *etc.* It also supports directly replaying the trace collected from the *RDMA-tracer*.

One interesting use case of *Synapse* is to replay RDMA traces generated by NCCL. *Synapse* can re-generate the training job's RDMA traffic without actually requiring a GPU cluster. This allows us to evaluate the switch and NIC's CC setting without GPU servers, which have strict requirements IDC infrastructure, such as power delivery and rack configurations.