RoGUE: RDMA over Generic Unconverged Ethernet

Yanfang Le
University of Wisconsin-Madison
yanfang@cs.wisc.edu

Brent Stephens*
University of Illinois at Chicago
brents@uic.edu

Arjun Singhvi
University of Wisconsin-Madison
asinghvi@cs.wisc.edu

Aditya Akella
University of Wisconsin-Madison
akella@cs.wisc.edu

Michael M. Swift
University of Wisconsin-Madison
swift@cs.wisc.edu

ABSTRACT

RDMA over Converged Ethernet (RoCE) promises low latency and low CPU utilization over commodity networks, and is attractive for cloud infrastructure services. Current implementations require Priority Flow Control (PFC) that uses backpressure-based congestion control to provide lossless networking to RDMA. Unfortunately, PFC compromises network stability. As a result, RoCE's adoption has been slow and requires complex network management. Recent efforts, such as DCQCN, reduce the risk to the network, but do not completely solve the problem.

We describe RoGUE, a new congestion control and recovery mechanism for RDMA over Ethernet that does not rely on PFC. RoGUE is implemented in software to support backward compatibility and accommodate network evolution, yet allows the use of RDMA for high performance, supporting both the RC and UC RDMA transports. Our experiments show that RoGUE achieves performance and CPU utilization matching or outperforming native RDMA protocols but gracefully tolerates congested networks.

CCS CONCEPTS

• **Networks → Transport protocols**;

KEYWORDS

Congestion control, RDMA, datacenter transport

ACM Reference Format:
Yanfang Le, Brent Stephens, Arjun Singhvi, Aditya Akella, and Michael M. Swift. 2018. RoGUE: RDMA over Generic Unconverged Ethernet. In *Proceedings of SoCC '18: ACM Symposium on Cloud Computing, Carlsbad, CA, USA, October 11–13, 2018 (SoCC '18),* 12 pages. https://doi.org/10.1145/3267809.3267826

1 INTRODUCTION

Remote Direct Memory Access (RDMA) provides direct access from user mode, transferring data (segments) directly to and from application buffers. RDMA bypasses expensive system calls, and performs packetization and packet parsing in hardware. Furthermore, it enables one-sided operations, whereby one side of a connection can be handled entirely in hardware with no host software involvement. Thus, RDMA significantly improves both end-to-end latency and CPU overhead compared to TCP/IP networking [7, 16]. Furthermore, recent work has shown that these benefits of RDMA can speed up cloud applications and services, such as key-value stores, replicated state machines, and HPC workloads [7, 14, 16, 24]. As such, RDMA-enabled NICs (RNICs) are increasingly being rolled out in data centers and cloud services in both public and private settings [11, 14].

Originally developed for boutique Infiniband networks in high-performance computing, RDMA over Converged Ethernet (RoCE) provides RDMA capabilities over Ethernet networks. However, RDMA protocols continue to assume a lossless network. As a result, RoCE traffic requires a suite of enhancements to coexist with standard Ethernet traffic [2, 5, 6, 8]. In particular, it requires Priority Flow Control (PFC) to enable lossless forwarding. PFC causes switches/hosts to generate pause frames when their buffer occupancy crosses a threshold, which throttles senders. Thus, PFC provides congestion control for RDMA networks. Furthermore, with PFC enabled, packet losses only occur due to rare bit corruptions, from which RNICs recover using simple hardware retransmission schemes.

Unfortunately, PFC has known fundamental issues that create serious negative side-effects [11, 20, 26, 28, 31]. Latencies can increase due to head-of-line (HOL) locking [20, 26, 31]. Malfunctioning devices and incorrect routing can deadlock an entire network [4, 11, 26, 28]. Due to these hazards, datacenters and cloud administrators today either limit the scale of their RDMA deployments or disallow RDMA altogether.

Our goal is to develop an approach that *preserves the benefits of running RDMA for commodity Ethernet networks but without any reliance on PFC.* Giving up on PFC means we must design new schemes for congestion control and recovery from congestion-induced losses.

A key question is where and how to implement these functionalities. One possibility is to implement them entirely in RNIC hardware. We argue that such hardware-based approaches have crucial drawbacks (Section 2.4). First, administrators, especially of small- to medium-scale on-premise/private clouds, cannot rely on RNIC vendors to roll out custom hardware changes; thus, they may be forced to deal with legacy RNICs. Second, approaches baked into hardware cannot handle non-standard protocol implementations in switches (Section 2.4), nor can they handle evolution in switch functionality or end-host congestion control protocols.

*Work done while at University of Wisconsin-Madison

Thus, to support backward compatibility and network evolution, we seek a *software-based solution*. However, designing such a solution is non-trivial. First, indiscriminate use of software-driven control, e.g., using software to pace each packet at the appropriate rate, or to identify and recover from packet losses, can severely undermine RDMA CPU and latency benefits. Second, signals of congestion that traditional schemes leverage, e.g., packet drops and ECN bits, are consumed by the RNIC and not available in software. Third, achieving low CPU overhead with RNICs requires leveraging their native support for certain crucial functions, e.g., high performance messaging, high precision timestamping, and rate limiting. Together these imply that in order to meet our goal we must strike the right balance and leverage hardware support judiciously in software.

Our RDMA transport layer, RoGUE, achieves this judicious division of labor. It is implemented as a shim layer above OpenFabrics Enterprise Distribution (OFED) userlevel API [23]. It lifts RDMA congestion control and loss recovery functionality into software, while leveraging existing hardware to assist software and to accelerate performance. It works with both RC and UC RDMA transports (Section 2.4).

RoGUE offloads expensive messaging operations to hardware and implements core congestion control logic in software. In particular, it transforms an input set of application segments into RoGUE segments whose size is optimized to balance RoGUE's CPU overhead with its ability to react quickly to congestion. RoGUE relies on delay to estimate and respond to congestion, instead of packet drops and ECN bits. This is because ECN marks only indicate network congestion, while delay can indicate congestion in both the RNIC and the network. Furthermore, today's RNICs consume ECN marks, making them transparent to software. Our custom algorithms provide accurate and low overhead RTT estimates for different RDMA transports using RNICs' hardware timestamping support, coupled with tuning RDMA signaling frequency from software. We leverage the TCP Vegas algorithm [1] for congestion response. RoGUE uses a congestion window to ACK-clock verbs from software, coupled with hardware rate limiters to pace individual packets out the RNIC; together these ensure stable congestion control behavior.

RoGUE's congestion control helps it keep queue lengths low and drops to a minimum even without PFC turned on. When occasional drops do occur, RoGUE first relies on the RNIC's hardware retransmission scheme. But when the scheme's inefficiency and slow reaction can hurt throughput (e.g., under burst losses), RoGUE uses a backup *shadow queue pair* to drive retransmissions from software without losing performance.

We conduct an extensive evaluation of RoGUE over a testbed consisting of 32 servers connected using 10 Gbps RNICs. We find that RoGUE's use of large segments helps keep CPU overhead low (17% vs. 5% of one CPU with native RoCE that uses 4X larger segments than RoGUE). RoGUE can adapt quickly to congestion while keeping queuing low. RoGUE offers comparable throughput to DCTCP, but results in substantially lower network latency, nearly one-third for small RPC traffic. Finally, RoGUE offers fair allocation to large application flows, and the lowest completion times, when compared with alternatives, for short flows.

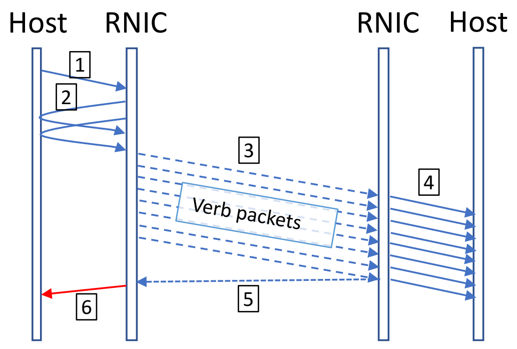


Figure 1: RC Workflow.

2 BACKGROUND AND MOTIVATION

We start with overviews of key RDMA protocols and RoCE. We then highlight the drawbacks of current RDMA deployments, and end with a specification of the problem we seek to solve and our objectives.

2.1 RDMA Preliminaries

With RDMA, user-space applications invoke the kernel to allocate a NIC queue and then establish a connection to a remote machine. Applications interface directly with RNICs using a client library to send RDMA **verbs** to queues. The most commonly used verbs are READ, WRITE, SEND and RECV. READ fetches data from the memory of a remote host, and WRITE transfers data into the memory of a remote host. READ and WRITE are considered "one-sided" because they only require host activity on the sending end: the RNIC at the receiver processes the request without software involvement. SENDs are "two sided": they transfer a message to a remote host, where software enqueues a RECV verb to receive the message. The data size in a verb can generally be up to 2GB. WRITE_WITH_IMM and SEND_WITH_IMM are variants of WRITE and SEND that carry an additional 4 bytes of immediate data that can be read by software on a remote host.

READs, WRITEs, and SENDs are posted to *send queues*. RNICs also create *receive queues* when a connection is set up. For READ and WRITE, receive queues are ignored, while for SENDs software must post corresponding RECVs to them. All send and receive queues are paired, and each queue pair (QP) is associated with a completion queue that signals the completion of events and delivers immediate data. User-space applications can use interrupts or poll the completion queue.

RDMA supports three **transport types**: Reliable Connection (RC), Unreliable Connection (UC), and Unreliable Datagram (UD). SEND/RECV are supported by all transports as they have the least requirements. WRITEs are supported by RC and UC, but READs are only supported by RC. In RC, the RNIC is responsible for retransmissions following a loss. In contrast, packet drops are ignored by the RNIC in UC and UD.

RNICs can **signal** the application of the completion of a verb via an event. If an application does not need to be notified when a verb completes, then it can disable signaling, which reduces CPU utilization. Figure 1 shows the complete sequence of operations for a WRITE verb using RC. (1) Host software enqueues the verb.

(2) The RNIC then uses DMA to retrieve the data for the write from host memory and (3) transmits it over the network in one or more packets. (4) The receiving RNIC writes the data to memory, and after receiving the last packet, (5) sends back an ACK. The sending RNIC, on receiving the ACK, (6) optionally generates a signal notifying the application that the WRITE completed (red arrow). UC verbs are similar except that the receiving side does not ACK and signals are delivered when the last packet of a verb is sent, not when an ACK is received.

2.2 RoCE

RDMA originally required a lossless Infiniband network. RDMA over Converged Ethernet (RoCE and RoCEv2) enables RDMA over commodity Ethernet networks. It achieves this by using Priority Flow Control (PFC) [6] to avoid congestion losses and RNIC retransmissions to recover from drops due to corruption.

Lossless forwarding: PFC-configured devices generate **pause** frames when their buffer occupancy exceeds a threshold. As congestion persists, pause frames propagate back towards the source of the congestion, exerting "backpressure," slowing senders, and preventing buffers from filling more. RoCE uses PFC to avoid packet drops caused by congestion.

Loss recovery: With PFC enabled, packets can only be dropped due to corruption. RNICs automatically handle such losses without software intervention. Because such losses are rare, RNICs do not implement sophisticated retransmission schemes. When an RNIC receives a NACK or times out waiting for an ACK, it retransmits the remainder of the verb starting with the dropped packet even if packets following the drop were already successfully delivered.

2.3 Drawbacks of PFC

While PFC enables applications to extract the performance of RDMA, its drawbacks make it difficult and risky to use in a datacenter. A significant problem is that a single malfunctioning device can cause the network to become *deadlocked* and unable to forward packets [11, 28]. Incorrect routing can also lead to deadlock [4, 11, 26]. Because routing must now be deadlock-free, network throughput on some topologies is reduced [26]. PFC also suffers from bufferbloat, HOL-blocking, and unfair packet scheduling [26, 31].

Recent advances in datacenter congestion control, notably DCQCN [31], mitigate how often PFC is invoked. However, serious problems remain: a slow or malicious host that stops reading from its receive queue can generate pause frames and deny other tenants access to the network [31]. Recent works on pervasive monitoring [11] partially address this problem by identifying when such issues are about to occur. Yet, the inherent limitations of monitoring, such as inaccuracies and inability to scale to large deployments, mean that serious PFC problems still arise in production [11].

2.4 Problem Statement

These drawbacks lead to the main question we address: can we extract the benefits of RDMA on commodity Ethernet networks without **any** reliance on PFC? Because PFC is central to avoiding congestion and losses, answering the above requires rethinking *RDMA congestion control and recovery mechanism in a way that retains the latency and CPU benefits of RDMA but tolerates congestion-induced losses.*

The main design issue we face is whether to implement the above schemes in **software or hardware**.

Congestion control: Hardware approaches to congestion control suffer from a number of problems [22]. In particular, there are three situations where software congestion control is superior to reliance on the existence of custom RNIC hardware or switch support.

First, while large cloud operators such as Google and Microsoft can work with vendors to roll out custom switch/NIC features that incorporate advancements for congestion control, this is often not possible for admins of enterprises with small to medium sized private datacenters and clouds (see Judd [15] for an example perspective from Morgan Stanley). Such operators are limited to inflexible commodity hardware. Software congestion control enables such clusters to extract RoCE's benefits without the pain of PFC.

Second, clusters (both big and small) may have heterogeneous network hardware with non-standard protocol implementations. For example, the switches in CloudLab [3] that we use for experiments have a non-standard implementation of RED [9] (Section 4). This can hurt throughput of hardware congestion control schemes, e.g., DCQCN [31], a state-of-the-art approach implemented on RNICs that relies on ECN marks. Others have reported similar issues [15]. In some networks, ECN may not be available at all [15].

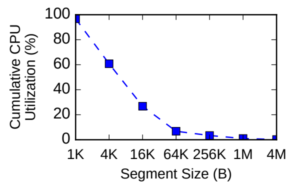
Third, even with correct and homogeneous network support, implementing congestion control in hardware complicates network evolution. For example, DCQCN requires that switches implement RED [9] to decide which packets to mark. Thus, using an alternate in-network AQM scheme would require updating the DCQCN algorithm at hosts. Datacenter congestion control is rapidly evolving [10, 12, 20, 29, 31], with tens of novel congestion control algorithms proposed every year. Implementing congestion control in hardware can delay or prevent the adoption of these new algorithms.

Loss recovery: Existing RNICs' loss recovery mechanisms were designed to work in a low packet-error regime. With PFC turned off, however, flows can experience severe loss, e.g., under a burst of newly starting connections. Unfortunately, the RNICs' hardware mechanisms to recover from these losses are very inefficient in this case: it can take up to a few hundred milliseconds to detect a drop and recover. For the same reasons as above, we cannot rely on custom hardware to address these inefficiencies.

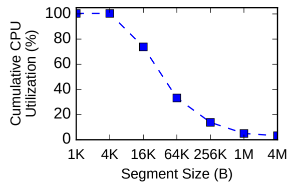
Division of labor: Designing entirely software-based approaches faces two challenges:

(1) Hardware masks congestion signals: Congestion signals are not always available or precise in RDMA, and per-packet RTT measurements are unavailable. For example, in the RC transport, some losses are handled by the RNIC and may be transparent to software and, in UC, losses are silently ignored by the RNIC. Likewise, ECN marks on packets are consumed by the RNIC and are transparent to software.

(2) Software is inefficient: Implementing the needed functionality, e.g., per-packet pacing, RTT calculation, congestion window updates, tracking and retransmitting losses, etc., as it happens in today's TCP stacks, entirely in software is possible by using RDMA verbs as small as a packet and signaling per small verb. However, this creates unreasonable CPU overhead (Section 3.1).



(a) 10 Gbps



(b) 100 Gbps

Figure 2: CPU utilization of a single core using READs to saturate a link, signaling once per segment. 10 Gbps results use an Intel Xeon D-1548 CPU with a ConnectX-3 Pro [18]; 100 Gbps results use Intel Xeon E5-2660 CPU with a ConnectX-4 [19].

But, hardware presents an *opportunity*. Almost all NICs provide native support for several functions, e.g., RNICs implement messaging operations in hardware; NICs (including RNICs) today provide precise hardware timestamps; and all NICs support hardware rate limiting. By carefully leveraging this via a suitable division of labor between hardware and software, we can overcome the above challenges.

3 ROGUE DESIGN

RoGUE is a new transport that enables use of existing RDMA hardware to achieve near-native performance on unreliable networks. It lifts congestion control and loss recovery functionality into software, while using hardware to assist software's action and for common-case performance acceleration.

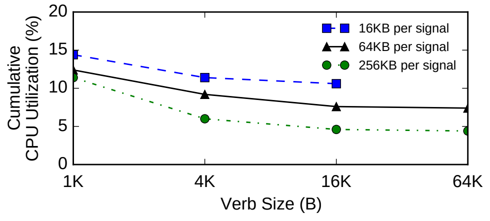
RoGUE is a layer above OpenFabrics Enterprise Distribution (OFED) userlevel API [23], and takes as input a set of (large) verbs from an application. It then implements congestion control by (i) transforming those verbs into smaller segments for transmission and imposing just the right amount of signaling for feedback, (ii) using RTTs as well as drops to estimate congestion, and a window to clock out verbs, and (iii) imposing hardware rate limits to pace out packets.

In order to provide low overhead, good performance, congestion control and reliability, the key questions that must be answered by RoGUE are: (a) how to size segments? (b) how can congestion be estimated and effectively controlled given RNIC capabilities and constraints? (c) how to accommodate different underlying transports (e.g., RC vs. UC)? and (d) how to recover from congestion-induced losses?

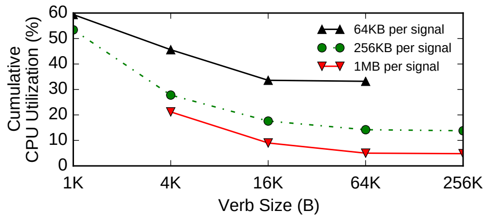
3.1 Transmitting Data

A key benefit of RDMA is that using *large segments* and offloading packetization and parsing to the RNIC minimizes CPU involvement and load. But, large segments also reduce the granularity of feedback from the network which impacts congestion control.

Thus, central to our design are two key considerations: *segmentation*, or how large a verb should RoGUE transmit; and *signaling*, or how often should the RNIC notify software that a verb completes. RoGUE uses large enough segments and/or infrequent enough signaling to preserve most of the CPU utilization benefits of RDMA but not sacrifice the ability to respond to congestion.



(a) A 10 Gbps ConnectX-3 Pro [18]



(b) A 100 Gbps ConnectX-4 [19]

Figure 3: The CPU utilization of infrequent signaling for different small verb sizes.

We take a quantitative approach to determine the best segment size and signaling granularity. We consider two different RNICs: the 10 Gbps ConnectX-3 Pro and the 100 Gbps ConnectX-4 RNIC. We used the `ib_read_bw` OFED RDMA benchmark tool to drive line-rate using different sized READs while using the `dstat` tool to measure CPU utilization. The experimental setup is given in Section 4. The results of our experiments are shown in Figures 2 and 3. We expect our approach and results to generalize to other RNICs with similar line-rates.

From Figure 2, we observe that large verbs have low CPU utilization even at once-per-segment signaling frequency. Signaling every 64KB verb uses less than 10% utilization on the ConnectX-3 Pro, and signaling every 256KB verb uses less than 15% utilization on the ConnectX-4. Thus, to receive frequent feedback about network conditions, RoGUE chops large verbs into 64KB at 10 Gbps and 256KB at 100 Gbps, and signals every verb. Even larger-size RoGUE verbs would be slightly more CPU efficient, but they provide less frequent feedback about network conditions, hurting the ability to respond to congestion.

When an application issues small verbs (< 64KB and < 256KB), we can see from Figure 2 that signaling every verb can result in high CPU: signaling every 16K verb at 10 Gbps (100 Gbps) incurs 30% (70%) CPU! However, Figure 3 shows that infrequent signaling, e.g., once per 64KB (256KB) of data at 10G (100G), drops CPU use to 8% (18%) of one core for 16KB verbs. As noted before, making signaling even more infrequent can hurt congestion response ability. Thus, RoGUE keeps small application verbs (< 64KB) as they are, but signals every 64KB and 256KB at 10 Gbps and 100 Gbps line-rates unless the application requests signals more often. When this is the case, RoGUE does not require any additional signaling. Henceforth, we define *batch* as the collection of bytes for which we receive one signal.

This design prioritizes CPU efficiency over faster reaction, similar to Linux where interrupt coalescing, generic receive offload, and delayed ACKs inflate the time to react to congestion in order to reduce CPU overheads.

A final issue we must consider is *starvation*. RoGUE uses a congestion window, similar to TCP, to avoid sending data too fast. Thus, new data will not be sent until prior sends have been acknowledged by the receiver. On high-speed networks, the congestion window may be as small as 16KB. When the batch size exceeds the congestion window, the RNIC will starve if just one batch is enqueued because RoGUE waits for a signal of the prior batch's completion before enqueuing verbs from the next batch, and these signals are not received until one RTT after the last packet of a batch was sent. To avoid starvation, RoGUE therefore ensures that at least 2 batches are enqueued if there is still application data to send.

3.2 Congestion control

Overview: A basic question is whether to rely on drops or delay as the congestion signal. In RDMA, signals from packet drops are not immediately available to software (Section 2.4). Thus, RoGUE heavily uses delay as the congestion signal. RoGUE uses a congestion window, as opposed to rate, to transmit segments. The use of congestion window limits the total number of outstanding segments and allows RoGUE to ACK clock segments in a batch and avoid congestion collapse. However, the packets in a segment are transmitted at line rate, and therefore RoGUE can momentarily generate a burst of packets into the network resulting in heavy losses. To avoid this, RoGUE configures hardware rate limiters, based on current congestion window and RTT estimates, to gradually pace packets out.

3.3 Details

With this overall description in mind, we now describe the low level congestion control behavior in RoGUE.

Connection startup: Packet loss is possible if connections start by transmitting at line rate, which can dramatically increase latency and lower throughput. To avoid this, RoGUE uses slow-start. When a QP is first created, RoGUE starts transmission by sending an initial congestion window worth of data at line-rate. After that, RoGUE doubles the congestion window until congestion is detected. Like Linux, we use an initial congestion window size of 10 packets.

RTT measurement: RoGUE's use of RTT as a congestion signal benefits from the fact that RNICs provide a high precision timer for timestamping packets when they arrive at the RNIC. That said, RTT calculation in RoGUE is not straightforward because the RNIC interface operates on large verbs and not single packets. Consider Figure 4 where a batch of two signaled verbs is enqueued. First, $Verb1$ is enqueued when the NIC is idle, and then $Verb2$ soon after (to avoid starvation). However, because of the congestion window, $Verb3$ cannot be enqueued until the RNIC signals the completion of $Verb1$. Figure 4 illustrates that $t_{enq\_s_i}$ (e.g., $t_{enq\_s_2}$) is not always accurate to use as the start time of a batch ($Verb2$) because the RNIC may still be transmitting an earlier batch ($Verb1$). However, the completion time of the last verb of the previous batch ($Verb1$) is known, and this time is exactly one network RTT after the last byte of the verb was sent, which is also when the first byte in the current batch ($Verb2$) will be sent.

RoGUE uses this property to compute RTT as follows. Whenever there is room in the congestion window to send more data, RoGUE enqueues as many batches as possible. First, the NIC time is read

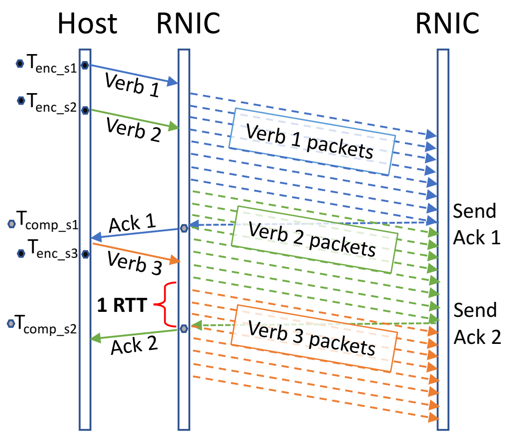


Figure 4: An example of how batches of 1 signaled verb are enqueued and signaled in RoGUE. These events are used to compute RTT.

before enqueuing a batch S_i ($t_{enq\_s_i}$). The signal of acknowledging the batch of B_{sig} bytes from the remote RNIC is also recorded ($t_{comp\_s_i}$). Then, the expected start time of the first verb in the batch is:

$$t_{start\_s_i} = max(t_{enq\_s_i}, t_{comp\_s_{i-1}} - RTT_{i-1})$$

Then, with *rate_limit* being the current applied rate-limit, the RTT sample for batch S_i is computed as:

$$RTT_i = t_{comp\_s_i} - t_{start\_s_i} - \frac{B_{sig}}{rate\_limit}$$

This reflects all the queuing delay at the sender and in the network but not the delay incurred by the NIC serializing the batch of verbs. RoGUE also records the lowest RTT it has seen as the *base RTT* of the network.[1]

Congestion control: To control sending rate, RoGUE relies on: (1) RTT-based updates to the congestion window and (2) hardware rate limiter configuration.

RoGUE's updates to congestion are the same as TCP Vegas [1]. Briefly, at the end of every batch, the RTT estimate *curr_rtt* is used to estimate the difference between the expected and actual congestion windows as follows:

$$diff = cwnd * (curr\_rtt - base\_rtt)/current\_rtt$$

If $diff > \beta$, RoGUE additively decreases $cwnd$; if $diff < \alpha$, RoGUE additively increases $cwnd$. α and β are constants in TCP Vegas.

Hardware rate limiting: Transmitting the verbs in each batch at line-rate would create a burst of packets that can lead to congestion and drops. To avoid sending a burst of packets, RoGUE uses *rate-limiters* in RNICs to perform packet pacing.

Ideally RoGUE is able to program rate-limiters on a per-QP basis with a rate-limit of:

$$rate\_limit = cwnd/base\_rtt$$

[1]RoGUE lower-bounds base RTT with an estimate of the minimum RTT possible for all destinations.

Rate-limiters are crucial when *cwnd* < 64KB. In such cases, if *cwnd* were the only mechanism to control transmission of packets, then RoGUE would be forced to use small *cwnd*-sized verbs, leading to high CPU utilization. In contrast, using rate-limiters that are configured based on the *cwnd* value helps RoGUE use large segments in this operating regime by ensuring that no more than *cwnd* of data is injected into the network per RTT.

A practical challenge we faced in applying hardware rate limiters is that the new rate limit may not take effect immediately. Indeed, in two different Mellanox NICs (ConnectX-3 and ConnectX-4), we measured that it can take up to 160*KB* for the new rate to take effect (for various combinations of new/old rate). In such a case, RTT samples computed using the approach above are likely to be incorrect until the rate limit kicks in. In turn, this leads RoGUE to take incorrect congestion response.

To overcome this, each time a new rate limit is computed and applied, RoGUE holds off taking any RTT samples for the upcoming H = 160KB of data transmitted. Other RNICs may have different values for H. This holding off, however, causes congestion response behavior to be somewhat slow on occasion: RoGUE does not collect congestion signals for 160KB after rate limiter update. In practice, we have found this to have little impact on performance (Section 5).

Also, not all RNICs provided per-QP rate limiters. While new RNICs like the ConnectX-4 [19] do, the ConnectX-3 Pro [18] does not. It does, however, support per-priority rate limiters. Thus, we assign each queue pair a separate priority, and each priority is assigned its own rate limit. The prototype of RoGUE builds on top of ConnectX-3 Pro. RoGUE limits the number of active QPs to the number of rate limiters — 8 for the ConnectX-3 Pro. Furthermore, to avoid dramatic rate swings, RoGUE bounds the change in rate to 1 Gbps.

3.4 Transport Specific Design

Next, we discuss the differences that arise in applying the above congestion control algorithm to the RC and UC transport types. RoGUE is the first approach to offer congestion control for the UC transport type. Supporting the UC transport type is important because recent work has shown that it is more scalable than RC [16, 17] and because UC traffic can impact other congestion-controlled traffic such as RC or TCP traffic.

RC Transport: For WRITE and SEND verbs over RC, RoGUE closely follows the design above. The main issues in RC arise due to the READ verb. First, maintaining a congestion window is complicated because congestion in WRITE and READ are caused in the opposite direction. To overcome this, RoGUE maintains independent READ and WRITE congestion windows for a single RC QP. Second, READ requires setting rate limits on the remote host. RoGUE uses a receiver-side library that asynchronously applies the limit to a QP, and sends a WRITE_IMM to the remote host with the READ rate-limit whenever it changes.

UC Transport: Unlike the RC transport, the RNIC does not generate ACKs in the UC transport. Because of this, it is not immediately possible to use signals to compute network RTT. Signals in the UC transport only indicate that the message has been sent on the network, not that it has been acknowledged by the remote RNIC.

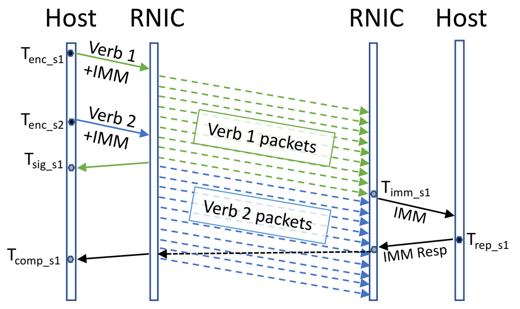


Figure 5: Timestamping positions with UC transport.

To overcome these limitations, RoGUE emulates the effect of ACKs in RC by including the remote host in computing the RTT. In each batch, RoGUE modifies the last verb to pass immediate data, which triggers the RoGUE library on the remote host to reply via a WRITE_IMM of size 0. Figure 5 shows the messages and when timestamps are taken.

To compute the RTT, we measure in software when packets are enqueued: $t_{enq\_s_i}$ when the sender enqueues a batch, and $t_{rep\_s_i}$ when the receiver enqueues a WRITE_IMM response. Hardware signals provide the timestamps of when the last verb in the batch completed transmission ($t_{sig\_s_i}$) and when the WRITE_IMM from the server is received ($t_{comp\_s_i}$). We compute the start of a batch as:

$$t_{start\_s_i} = max(t_{enq\_s_i}, t_{sig\_s_{i-1}})$$

The processing time on the server to generate the reply is

$$T_{response} = t_{rep\_s_i} - t_{imm\_s_i}$$

or the timestamp of when the reply was sent less the RNIC timestamp of when the immediate data was delivered. The server passes this value back in the immediate data. We ignore the small delay between enqueuing and transmitting the reply. From these values, the RTT can be computed as:

$$RTT_i = t_{comp\_s_i} - t_{start\_s_i} - t_{response} - \frac{B_{sig}}{rate\_limit}$$

Finally, RoGUE must be able to handle packet losses during RTT measurement in the UC transport. An RTT measurement for batch $j > i$ that arrives before the measurement for batch i is interpreted as a packet loss in batch i, and the congestion window is reduced multiplicatively. When all of the outstanding RTT measurements are dropped, RoGUE uses a timeout to retry sampling the RTT. After timeout, RoGUE resets the congestion window to its initial size and does not enqueue new batches until it successfully samples the RTT.

UD Transport: We do not address UD traffic because it is the RDMA equivalent of UDP, which does not use congestion control. That said, supporting congestion control for UD would enable it to coexist with other transports. Adding this support requires overcoming two challenges: (1) UD transport allows a single QP to send to multiple different destinations; and (2) only a single rate limit can be set per QP. We leave this for future work.

3.5 Reliability

When drops occur, the RNIC retransmits in hardware. Unfortunately, the performance of RNIC-based retransmissions can be poor under heavy losses. RoGUE's aforementioned design naturally accommodates the hardware's inefficacy.

For UC, RoGUE silently drops data similar to ROCE. Only RC provides reliable service to applications. For RC, RoGUE's delay-based congestion control scheme keeps queues small and losses become very rare. Nevertheless, when losses do occur, e.g., due to unexpected burst of competing traffic, RoGUE uses a combination of the RNIC's default hardware-based retransmission strategy, and a software driven approach as outlined below.

With RC, the RNIC provide two controls over how packet losses are handled. First, for each QP, software can set a timeout of how long to wait for a response or ACK before detecting a packet loss. We note that only when the last packet of a verb is lost does this occur, as normally the receiving RNIC will immediately send a NACK if it receives an out-of-order packet. Because of this, we use the QP timeout configured by the application. Second, RNICs allow a per-QP count of how many times to retry sending the verb before signaling a loss to software. Datacenter loss rates are generally low and RoGUE congestion control keeps queues small, so hardware retry has a very high probability of succeeding. Thus, RoGUE sets a low retry count of one.

Recovering from losses in software, while flexible, is difficult for two reasons. First, the RNIC does not notify RoGUE software of how much of the last verb was correctly transmitted. As a result, software must retransmit from the beginning of the verb. Second, the QP is placed into an error state and must be recovered before it can be used to issue additional verbs. From the state, the local and remote QPs must be synchronized. While most of the QP state has already been negotiated (*e.g.*, QP number, port, and GID index), this state also includes the dynamically changing packet serial number (PSN). If the PSN of the local and remote RNICs do not match, the remote RNIC will silently drop packets. Synchronizing the PSN requires extra communication, delaying recovery.

RoGUE masks this delay with a novel *shadow QP* mechanism. To avoid the delay of recovering queue pair state, RoGUE maintains a shadow QP for every active QP created. After a QP enters the error state, RoGUE immediately begins issuing verbs on its paired shadow QP while it re-synchronizes the original QP. Because inactive QPs use no RNIC resources [11], the overhead of maintaining shadow QPs is negligible. When losses occur, the shadow QP may be used immediately. Importantly, this does not require any network communication. While the shadow QP is used to issue subsequent verbs, RoGUE uses an additional UD QP maintained by the receiver-side library to re-synchronize the PSN of the failed QP, which becomes the new shadow QP.

3.6 RoGUE Library

RoGUE deviates from purely one-sided operations by relying on a lightweight receiver-side library. This library sits between OFED library and application, which gives much of the performance of RDMA at slight cost. It is used to set rate-limits for READs, provide efficient RTT estimates for the UC transport using verbs with "immediate" data (Section 3.1), and to re-establish connections following a timeout (Section 3.5). However, these are all background operations. RoGUE does not change any of RoCE's optimization options, such as inline data, signalling frequency, and immediate operations, which applications may leverage. If the application verb size is less than 64KB (small verbs), RoGUE will keep as it is. Note that the inline data optimization can only be used with small verbs. For large application verbs, RoGUE will signal and send immediate data on the last segment. In addition, the sender can continue issuing verbs before the receiver responds. Thus, these operations do not delay READs and WRITEs issued by the sender or otherwise interfere with their one-sided nature. Figure 6 quantifies the overhead of this library. At most, this library increases utilization by 9.8% of one CPU (Section 5).

4 METHODOLOGY

We evaluate RoGUE on a cluster of servers and in simulation, and compare against RoCE with and without PFC.

Hardware platform. We evaluate RoGUE with a Mellanox ConnectX-3 Pro 10 Gbps RNIC, which supports DCQCN. We use a cluster of 32 servers on CloudLab [3], each server with an 8-core Intel Xeon D-1548 CPU, and 64GB of memory. Each server connects to one of two HP Moonshot-45XGc ToR switches (16 servers per switch), which in turn connect to a HP FlexFabric 12910 core switch via a 40 Gbps uplink. All of the servers we use run Ubuntu 16.04 with Linux kernel version 4.4 and Mellanox OFED version 4.1.

For the Mellanox ConnectX-4 [19] 100Gbps experiments in Section 3.1, we use two servers each with two 10-core Intel Xeon E5-2660 CPUs, and 160GB of memory, which are directly connected to each other.

Network configuration. To enable both DCTCP and DCQCN, we configure the switch to perform ECN marking. We are able to enable ECN on the Moonshot-45XGc switches, but are administratively prevented from enabling ECN on the HP FlexFabric 12910 core switch. Thus we can only evaluate DCTCP and DCQCN with experiments where all traffic stays within a single Moonshot-45XGc switch.

The Moonshot-45XGc implements RED with ECN in a non-standard way that is detrimental to DCQCN. To get the best possible results from our hardware, we conducted a broad parameter sweep and found that these RED parameter settings offered the best performance for both DCTCP and DCQCN: $K_{min} = 99$ packets, $K_{max} = 38,000$ packets (the maximum allowed, so no unnecessary drops), and $P_{max} = 1$. We set TCP Vegas parameters α and β to 4 and 2 respectively.

Simulations. We used the open-source ns3-rdma simulator [30, 32] and added support for RoGUE to evaluate configurations not possible for us on real hardware. We evaluate a 40Gbps network environment with 1μs network delay, and second, we evaluate TIMELY and DCQCN: We used TIMELY's implementation included with the simulator.

5 EVALUATION

We conduct a detailed evaluation of RoGUE. We study its CPU overhead, the accuracy and effectiveness of RTT computation, and the efficacy of RoGUE's congestion control algorithm. Kernel bypass

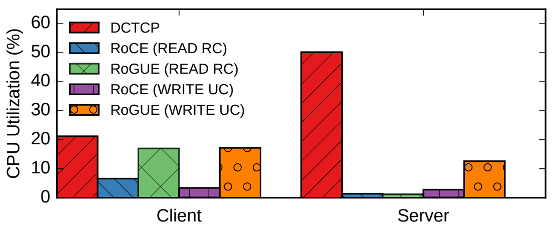


Figure 6: CPU utilization of network transports.

mechanisms such as mTCP [13] typically use polling for low latency, and hence 100% of a CPU. In addition, these mechanisms do not focus on congestion control. Thus, we do not compare RoGUE against them.

We study the benefits of RoGUE for two benchmark applications where we also contrast against competing systems. We conclude with a simulation-based comparison against DCQCN and TIMELY. We expect that RoGUE has comparable performance as RoCE, but lower CPU utilization and faster responsiveness than DCTCP.

5.1 CPU Utilization

We measure CPU utilization while using a QP or TCP flow to drive line-rate traffic between two machines in our testbed ("client" sending to a "server"), and compare DCTCP against RoCE and RoGUE for RC and UC transport types with READ and WRITE verbs, respectively. For READs, server sends data to the client. We use `dstat` to measure CPU utilization every 10s (avg. over 5 runs) at both the client and server.

Figure 6 compares the CPU utilization of RoGUE to both RoCE and DCTCP. RoCE uses 1MB verbs, whereas RoGUE uses 64KB verbs. We heavily optimize DCTCP: we enable TCP Segmentation Offload (TSO) with a segment size of 64KB, and generic receive offload (GRO). Also, we use `sendfile` to enable zero-copy transmissions of data that is already resident in the kernel; sending data from user-space would incur extra CPU for system calls and copying data.

For RC READs, RoGUE has higher CPU utilization at the client side (17% of one core) than RoCE (7%) because RoCE uses larger verbs and hence signals less frequently. RoGUE has lower CPU overhead than DCTCP (21%) despite the heavy optimizations we applied to the latter. For UC writes, RoGUE's CPU use is similar (15%), whereas RoCE's is slightly lower (5%) due to a simpler protocol.

At the server, RoCE has negligible CPU use with both RC and UC as both READ and WRITE verbs are one-sided. RoGUE with RC has negligible CPU use for the same reason. Despite RoGUE's library and use of immediate data with UC, it uses just 12% of a server CPU. In contrast, DCTCP's CPU use is high (51%): even though GRO is enabled, the driver at the DCTCP receiver must handle individual packets.

5.2 Loss Recovery

We evaluate RoGUE's loss recovery mechanism using an incast from two hosts, the minimum required to cause congestion, each sending a 128MB flow. We simulate loss with a switch ACL that causes some packets to be dropped. Because the RNIC generates RDMA traffic with monotonically increasing IP identification fields, we configure a switch ACL to drop packets with specific IP identification field [11]. For example, packet loss rate 1/4K is achieved by dropping packets with IP id field of 0x*fff. As the IP id field in header is 16 bits, the lowest packet loss rate possible with this mechanism is 1/64K.

Figure 9 shows the median throughput of big (64KB) and small (1KB) application verbs with different loss rates. As expected, throughput decreases as the packet loss rate increases for both verb sizes. The throughput for big verbs is better than for small verbs because of the RNIC's hardware retransmission scheme. Upon learning of a loss via a NACK or timeout, the ConnectX-3 Pro's RNIC's Go-Back-N mechanism immediately retransmits all the packets in the verb since the lost packet, but delays sending the next verb until the entire verb has been ACKed. This continues until all N packets between the lost packet and the packet sent just before learning of the drop, are retransmitted. For large verbs, the remainder of the verb is sent immediately, which likely includes all N retransmitted packets, so the next verb can be sent immediately. However, retransmission of small verbs are serialized and wait for an ACK of the preceding verb before being sent, leading to much more delay.

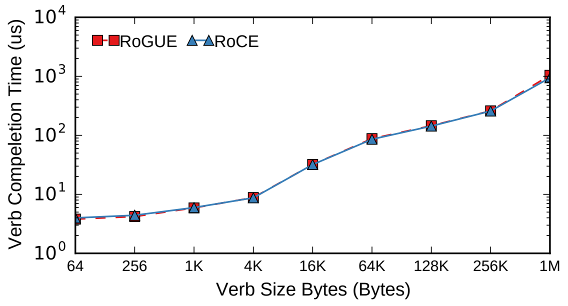


Figure 7: RDMA verb completion times for both RoCE and RoGUE given different verb sizes.

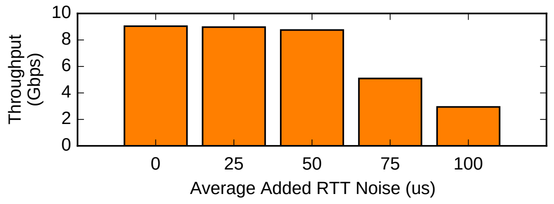


Figure 8: Impact of inaccurate samples on Throughput

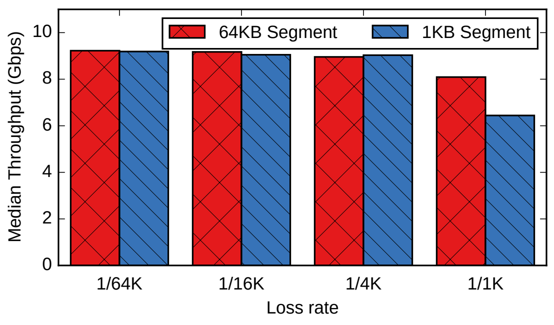


Figure 9: Impact of packet loss

Transport	10%ile	50%ile	90%ile
RC WRITE(0)	2μs	2μs	2μs
RC READ(64KB)	4μs	4μs	6μs
UC WRITE(64KB)	8μs	8μs	8μs

Table 1: Accuracy of RoGUE RTT computations.

We can see that RoGUE performs well when the loss probability is up to 0.025%. We believe that this is sufficient to handle almost all packet loss.

5.3 Verb Completion Time

Because of RDMA's low latency (μs level), it may be sensitive to the extra code RoGUE executes on the data path for congestion control. We measure the completion time for each individual verb by varying verb size, and compare RoGUE against RoCE. Figure 7 shows the completion time of RDMA READs in an uncongested network, averaged over 1M iterations. Small 64B verbs complete in 4μs, identical to RoCE. For large 1MB verbs, RoGUE requires 1040μs while RoCE requires 944μs due to the cost of reading timestamps from the RNIC and processing completion signals every $64KB$. These results show that RoGUE overheads do not significantly affect application performance.

5.4 RTT Computation

RTT computation is central to RoGUE as it drives *cwnd* updates and hence the sending rate. It is therefore also crucial to avoid losses as much as possible.

Accuracy: To evaluate RTT computation accuracy, we use a single QP on an otherwise uncongested network to transfer 64KB verbs at line-rate. We then look at the RTTs that are computed by RoGUE for the different transports. As an oracle, we issue zero-byte signaled WRITE verbs over RC. Because the network is uncongested, any differences between the computed and WRITE(0) RTTs are due to measurement error. Table 1 lists the 10th, median, and 90th percentile results.

We find that 100% of WRITE(0) measurements return 2μs. For READ verbs over RC, our RTT calculation overestimates median RTT by 2μs. The higher median RTT can be accounted for by the extra time taken for PCI transactions to copy data to and from memory on both ends for the READ operation, about 1μs each. WRITE(0) has no data, and does not pay this cost. The variance occurs because RoGUE maintains queues of at most 2 batches (128KB or 100μs of data). When the client program runs immediately in response to a completing signal, it can keep the queue full and use accurate RNIC timestamps t_{comp_s} to calculate the RTT. But, if the client program is delayed from running, RoGUE instead reads an RNIC timestamp t_{enc_s} before enqueuing data. This adds both the delay of extra PCI transactions to read the timestamp, and also some scheduling delay if RoGUE gets preempted before enqueuing the next verb.

With UC, we see higher RTT measurements but less variation. The higher estimate is again due to extra PCI transactions - to read/write the data at client/server respectively, but also due to two additional transactions to pass the immediate data to and from the RoGUE library. There is less jitter because delays in scheduling the library are already accounted for when the library reads t_{rep_s} just before enqueuing its response.

Impact of inaccurate samples: We measure how robust our RTT samples need to be to provide effective congestion control. We conduct an 8$\rightarrow$1 incast. Each flow sends 128MB to a single server using RC (our results for UC were similar). We add random noise to the RTT samples picked uniformly from the range $[0, x]\mu$s, where $x = 0, 25, 50, 75, 100$ (similar to [20]). Figure 8 shows the aggregate throughput of all incast flows as a function of x. We see that an average noise of 50μs causes visible degradation in throughput; thus, RoGUE requires reasonably accurate timestamps and completion events. It also suggests that RoGUE can tolerate RTT samples of up to 50μs noise, well above typical OS and RNIC noise.

5.5 Congestion Response Efficacy

A key concern for our congestion control scheme is that our batch sizes are large and it pauses RTT sampling for $160KB$ (3 batches) after adjusting a rate limiter. This may make RoGUE slow to react to congestion, and hurt latency and throughput. We measure the instantaneous throughput and end-to-end latency to show the efficiency of RoGUE's reaction to congestion.

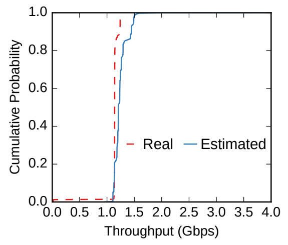


Figure 10: Real vs estimated throughput.

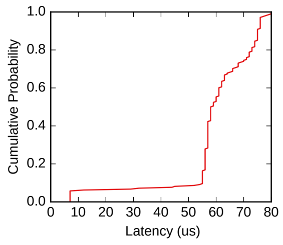


Figure 11: Latency in Parking Lot topology.

Incast: We first experiment with an 8-to-1 incast. Figure 10 plots a CDF of all flows' instantaneous throughput averaged at 1s intervals and the sending rate (*cwnd*/RTT) estimated by RoGUE, which is computed every congestion window update to verify accuracy of the estimated throughput. We see that the estimated throughput of RoGUE matches well with the real throughput. The tail 10th percentile throughput is 1.12Gbps, which happens at the slow start stage. The median of instantaneous throughputs is 1.20Gbps, which is almost identical with the ideal throughput, $1.21Gbps$ in $8 \rightarrow 1$ incast [2]. It indicates that RoGUE's congestion control can estimate the flow sending rate accurately and converges to a fixed point as we see that the estimated throughput does not have big variance in Figure 10.

In addition, both the estimated throughput and real throughput result in a near-perfect bandwidth allocation across the 8 flows. This shows that RoGUE's batch-driven design has a negligible impact and achieves both fair sharing of bandwidth and maintains a stable and evenly shared throughput.

Arriving/departing flows: Here, we start with one long-running flow being sent in isolation, and then add/remove up to 4 additional flows, one at a time. We measure the instantaneous throughput of all the five flows averaged over 1s intervals for RoGUE in a 10Gbps

[2]Note that the maximum throughput for RoCE is less than $10Gbps$ due to the RoCEv2 header added to form packets.

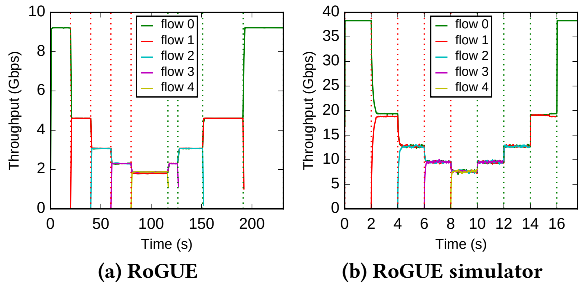


Figure 12: The instantaneous throughput (1s-average, a; 0.1s-average, b) of multiple RoGUE flows from different clients sharing a congested link to a single server. The red lines indicate the arrival of a new flow and the green lines indicate the completion of a flow.

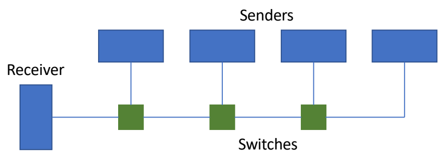


Figure 13: Multi-bottleneck (parking lot) topology.

network and at 0.1s intervals for the RoGUE simulator in a 40Gbps network.

Figure 12a shows the throughput of all RoGUE flows over time. All the five flows are able to achieve their fair share of a bottleneck link, respond quickly to changes in congestion, and converge to a new bandwidth share which is very stable.

The simulation results in Figure 12b verifies the efficiency of RoGUE, i.e., it has quick response and fairness on a high-speed network, and that it is stable even at short time scales.

Parking Lot: In addition, we also run an experiment with a multi-bottleneck network topology as shown in Figure 13. In this experiment, we start a $4 \rightarrow 1$ incast and each flow traverses a different number of bottleneck links, i.e., 1, 2, or 3 congested links. RoGUE has an average per-flow throughput of 2.27Gbps, with Jain's fairness index of 0.995. It indicates that our congestion control can work well in mutilple bottleneck scenerio. Figure 11 plots a CDF of latency measurement taken via a periodic WRITE(0) request/response on a QP from a seperate server to the incast server. Note that this latency includes both switch queuing and receiver delay. The median and 90^{th} percentile RTT are 58μs and 75μs, respectively. We saw a small variation in the CDF figure, which is expected because the RoGUE's congestion control requires enough queuing in the network to adjust the congestion window as well as the rate limiter. However, the latency is still below 80μs, which indicates that the RoGUE sets the rate limiter properly to avoid the burstiness in the network, and furthermore that RoGUE's Vegas-based congestion avoidance scheme is effective at keeping queues small.

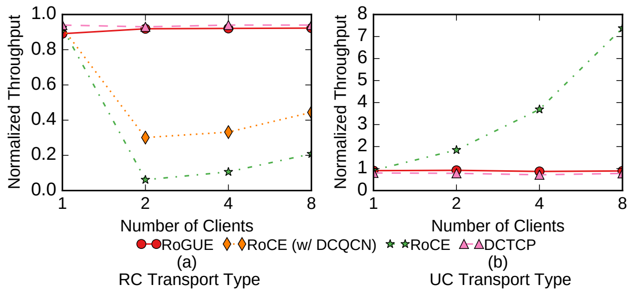


Figure 14: The 10^{th} percentile throughput of large incasts in the RC/UC transport type.

5.6 Benchmark Workloads

We study two workloads that are representative of application traffic patterns one may observe in cloud data centers where RDMA is used. We use these benchmarks to compare RoGUE against alternatives.

N→1 incasts: Here, N servers simultaneously send data to 1 client. This reflects several common data center scenarios such as disk recovery in cloud storage, where a failed disk is recovered by reading data from multiple other disks, or the aggregation stage of a partition-aggregate workload that forms the basis of search engines. We use 128MB segments.

As stragglers often determine the above applications' performance, we are primarily concerned with the 10th %-ile of the per-flow throughput distribution (tail throughput). The tail throughput being close to fair share indicates that RoGUE offers good throughput and fairness.

RC transport: Figure 14a shows the tail throughput of different transports normalized to each flow's fair share of the network (10Gbps / # Clients). When we tried RoCE without PFC, many connections failed due to lost packets, and those that do not, suffered from frequent retries that hurt throughput. RoCE with DCQCN congestion control improves the situation, but tail throughput is still less than 50% of fair share because of our switch's RED implementation, showing that DCQCN may not be usable in some networks. DCTCP is compatible with our switch and is able to provide near-perfect fairness. RoGUE achieves more than 90% of it share. We note that RoGUE can not ramp up quickly in a single client case. This is because our workloads use 128 MB segments, and slow start coupled with slow-to-respond hardware rate limiters cannot reach line rate quickly (well before segment completion).

UC transport: Figure 14(b) shows the tail throughput when there are incasts using UC WRITEs. We repeat the DCTCP results for comparison purposes, and omit RoCE with DCQCN, as it does not support the UC transport. Most importantly, we observe that, as PFC is unavailable, RoCE is unable to control sending rate. Every client sends at line rate, which overwhelms the switch and leads to high packet loss.

These results show that RoGUE can provide the same throughput as DCTCP even when used with UC. The 10th %-ile throughput results are within 5% of the fair share in all cases. Despite the inability to detect or retransmit dropped packets, goodput is near 100%.

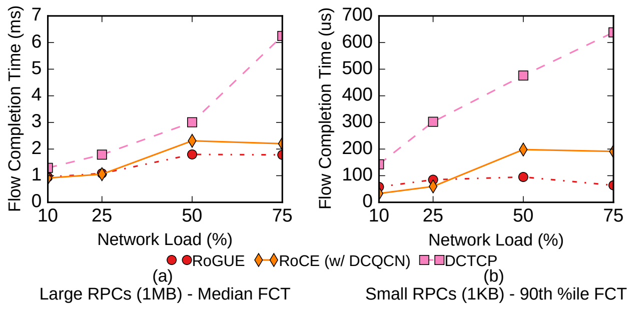


Figure 15: Comparing different network transports for a storage application's user request-response traffic.

Many pairs: This emulates the request-response traffic of an application co-located with storage. Each of 16 hosts generates 1MB flows for random destinations; the flows' inter-arrival times are sampled from the exponential distribution. We chose different inter-arrival times to vary the expected load on the network. To understand the impact on short latency-sensitive requests/responses, we also send a short 1KB message to a random server once every ten 1MB messages.

Figures 15a and 15b respectively show the median large flow completion time and 90th percentile short flow completion time (FCT). The former reflects throughput, which matters for large flows, and the latter reflects worst case response latency, which matters for short flows. In both cases, we see that RoGUE performance is consistent across network loads. For small RPCs, RoGUE provides better completion times than DCTCP because it is consistently able to provide lower latencies when there is congestion. Similarly, one reason that RoGUE can provide better flow completion times for small RPCs than RoCE is because RoGUE implements slow start. In RoCE, a new large transfer starts transferring data at line rate causing congestion that impacts the completion time of the small transfers. In large RPCs, DCTCP sees worse flow completion times because it uses larger congestion windows to compensate for its higher RTT.

5.7 TIMELY and DCQCN

We compare RoGUE with state-of-the-art RDMA congestion control schemes, i.e., TIMELY and DCQCN. TIMELY is also an RTT-based congestion control scheme. We look at convergence and fairness, which reflects the efficiency of a congestion control scheme, and the dependence on PFC. As noted in Section 4, we use a simulator for these results because a real TIMELY implementation is not available. Furthermore, DCQCN is compatible with our simulated switches.

Convergence: To demonstrate convergence and fairness, using the parking lot topology we start 4 flows simultaneously and report the aggregate throughput every 0.1s for each flow, as shown in Figure 16. We can see that RoGUE converges quickly and stays at the fixed point, i.e., 9.4*Gbps*. This is because RoGUE can estimate network RTT accurately, uses additive increase/decrease to do congestion avoidance and uses rate limits to avoid packet bursts into the network.

Figure 16(b) shows that TIMELY has a big variance and does not converge. This occurs because TIMELY's congestion avoidance does not have a fixed point as noted in [32]. Figure 16(c) shows that DCQCN has a small jitter, but can roughly converge to a fixed point.

Also, each flow in RoGUE evenly shares the full line rate with average per-flow throughput of 9.55Gbps and Jain's fairness index of 0.999 . The average per-flow throughput of TIMELY is 8.76Gbps (fairness index = 0.992), and for DCQCN the throughput is 9.74Gbps (fairness index = 0.999).

Use of PFC: PFC pause frames used by RoCE to provide lossless networking can be risky in a datacenter. While both TIMELY and DCQCN largely manage congestion, we measure whether and when TIMELY and DCQCN still depend on PFC pause frames for congestion control. Ideally, switch queues should stay short so that pause frames are not triggered.

In this experiment, we overflowed the switch buffer by initiating 32 flows, simultaneously, as it might happen at the start of a parallel job retrieving data. We measure whether pause frames are generated, and for how long. With RoGUE, there are no pause frames due to its slow start. In contrast, both TIMELY and DCQCN require pause frames when they first start transmitting, as they do not have slow start. For TIMELY, pause frames are sent for 1563μs and for DCQCN 620μs. Thus, while the other protocols in general avoid congestion leading to pause frames, under burst behavior they may still occur. In contrast, RoGUE does not have persistent queue build-up or packet loss, even with persistent congestion.

6 OTHER RELATED WORK

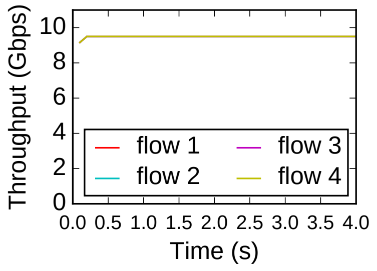
Our work draws on the long history of congestion-control mechanisms, many of which were mentioned previously. Here, we focus on closely related work.

TIMELY: There are several important differences between RoGUE and TIMELY [20]. While TIMELY is rate based, RoGUE is congestion-window based. Using a congestion window helps RoGUE avoid congestion collapse by adhering to packet conservation, applied to batches. TIMELY also relies on PFC and assumes RC, while RoGUE must tolerate congestion drops and also supports UC. Finally, we develop different custom RTT estimators; for RC, TIMELY's RTT estimation approach does not apply to our setting because we enqueue multiple batches to avoid RNIC starvation.

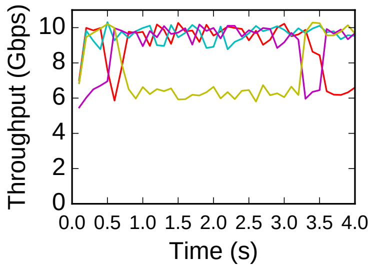
Congestion Control Algorithm Limitations: Zhu *et al.* [32] find that both DCQCN with RED/ECN and delay-based congestion control algorithms (RoGUE and TIMELY) suffer from convergence and fairness problems. We believe these findings highlight the continued need for congestion control evolution.

iWARP: RoGUE targets RoCE because RoCE RNICs are the most commonly deployed RNICs in datacenter networks. However, iWARP [25] NICs are available and implement the TCP protocol on the NIC, and can in theory be used instead of RoGUE. However, iWARP suffers from the many known problems with TCP Offload Engines (TOEs) [22, 27]. Architecturally, we believe that RoGUE's approach of offloading expensive messaging operations to hardware is better than iWARP's approach of offloading congestion control logic.

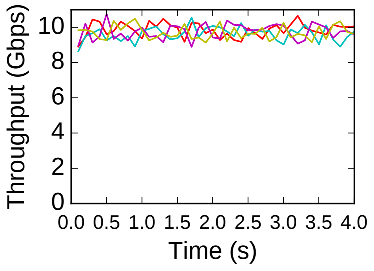
IRN: This work [21] tries to get rid of PFC by implementing selective ACK (SACK) in the hardware RNIC for faster recovery, and by limiting the number of outstanding packets to the bandwidth-delay



(a) RoGUE



(b) TIMELY



(c) DCQCN

Figure 16: Convergence and Fairness.

product of the network. But, it still relies on a software congestion control protocol such as TIMELY or DCQCN, and thus could be used with RoGUE.

7 CONCLUSIONS

Reliance on PFC remains a major hurdle in the adoption of RDMA over Ethernet (RoCE). While PFC is essential for congestion control, it is susceptible to serious safety problems. We show that it is possible to support effective congestion control for RoCE without PFC. RoGUE uses novel RTT estimators to determine congestion, large segment transfers to lower CPU utilization, a congestion window to clock data, and rate limiters to smooth packet bursts. Our evaluation of a full RoGUE implementation over a real testbed shows that these mechanisms help RoGUE offer effective congestion control at low cost: its throughput matches that of state-of-the-art datacenter congestion control, yet its latency, completion times, and CPU utilization are substantially lower.

Acknowledgments

We would like to thank the anonymous reviewers for their thoughtful feedback. Yanfang Le, Brent Stephens, Arjun Singhvi, Aditya Akella and Michael Swift are supported in part by NSF grants CNS-1330308 and CNS-1717039.

REFERENCES

[1] Lawrence S. Brakmo, Sean W. O'Malley, and Larry L. Peterson. 1994. TCP Vegas: New Techniques for Congestion Detection and Avoidance. In *Proceedings of the Conference on Communications Architectures, Protocols and Applications (SIGCOMM '94)*.
[2] Craig Carlson. 2009. IEEE 802.1: 802.1Qaz - Enhanced Transmission Selection. http://www.ieee802.org/1/pages/802.1az.html. (2009).
[3] CloudLab [n. d.]. CloudLab. http://cloudlab.us/. ([n. d.]).
[4] William J. Dally and Charles L. Seitz. 1987. Deadlock-Free Message Routing in Multiprocessor Interconnection Networks. *IEEE Trans. Comput.* C-36, 5 (May 1987), 547–553.
[5] Data Center Bridging Task Group. [n. d.]. http://www.ieee802.org/1/pages/dcbridges.html. ([n. d.]).
[6] Claudio DeSanti. 2009. IEEE 802.1: 802.1Qbb - Priority-based Flow Control. http://www.ieee802.org/1/pages/802.1bb.html. (2009).
[7] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. 2014. FaRM: Fast Remote Memory. In *NSDI*. USENIX.
[8] Norm Finn. 2008. IEEE 802.1: 802.1Qau - Congestion Notification. http://www.ieee802.org/1/pages/802.1au.html. (2008).
[9] Sally Floyd and Van Jacobson. 1993. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Trans. Netw.* (1993).
[10] Alan Ford, Costin Raiciu, Mark J. Handley, and Olivier Bonaventure. 2013. TCP Extensions for Multipath Operation with Multiple Addresses. RFC 6824. (Jan. 2013).
[11] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. 2016. RDMA over Commodity Ethernet at Scale. In *SIGCOMM*. ACM, New York, NY, USA. https://doi.org/10.1145/2934872.2934908
[12] Keqiang He, Eric Rozner, Kanak Agarwal, Yu (Jason) Gu, Wes Felter, John Carter, and Aditya Akella. 2016. AC/DC TCP: Virtual Congestion Control Enforcement for Datacenter Networks. In *SIGCOMM*.
[13] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association.
[14] Josh Simmons. 2000. RDMA in the Cloud: Enabling high-bandwidth, low-latency communication in virtual environments for HPC. (2000). https://octo.vmware.com/wp-content/uploads/2014/11/ndm2014-sc14-simons3.pdf
[15] Glenn Judd. 2015. Attaining the Promise and Avoiding the Pitfalls of TCP in the Datacenter. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation (NSDI'15)*. USENIX Association, Berkeley, CA, USA, 145–157. http://dl.acm.org/citation.cfm?id=2789770.2789781
[16] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2014. Using RDMA Efficiently for Key-Value Services. In *SIGCOMM*. Chicago, IL.
[17] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. Design Guidelines for High Performance RDMA Systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association.
[18] Mellanox Technologies. [n. d.]. ConnectX-3 Pro. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_ConnectX-3_Pro_Card_VPI.pdf. ([n. d.]).
[19] Mellanox Technologies. [n. d.]. ConnectX-4 VPI. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_ConnectX-4_VPI_Card.pdf. ([n. d.]).
[20] Radhika Mittal, Terry 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. In *SIGCOMM*.
[21] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. 2018. Revisiting Network Support for RDMA. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*.
[22] Jeffrey C. Mogul. 2003. TCP Offload is a Dumb Idea Whose Time Has Come. In *HotOS*. USENIX Association, 5–5.
[23] OpenFabrics Alliance. [n. d.]. OFED Overview. https://www.openfabrics.org/index.php/openfabrics-software.html. ([n. d.]).
[24] Marius Poke and Torsten Hoefler. 2015. DARE: High-Performance State Machine Replication on RDMA Networks. In *HPDC*.
[25] RDMA Consortium. [n. d.]. Architectural Specifications for RDMA over TCP/IP. http://rdmaconsortium.org/. ([n. d.]).
[26] Brent Stephens, Alan L. Cox, Ankit Singla, John Carter, Colin Dixon, and Wesley Felter. 2014. Practical DCB for Improved Data Center Networks. In *INFOCOM*.
[27] The Linux Foundation. [n. d.]. toe. http://www.linuxfoundation.org/collaborate/workgroups/networking/toe. ([n. d.]).
[28] Sven Ulland. 2011. Kernel panic/crash, bnx2 flow control flooding and network outages. Linux-PowerEdge – Linux on Dell PowerEdge Servers discussion http://lists.us.dell.com/pipermail/linux-poweredge/2011-October/045485.html. (2011).
[29] Keith Winstein and Hari Balakrishnan. 2013. TCP ex Machina: Computer-Generated Congestion Control. In *SIGCOMM*. Hong Kong,.
[30] Yibo Zhu. [n. d.]. ns3-rdma. https://github.com/bobzhuyb/ns3-rdma/tree/timely. ([n. d.]).
[31] 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. In *SIGCOMM*. ACM. http://research.microsoft.com/apps/pubs/default.aspx?id=252307
[32] Yibo Zhu, Monia Ghobadi, Vishal Misra, and Jitendra Padhye. 2016. ECN or Delay: Lessons Learnt from Analysis of DCQCN and TIMELY. In *Proceedings of 2016 ACM Conference on Emerging network experiment and technology (CoNEXT 2016)*.