Empowering Azure Storage with RDMA

Microsoft

MSR-TR-2023-13

Abstract

Given the wide adoption of disaggregated storage in public clouds, networking is the key to enabling high performance and high reliability in a cloud storage service. In Azure, we choose Remote Direct Memory Access (RDMA) as our transport and aim to enable it for both storage frontend traffic (between compute virtual machines and storage clusters) and backend traffic (within a storage cluster) to fully realize its benefits. As compute and storage clusters may be located in different datacenters within an Azure region, we need to support RDMA at regional scale.

This work presents our experience in deploying intra-region RDMA to support storage workloads in Azure. The high complexity and heterogeneity of our infrastructure bring a series of new challenges, such as the problem of interoperability between different types of RDMA network interface cards. We have made several changes to our network infrastructure to address these challenges. Today, around 70% of traffic in Azure is RDMA and intra-region RDMA is supported in all Azure public regions. RDMA helps us achieve significant disk I/O performance improvements and CPU core savings.

1 Introduction

High performance and highly reliable storage is one of the most fundamental services in public clouds. In recent years, we have witnessed significant improvements in storage media and technologies [73] and customers also desire similar performance in the cloud. Given the wide adoption of disaggregated storage in the cloud [35, 46], the network interconnecting compute and storage clusters becomes a key performance bottleneck for cloud storage. Despite the sufficient bandwidth capacity provided by Clos-based network fabrics [25, 48], the legacy TCP/IP stack suffers from high processing delay, low single-core throughput, and high CPU consumption, thus making it ill-suited for this scenario.

Given these limitations, Remote Direct Memory Access (RDMA) offers a promising solution. By offloading the network stack to the network interface card (NIC) hardware, RDMA achieves ultra-low processing latency and high throughput with near zero CPU overhead. In addition to performance improvements, RDMA also reduces the number of CPU cores reserved on each server for network stack processing. These saved CPU cores can then be sold as customer virtual machines (VMs) or used for application processing.

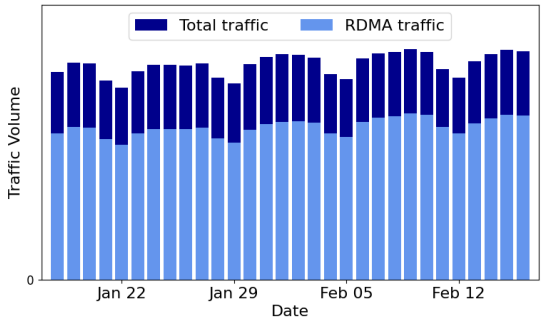

Figure 1: Traffic statistics of all Azure public regions between January 18 and February 16, 2023. Traffic was measured by collecting switch counters of server-facing ports on all Top of Rack (ToR) switches. Around 70% of traffic was RDMA.

To fully utilize the benefits of RDMA, we aim to enable it for *both* storage frontend traffic (between compute VMs and storage clusters) and backend traffic (within a storage cluster). This is different from previous work [46] that targets RDMA only for the storage backend. In Azure, due to capacity issues, corresponding compute and storage clusters may be located in different datacenters within a region. This imposes a requirement that our storage workloads rely on support for RDMA at regional scale.

In this paper, we summarize our experience in deploying intra-region RDMA to support Azure storage workloads. Compared to previous RDMA deployments [46, 50], intra-region RDMA deployment introduces many new challenges due to high complexity and heterogeneity within Azure re-

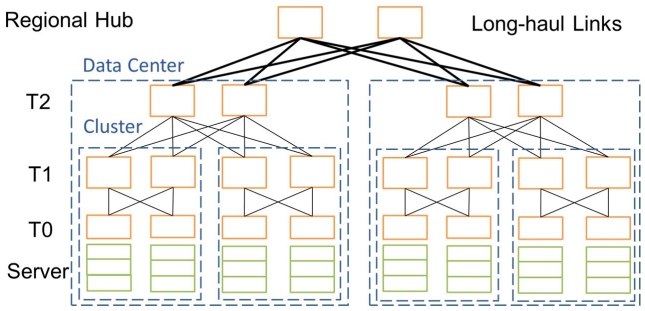

Figure 2: The network architecture of an Azure region.

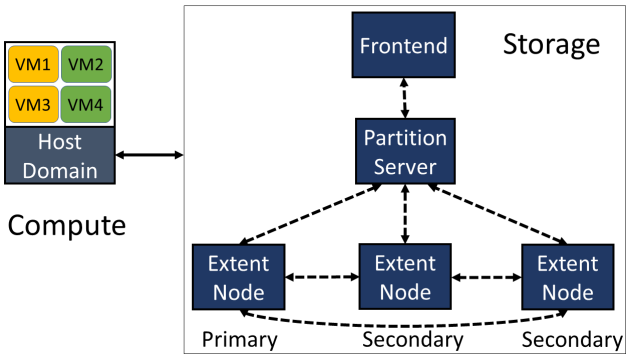

Figure 3: High-level architecture of Azure storage.

gions. As Azure infrastructure keeps evolving incrementally, different clusters may be deployed with different RDMA NICs. While all the NICs support DCQCN [112], their implementations are very different. This results in many undesirable behaviors when different NICs communicate with each other. Similarly, heterogeneous switch software and hardware from multiple vendors significantly increase our operational effort. In addition, long-haul cables interconnecting datacenters cause large propagation delays and large round-trip time (RTT) variations within a region. This brings new challenges to congestion control.

We have made several changes to our network infrastructure, from application layer protocols to link layer flow control, to safely enable intra-region RDMA for Azure storage traffic. We developed new RDMA-based storage protocols with many optimizations and failover support, and seamlessly integrated them into the legacy storage stack (§4). We built RDMA Estats to monitor the status of the host network stack (§5). We leveraged SONiC to enforce a unified software stack across different switch platforms (§6). We updated firmware of NICs to unify their DCQCN behaviors and used the combination of Priority-based Flow Control (PFC) and DCQCN to achieve high throughput, low latency and near zero packet losses (§7).

In 2018, we started to enable RDMA for storage backend traffic. In 2019, we started to enable RDMA to serve customer frontend traffic. Figure 1 gives traffic statistics of all Azure public regions between January 18 and February 16, 2023. As of February 2023, around 70% of traffic in Azure was RDMA and intra-region RDMA was supported in all Azure public regions. RDMA helps us achieve significant disk I/O performance improvements and CPU core savings.

2 Background

In this section, we first present background on Azure's network and storage architecture. Then, we introduce the motivation for and challenges to enabling intra-region RDMA.

2.1 Network Architecture of an Azure Region

In cloud computing, a region [2,5,8] is a group of datacenters deployed within a latency-defined perimeter. Figure 2 shows the simplified topology of an Azure region. The servers within a region are connected through an Ethernet-based Clos network with four tiers of switches[1]: tier 0 (T0), tier 1 (T1), tier 2 (T2) and regional hub (RH). We use external BGP (eBGP) for routing and equal-cost multi-path (ECMP) for load balancing. We deploy the following four types of units.

- Rack: a T0 switch and the servers connected to it.
- Cluster: a set of racks connected to the same set of T1 switches.
- Datacenter: a set of clusters connected to the same set of T2 switches.
- Region: datacenters connected to the same set of RH switches. In contrast with short links (several to hundreds of meters) in datacenters [50], T2 and RH switches are connected by *long-haul* links whose lengths can be as long as tens of kilometers.

There are two thing to notice about this architecture. First, due to long-haul links between T2 and RH, the base round-trip time (RTT) varies from a few microseconds within a datacenter to as large as 2 milliseconds within a region. Second, we use two types of switches: pizza box switches for T0 and T1, and chassis switches for T2 and RH. The pizza box switch, which has been widely studied in the research community, typically has a single switch ASIC with shallow packet buffers [31]. In contrast, chassis switches are built using multiple switch ASICs with deep packet buffers based on the Virtual Output Queue (VoQ) architecture [3,6].

2.2 High Level Architecture of Azure Storage

In Azure, we disaggregate compute and storage resources for cost savings and auto-scaling. There are two main types of

[1]In this paper, we use switch to denote the layer 3 switch which can perform IP routing. We use the terms switch and router interchangeably.

clusters in Azure: compute and storage. VMs are created in compute clusters but the actual storage of Virtual Hard Disks (VHDs) resides in storage clusters.

Figure 3 shows the high-level architecture of Azure storage [35]. Azure storage has three layers: the frontend layer, the partition layer, and the stream layer. The stream layer is an append-only distributed file system. It stores bits on the disk and replicates them for durability, but it does not understand higher level storage abstractions, e.g., Blobs, Tables and VHDs. The partition layer understands different storage abstractions, manages partitions of all the data objects in a storage cluster, and stores object data on top of the stream layer. The daemon processes of the partition layer and the stream layer are called the Partition Server (PS) and the Extent Node (EN), respectively. PS and EN are co-located on each storage server. The frontend (FE) layer consists of a set of servers that authenticate and forward incoming requests to corresponding PSs. In some cases, FE servers can also directly access the stream layer for efficiency.

When a VM wants to write to its disks, the disk driver running in the host domain of the compute server issues I/O requests to the corresponding storage cluster. The FE or PS parses and validates the request, and generates requests to corresponding ENs in the stream layer to write the data. At the stream layer, a file is essentially an ordered list of large storage chunks called *"extents"*. To write a file, data is appended to the end of an active *extent*, which is replicated three times in the storage cluster for durability. Only after receiving successful responses from all the ENs, the FE or PS sends the final response back to the disk driver. In contrast, disk reads are different. The FE or PS reads data from any EN replica and sends the response back to the disk driver.

In addition to user-facing workloads, there are also many background workloads in the storage clusters, e.g., garbage collection and erasure coding [57]. We classify our storage traffic into two categories: frontend (between compute and storage servers, e.g., VHD write and read requests) and backend (between storage servers, e.g., replication and disk reconstruction). Our storage traffic has incast-like characteristics. The most typical example is data reconstruction, which is implemented in the stream layer [57]. The stream layer erasure codes a sealed extent to several fragments, and then sends encoded fragments to different servers to store. When the user wants to read a fragment which is unavailable due to a failure, the stream layer will read the other fragments from multiple storage servers to reconstruct the target fragment.

2.3 Motivation for Intra-Region RDMA

Storage technology has improved significantly in recent years. For example, Non-Volatile Memory Express (NVMe) Solid-State Drives (SSDs) can provide tens of Gbps of throughput with request latencies in the hundreds of microseconds [105]. Many customers demand similar performance in the cloud. High performance cloud storage solutions [1, 4] impose stringent performance requirements to the underlying network due to the disaggregated and distributed storage architecture (§2.2). While datacenter networks generally provide sufficient bandwidth capacity, the legacy TCP/IP stack in the OS kernel becomes a performance bottleneck due to its high processing latency and low single-core throughput. What is worse, the performance of the legacy TCP/IP stack also depends on OS scheduling. To provide predictable storage performance, we must reserve enough CPU cores on both compute and storage nodes for the TCP/IP stack to process peak storage workloads. Burning CPU cores takes away the processing power that could otherwise be sold as customer VMs, thus increasing the overall cost of providing cloud services.

Given these limitations, RDMA offers a promising solution. By offloading the network stack to the NIC hardware, RDMA achieves predictable low processing latency (a few microseconds) and high throughput (line rate for a single flow) with near zero CPU overhead. In addition to its performance benefits, RDMA also reduces the number of CPU cores reserved on each server for network stack processing. These saved CPU cores can then be sold as customer VMs or used for storage request processing.

To fully achieve the benefits of RDMA, we must enable RDMA for both storage frontend traffic and backend traffic. Enabling RDMA for backend traffic is relatively easy because almost all the backend traffic stays within a storage cluster. In contrast, frontend traffic crosses different clusters within a region. Even though we try to co-locate corresponding compute and storage clusters to minimize latency, sometimes they may still end up located in different datacenters within a region due to capacity issues. This imposes the requirement that our storage workloads rely on support for RDMA at regional scale.

2.4 Challenges

We faced many challenges when enabling intra-region RDMA because our design was limited by many practical constraints.

Practical considerations: We aimed to enable intra-region RDMA over the legacy infrastructure. While we had some flexibility to reconfigure and upgrade software stacks, e.g., the NIC driver, the switch OS, and the storage stack, it was *operationally infeasible* to replace the underlying hardware, e.g., the NICs and switches. Hence, we adopted RDMA over commodity Ethernet v2 (RoCEv2) [29] to keep compatibility with our IP-routed networks (§2.1). Before starting this project, we had deployed a significant number of our first generation RDMA NICs, which implement go-back-N retransmission in the NIC firmware with limited processing capacity. Our measurements showed that it took hundreds of microseconds to recover a lost packet, which was even worse than the TCP/IP software stack. Given such a large performance degradation, we made the decision to adopt Priority-based Flow Control

(PFC) [60] to eliminate packet losses due to congestion.

Challenges: Before this project, we had deployed RDMA in some clusters to support Bing services [50], and we learnt several lessons from this deployment. Compared to intra-cluster RDMA deployments [46, 50], intra-region RDMA deployments introduce many new challenges due to the high complexity and heterogeneity of the infrastructure.

- Heterogeneous NICs: Cloud infrastructure keeps evolving incrementally, often one cluster or one rack at a time with the latest generation of server hardware [91]. Different clusters within a region may have different NICs. We have deployed three generations of commodity RDMA NICs from a popular NIC vendor: Gen1, Gen2 and Gen3. Each NIC generation has a different implementation of DCQCN. This results in many undesired interactions when different NIC generations communicate with each other.
- Heterogeneous switches: Similar to server infrastructure, we keep deploying new switches to reduce costs and increase the bandwidth capacity. We have deployed many switch ASICs and multiple switch OSes from different vendors. However, this has increased our operational effort significantly because many aspects are vendor specific, for example, buffer architectures, sizes, allocation mechanisms, monitoring and configuration, etc.
- Heterogeneous latency: As shown in §2.1, there are large RTT variations from several microseconds to 2 milliseconds within a region, due to long-haul links between T2 and RH. Hence, RTT fairness re-emerges as a key challenge. In addition, the large propagation delay of long-haul links also imposes large pressure on PFC headroom [12].

Like other services in public clouds, availability, diagnosis, and serviceability are key aspects for our RDMA storage system. To achieve high availability, we always prepare for unexpected *zero-day* problems despite large investments in testing. Our system must detect performance anomalies and perform automatic failover if necessary. To understand and debug faults, we must build fine-grained telemetry systems to deliver crystal clear visibility into every component in the end-to-end path. Our system also must be serviceable: storage workloads should survive NIC driver updates and switch software updates.

3 Overview

We have made several changes to our network infrastructure, from application layer protocols to link layer flow control, to safely empower Azure storage with RDMA. We developed two RDMA-based protocols: sU-RDMA (§4.1) and sK-RDMA (§4.2), which we have seamlessly integrated into our legacy storage stack to support backend communication and frontend communication, respectively. Between the storage protocols and the NIC, we deployed a monitoring system RDMA Estats (§5), giving us visibility into the host network stack by providing an accurate breakdown of cost for each RDMA operation.

In the network, we use the combination of PFC and DCQCN [112] to achieve high throughput, low latency, and near zero losses due to congestion. DCQCN and PFC were the state-of-the-art commercial solutions when we started the project. To optimize the customer experience, we use two priorities to isolate storage frontend traffic and backend traffic. To mitigate the switch heterogeneity problem, we developed and deployed SONiC [15] to provide a unified software stack across different switch platforms (§6). To mitigate the interoperability problem of heterogeneous NICs, we updated the firmware of NICs to unify their DCQCN behaviors (§7). We carefully tuned DCQCN and switch buffer parameters to optimize performance across different scenarios.

3.1 PFC Storm Mitigation Using Watchdogs

We use PFC to prevent congestion packet losses. However, malfunctioning NICs and switches can continually send PFC pause frames in the absence of congestion [50], thus *completely* blocking the peer device for a long time. Moreover, these endless PFC pause frames can eventually propagate into the whole network, thus causing collateral damage to innocent devices. Such *endless* PFC pause frames are called a PFC storm. In contrast, normal congestion-triggered PFC pause frames only *slow down* the data transmission of the peer device through *intermittent* pauses and resumes.

To detect and mitigate PFC storms, we designed and deployed a PFC watchdog [11, 50] on every switch and bump-in-the-wire FPGA card [42] between T0 switches and servers. When the PFC watchdog detects that a queue has been in the paused state for an abnormally long duration, e.g., hundreds of milliseconds, it disables PFC and drops all the packets on this queue, thereby preventing PFC storms from propagating into the whole network.

3.2 Security

We use RDMA to empower first-party storage traffic in a trusted environment, including storage servers, the host domain of compute servers, switches and links. Therefore we are secure against issues described in [69, 94, 104, 109].

4 Storage Protocols over RDMA

In this section, we introduce two storage protocols built on top of RDMA Reliable Connections (RC): sU-RDMA and sK-RDMA. Both protocols aim to optimize performance while keeping good compatibility with legacy software stacks.

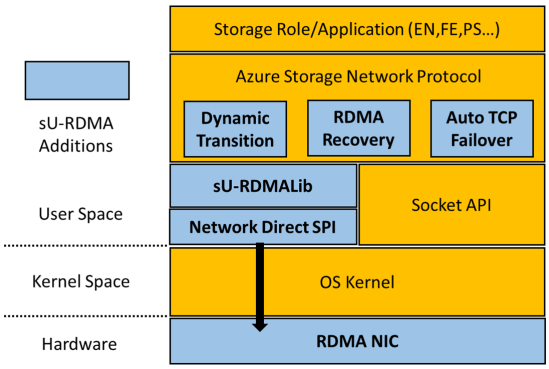

Figure 4: Azure storage backend network stack.

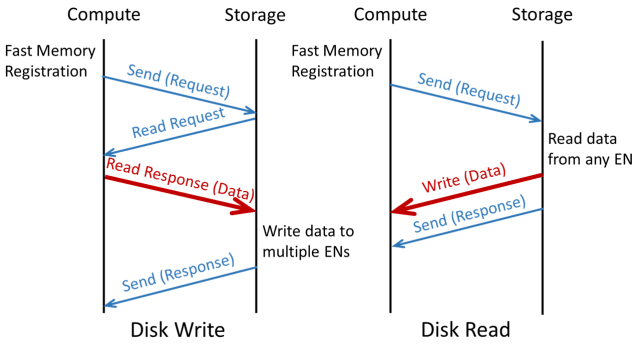

Figure 5: sK-RDMA's data flow. We use blue arrows and red arrows to represent control messages and data massages, respectively. Arrow width represents data size.

4.1 sU-RDMA

sU-RDMA [87] is used for storage backend (storage to storage) communication. Figure 4 shows the architecture of our storage backend network stack with the sU-RDMA modules highlighted. The Azure Storage Network Protocol is an RPC protocol directly used by applications to send request and response objects. It leverages socket APIs to implement connection management, sending and receiving messages.

To simplify RDMA integration with storage stack, we built sU-RDMALib, a user space library that exposes socket-like byte-stream APIs to upper layers. To map socket-like APIs to RDMA operations, sU-RDMALib needs to handle the following challenges:

- When the RDMA application cannot directly write into an existing memory regions (MR), it must either register the application buffer as a new MR or copy its data into an existing MR. Both options can introduce large latency penalties and we should minimize these overhead.
- If we use RDMA `Send` and `Receive`, the receiver must pre-post enough `Receive` requests.
- The RDMA sender and receiver must be in agreement on the size of data being transferred.

To reduce memory registrations, which are especially expensive for small messages [44], sU-RDMALib maintains a common buffer pool of pre-registered memory shared across multiple connections. sU-RDMALib also provides APIs to allow applications to request and release registered buffers. To avoid Memory Translation Table (MTT) cache misses on the NIC [50], sU-RDMALib allocates large memory slabs from the kernel and registers memory over these slabs. This buffer pool can also autoscale based on runtime usage. To avoid overwhelming the receiver, sU-RDMALib implements a receiver-driven credit-based flow control where credits represent the resources (e.g., available buffers and posted `Receive` requests) allocated by the receiver. The receiver sends credit update messages back to the sender regularly. When we started designing sU-RDMALib, we did consider using RDMA `Send` and `Receive` with a fixed buffer size S for each `Send`/`Receive` request to transfer data. However, this design causes a dilemma. If we use a large S, we may waste much memory space because a `Send` request fully uses the receive buffer of the `Receive` request, regardless of its actual message size. In contrast, a small S causes large data fragmentation overhead. Hence, sU-RDMALib uses three transfer modes based on the message size [87].

- Small messages: Data is transferred using RDMA `Send` and `Receive`.
- Medium messages: The sender posts a RDMA `Write` request to transfer data, and a `Send` request with "Write Done" to notify the receiver.
- Large messages: The sender first posts a RDMA `Send` request carrying the description of the local data buffer to the receiver. Then the receiver posts a `Read` request to *pull* the data. Finally, the receiver posts a `Send` request with "Read Done" to notify the sender.

On top of sU-RDMALib, we built modules to enable dynamic transitions between TCP and RDMA, which is critical for failover and recovery. The transition process is gradual. We periodically close a small portion of all connections and establish new connections using the desired transport.

Unlike TCP, RDMA uses rate based congestion control [112] without tracking the number of in-flight packets (the window size). Hence, RDMA tends to inject excessive in-flight packets, thus triggering PFC. To mitigate this, we implemented a static flow control mechanism in the Azure Storage Network Protocol by dividing a message into fixed-sized chunks and only allowing a single in-flight chunk for each connection. Chunking can significantly improve performance under high-degree incast with negligible CPU overhead.

4.2 sK-RDMA

sK-RDMA is used for storage frontend (compute to storage) communication. In contrast with sU-RDMA which runs

RDMA in user space, sK-RDMA runs RDMA in kernel space. This enables the disk driver, which runs in kernel space in the host domain of compute servers, to directly use sK-RDMA to issue network I/O requests. sK-RDMA leverages and extends Server Message Block (SMB) Direct [14] which provides socket-like kernel-mode RDMA interfaces. Similar to sU-RDMA, sK-RDMA also provides credit-based flow control and dynamic transition between RDMA and TCP.

Figure 5 shows sK-RDMA's data flow for reading and writing disks. The compute server first posts a Fast Memory Registration (`FMR`) request to register data buffers. Then it posts an RDMA `Send` request to transfer a request message to the storage server. The request carries a disk I/O command, and a description of `FMR` registered buffers available for RDMA access. According to the InfiniBand (IB) specification, the NIC should wait for the completion of the `FMR` request before processing any subsequently posted requests. Hence, the request message is actually pushed onto the wire after the memory registration. The data transfer is initiated by the storage server using RDMA `Read` or `Write`. After the data transfer, the storage server sends a response message to the compute server using RDMA `Send With Invalidate`.

To detect data corruptions, which can happen *silently* due to various software and hardware bugs along the path, both sK-RDMA and sU-RDMA implement a Cyclical Redundancy Check (CRC) on all application data. In sK-RDMA, the compute server calculates the CRC of the data for disk writes. These calculated CRCs are included in the request messages, and used by the storage server to validate the data. For disk reads, the storage server performs the CRC calculations and includes them in the response messages, and the compute server uses them to validate the data.

5 RDMA Estats

To understand and debug faults, we need fine-grained telemetry tools to capture behaviors of every component in the end-to-end path. Despite many existing tools [51, 97, 114] to diagnose switch and link faults, none of these tools gives us good visibility into the RDMA network stack at end hosts.

Inspired by diagnostic tools for TCP [79], we developed RDMA Extended Statistics (Estats) to diagnose performance problems in both the network and the host. If an RDMA application is performing poorly, RDMA Estats enables us to tell if the bottleneck is in the sender, the receiver, or the network.

To this end, RDMA Estats provides a fine-grained breakdown of latency for each RDMA operation, in addition to collecting regular counters such as bytes sent/received and number of NACKs. The requester NIC records timestamps at one or more measurement points as the work queue element (WQE) traverses the transmission pipeline. When a response (ACK or `read` response) is received, the NIC records additional timestamps at measurement points along the receive

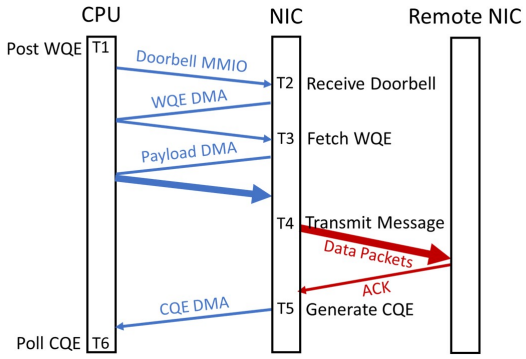

Figure 6: RDMA Estats measurement points. There are four NIC timestamps and two host timestamps. We use blue arrows and red arrows to represent PCIe transactions and network transfers, respectively. Arrow width represents data size.

pipeline (Figure 6). The following measurement points are required in any RDMA Estats implementation in Azure

T_1: WQE posting: Host processor timestamp when the WQE is posted to the submission queue.

T_5: CQE generation: NIC timestamp when the completion queue element (CQE) is generated in the NIC.

T_6: CQE polling: Host timestamp when the CQE is polled by software.

In Azure, the NIC driver reports various latencies derived from the above timestamps. For example, $T_6 - T_1$ is the operation latency seen by the RDMA consumer, while $T_5 - T_1$ is the latency seen by the NIC. A user-mode agent groups the latency samples by connection, operation type, and (success/-failure) status to create latency histograms for each group. By default, a histogram covers a one-minute interval. Each histogram's quantiles and summary statistics are fed into Azure's telemetry pipeline. As our diagnostics evolved, we added to our user-mode agent the ability to collect and upload NIC and QP state dumps during high latency events. Finally, we extended the scope of event-triggered data collection by the user-mode agent to include NIC statistics and state dumps in case of events not specific to RDMA (e.g., servicing operations that impact connectivity).

The collection of latency samples adds overhead to the WQE posting and completion processing code paths. This overhead is dominated by keeping the NIC and host time stamps synchronized. To reduce the overhead, we developed a clock synchronization procedure that attempts to minimize the frequency of reading the NIC clock registers, while maintaining low deviations.

RDMA Estats can significantly reduce the time to debug and mitigate storage performance incidents by quickly ruling out (or in) network latency. In §8.3, we share our experience in diagnosing the FMR hidden fence bug using RDMA Estats.

6 Switch Management

6.1 Overcoming Heterogeneity with SONiC

Our RDMA deployment heavily relies on the support of switches. However, heterogeneous switch ASICs and OSes from multiple vendors have brought significant challenges to network management. For example, commercial switch OSes are designed to satisfy diverse requirements of all the customers, thus leading to complex software stacks and slow feature evolution [39]. In addition, different switch ASICs provide different buffer architectures and mechanisms, thus increasing the effort to qualify and test them for Azure's RDMA deployment.

Our solutions to the above challenges were two-fold. On one hand, we worked closely with our vendors to define concrete feature requirements and test plans, and to understand their low-level implementation details. On the other hand, in collaboration with many partners, we developed and deployed an in-house cross-platform switch OS called Software for Open Networking in the Cloud (SONiC) [15]. Based on a Switch Abstraction Interface (SAI) [20], SONiC manages heterogeneous switches from multiple vendors with a simplified and unified software stack. It breaks apart monolithic switch software into multiple containerized components. Containerization provides clean isolation, improves development agility, and enables choices on a per-component basis. Network operators can customize SONiC with only the features they require, thereby creating a "lean stack".

6.2 Buffer Model and Configuration Practices of SONiC on Pizza Box Switches

SONiC provides all the features required by RDMA deployments, such as ECN marking, PFC, a PFC watchdog (§3.1) and a shared buffer model. In the interest of space, we briefly introduce the buffer model and configuration practices of SONiC on pizza box switches, which are used at T0 and T1 (§2.1). We provide a buffer configuration example in §A.

We typically allocate three buffer pools on a pizza box switch: (1) the `ingress_pool` for ingress admission control of all packets, (2) the `egress_lossy_pool` for egress admission control of lossy packets, and (3) the `egress_lossless_pool` for egress admission control of lossless packets. Note that these buffer pools and queues are not backed by separate dedicated buffers, but instead are essentially counters applied to a single physical shared buffer and used for admission control purposes. Each counter is updated only by the packets mapped to it, and the same packet can be mapped to multiple queues and pools simultaneously. For example, a lossless (lossy) packet of priority p from source port s to destination port d updates ingress queue (s, p), egress queue (d, p), `ingress_pool` and `egress_lossless_pool` (`egress_lossy_pool`). A packet is accepted only if it passes both ingress and egress admission controls. Counters increment by the size of the admitted packet, and decrement by the size of the departing packet. We use both dynamic thresholds [40] and static thresholds to limit the queue lengths.

We apply ingress admission control only to lossless traffic, and we apply egress admission control only to lossy traffic. If the switch buffer size is B, then the `ingress_pool` size must be smaller than B, reserving enough space for PFC headroom buffer (§7.1). When an ingress lossless queue hits the dynamic threshold, the queue enters the "paused" state, and the switch sends PFC pause frames to the upstream device. Future arriving packets on this ingress lossless queue use the PFC headroom buffer rather than `ingress_pool`. In contrast, for ingress lossy queues we configure a static threshold which equals to the switch buffer size B. Since ingress lossy queue lengths cannot hit the switch buffer size, lossy packets can *bypass* ingress admission control.

At egress, lossy and lossless packets are mapped to the `egress_lossy_pool` and `egress_lossless_pool`, respectively. We configure both the size of the `egress_lossless_pool` and the static thresholds for egress lossless queues to B so that lossless packets bypass egress admission control. In contrast, the size of the `egress_lossy_pool` must be no larger than the size of the `ingress_pool` because lossy packets should not use any of the PFC headroom buffer at ingress. Egress lossy queues are configured to use dynamic thresholds [40] to drop packets.

6.3 Testing RDMA Features with SONiC

We use nightly tests to track the quality of SONiC switches. In this section, we briefly introduce our methods for testing RDMA features with SONiC switches.

Software-based Tests: We leveraged the Packet Testing Framework (PTF) [10] to develop test cases for SONiC in general. PTF is mostly used for testing packet forwarding behaviors, with which testing RDMA features require additional effort.

Our testing approach is inspired by breakpoints in software debugging. To set a "breakpoint" for the switch, we first block the transmission of a switch port using SAI APIs. We then generate a series of packets destined for the blocked port and capture one or several snapshots of the switch states (e.g., buffer watermark), analogous to dumping the values of variables in software debugging. Next, we release the port and dump the received packets. We determine if the test passes by analyzing both the captured switch snapshots and the received packets. We use this approach to test buffer management mechanisms, buffer related counters, and packet schedulers.

Hardware-based Tests: While the above approach gives us good visibility into switch states and packet micro-behaviors, it cannot meet the stringent performance requirements of some tests. For example, to test PFC watchdog [50], we need to generate continuous PFC pause frames at high speed and ac-

curately control their intervals due to the small pause duration enforced by each PFC frame.

To conduct such performance-sensitive tests, we need to control traffic generation at μs or even ns timescales and have high-resolution measurement of data plane behaviors. This motivated us to build a hardware-based test system by leveraging hardware programmable traffic generators [9]. Our hardware-based system focuses on testing features like PFC, PFC watchdog, RED/ECN marking.

As of February 2023, we built 32 software test cases and 50 hardware test cases for RDMA features. The documentation and implementation of our test cases are available at [18].

7 Congestion Control

We use the combination of PFC and DCQCN to mitigate congestion. In this section, we discuss how we scale both techniques at regional scale.

7.1 Scaling PFC over Long Links

Once an ingress queue pauses the upstream device, it requires a dedicated headroom buffer to absorb in-flight packets before the PFC pause frame takes effect on the upstream device [50, 112]. The ideal PFC headroom value depends on many factors, e.g., link capacity and propagation delay [12]. The total demand on the headroom buffer for a switch is also in proportion to the number of lossless priorities[2].

To extend RDMA from cluster scale [46, 50] to regional scale, we must deal with long links between T2 and RH (tens of kilometers), and between T1 and T2 (hundreds of meters), which demand much larger PFC headroom than that of intra-cluster links. At first glance, it may seem that a T1 switch in our production environment can reserve half of the total buffer for PFC headroom and other usages. At T2 and RH, given the high port density (100s) of chassis switches and long-haul links, we need to reserve several GB of PFC headroom buffer.

To scale PFC over long links, we leverage the fact that pathological cases, e.g., all the ports are congested simultaneously, and ingress lossless queues of a port pause peers sequentially, are likely to be rare. Our solution is two-fold. First, on chassis switches at T2 and RH, we use deep packet buffers of off-chip DRAM[3] to store RDMA packets. Our analysis shows that our chassis switches in production can provide abundant DRAM buffers for PFC headroom. Second, instead of reserving PFC headroom per queue, we allocate a PFC headroom pool shared by all the ingress lossless queues on the switch. Each ingress lossless queue has a static threshold to limit its maximum usage in the headroom pool. We oversubscribe the headroom pool size with a reasonable ratio, thus leaving more shared buffer space to absorb bursts. Our production experience shows that the oversubscribed PFC headroom pool can effectively eliminate congestion losses and improve burst tolerance.

[2]For an ingress port, the worst case is that its lossless queues *sequentially* pause the peer queues, and none of its packets can be drained from the buffer.

[3]Unlike on-chip SRAM, the bandwidth of off-chip DRAM is slightly smaller than the forwarding capacity of the switch ASIC. When all the ports send and receive traffic at line rate, DRAM will suffer from packet drops.

7.2 DCQCN Interoperability Challenges

We use DCQCN [112] to control the sending rate of each queue pair (QP). DCQCN consists of three entities: the sender or reaction point (RP), the switch or congestion point (CP), and the receiver or notification point (NP). The CP performs ECN marking at the egress queue based on the RED algorithm [43]. The NP sends Congestion Notification Packets (CNPs) when it receives ECN-marked packets. The RP reduces its sending rate when it receives CNPs. Otherwise, it leverages a byte counter and a timer to increase the rate.

We deployed three generations of commodity NICs from a popular NIC vendor: Gen1, Gen2 and Gen3, for different types of clusters. While all of them support DCQCN, their implementation details differ significantly. This causes an interoperability problem when different generations of NICs communicate with each other.

DCQCN implementation differences: On Gen1, most of the DCQCN functionality, such as the NP and RP state machines, is implemented in firmware. Given the limited processing capacity of the firmware, Gen1 minimizes CNP generation through coalescing at the NP side. As described in [112], the NP generates at most one CNP in a time window for a flow, if any arriving packets within this window are ECN marked. Correspondingly, the RP reduces the sending rate upon receiving a CNP. In addition, Gen1 also has limited cache resources. Cache misses can significantly impact RDMA's performance [50, 63]. To mitigate cache misses, we increase the granularity of rate limiting on Gen1 from a single packet to a burst of packets. Burst transmissions can effectively reduce the number of active QPs in a fixed interval, thus lowering pressure on the very limited cache resources of Gen1 NICs.

In contrast, Gen2 and Gen3 have hardware-based DCQCN implementations and adopt a RP-based CNP coalescing mechanism, which is the exact opposite of the NP-based CNP coalescing used by Gen1. In Gen2 and Gen3, the NP sends a CNP for every arriving ECN-marked packet. However, the RP only cuts the sending rate for a flow at most once in a time window if it receives any CNPs within that window. It is worthwhile to note that RP-based and NP-based CNP coalescing mechanisms essentially provide the same congestion notification granularity. The rate limiting is on a per-packet granularity on Gen2 and Gen3.

Interoperability challenges: Storage frontend traffic, which crosses different clusters, may lead to communication between different generations of NICs. In this scenario, the DCQCN implementation differences cause undesirable behaviors. First, when a Gen2/Gen3 node sends traffic to a Gen1 node,

its per-packet rate limiting tends to trigger many cache misses on the Gen1 node, thus slowing down the receiver pipeline. Second, when a Gen1 node sends traffic to a Gen2/Gen3 node through a congested path, the Gen2/Gen3 NP tends to send excessive CNPs to the Gen1 RP, thus causing excessive rate reductions and throughput losses.

Our solution: Given the limited processing capacity and resources of Gen1, we cannot make it behave like Gen2 and Gen3. Instead, we try to make Gen2 and Gen3 behave like Gen1 as much as possible. Our solution is two-fold. First, we move the CNP coalescing on Gen2 and Gen3 from the RP side to the NP side. On the Gen2/Gen3 NP side, we add a per-QP CNP rate limiter and set the minimal interval between two consecutive CNPs to the value of CNP coalescing timer of the Gen1 NP. On the Gen2/Gen3 RP side, we minimize the time window for rate reduction so that the RP almost always reduces the rate upon receiving a CNP. Second, we enable per-burst rate limiting on Gen2 and Gen3.

7.3 Tuning DCQCN

There were certain practical limitations when we tuned DCQCN in Azure. First, our NICs only support global DCQCN parameter settings. Second, to optimize customer experience, we classify RDMA flows into two switch queues based on their application semantics, rather than RTTs. Hence, instead of using different DCQCN parameters for inter-datacenter and intra-datacenter traffic, we use global DCQCN parameter settings (on the NICs and switches) that work well given the large RTT variations within a region.

We took a three-step approach to tune DCQCN parameters. First, we leveraged the fluid model [113] to understand theoretical proprieties of DCQCN. Second, we ran experiments with synthetic traffic in our lab testbed to evaluate solutions to the interoperability problem and deliver reasonable parameter settings. Third, we finalized the parameter settings in test clusters, which use the same setup as production clusters carrying customer traffic. We ran stress tests with real storage applications and tuned DCQCN parameters based on the application performance.

To illustrate our findings, we use K_{min}, K_{max}, and P_{max} to denote the minimum threshold, the maximum threshold, and the maximum marking probability of RED/ECN [43], respectively. We make the following three key observations (more experiment results appear in §*B*):

- DCQCN does not suffer from RTT unfairness as it is a rate-based protocol and its rate adjustment is independent of RTT.
- To provide high throughput for DCQCN flows with large RTTs, we use *sparse* ECN marking with large $K_{max} - K_{min}$ and small P_{max}.
- DCQCN and switch buffers should be jointly tuned [112]. For example, before increasing K_{min}, we ensure that ingress thresholds for lossless traffic are large enough. Otherwise, PFC may be triggered before ECN marking.

8 Experience

In 2018, we started to enable RDMA to serve customer backend traffic. In 2019, we started to enable RDMA to serve customer frontend traffic, with storage and compute clusters co-located in the same datacenter. In 2020, we enabled intra-region RDMA in the first Azure region. As of February 2023, around 70% of traffic in Azure public regions was RDMA (Figure 1) and intra-region RDMA was supported in all Azure public regions.

8.1 Deployment and Servicing

We took a three-step approach to gradually enable RDMA in production environments. First, we leveraged the lab testbed to develop and test each individual component. Second, we conducted end-to-end stress tests in test clusters with the same software and hardware setups as those of production counterparts. In addition to normal workloads, we also injected common errors, e.g., random packet drops, to evaluate the robustness of the system. Third, we cautiously increased the deployment scale of RDMA in production environments to carry more customer traffic. During our deployment, NIC driver/firmware and switch OS updates were common. Thus it was crucial to minimize the impact of such updates to customer traffic.

Servicing switches: Compared to switches in T1 or tiers above, T0 switches, especially in compute clusters, were more challenging to service as they could be a single point of failure (SPOF) for customer VMs. In this scenario, we leveraged fast reboot [17] and warm reboot [19] to reduce the data plane disruption time from a few minutes to less than a second.

Servicing NICs: In some cases, servicing the NIC driver or firmware required unloading the NIC driver. The driver could safely unload only after all the NIC resources had been released. To this end, we needed to signal consumers, e.g., disk driver, to close RDMA connections and shift traffic to TCP. Once RDMA and other NIC features with similar concerns had been disabled, we could reload the driver.

8.2 Performance

Storage backend: Currently almost all the storage backend traffic in Azure is RDMA. It is no longer feasible to run large-scale A/B tests with customer traffic because the CPU cores saved by RDMA have been used for other purposes, not to mention customer experience degradation. Hence we demonstrate results of an A/B test conducted in a test cluster in 2018. In this test, we ran storage workloads with high transactions per second (TPS) and switched transport between RDMA and

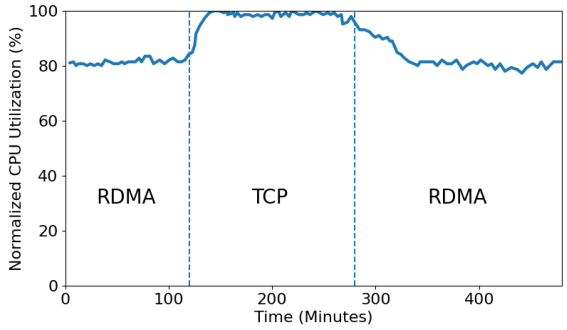

Figure 7: Average CPU usage of storage servers of a storage tenant. We normalize results to the maximum CPU usage. We switched traffic between RDMA and TCP twice.

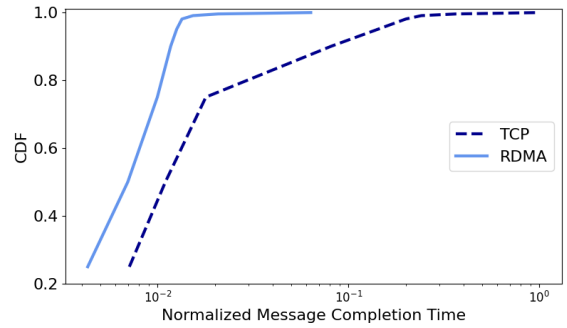

Figure 8: Message completion times of storage backend traffic measured in a test cluster. We normalize results to the maximum message completion time.

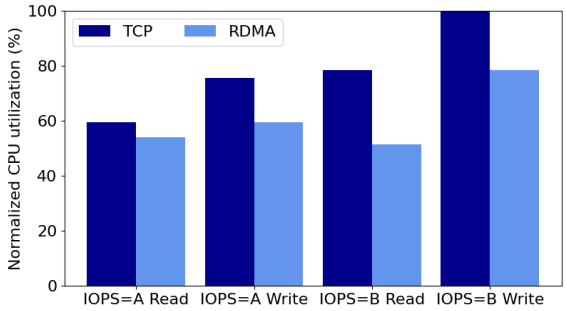

Figure 9: Average CPU usage of the host domain. We normalize results to the maximum value.

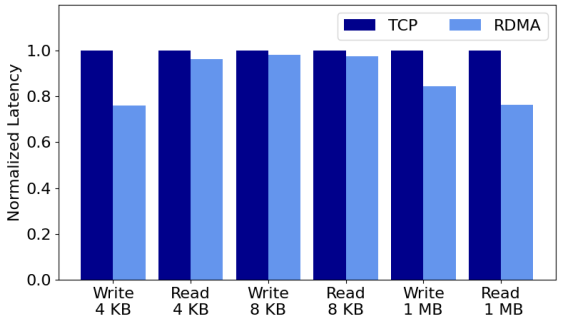

Figure 10: Average access latencies of a type of SSDs across *all* Azure public regions between February 22, 2022, and February 22, 2023. We normalize RDMA results to corresponding TCP results.

TCP. Figure 7 plots normalized CPU utilization of storage servers during two transport switches. It is worthwhile to note that CPU utilization here includes all the types of processing overhead, e.g., storage application, Azure Storage Network Protocol, and TCP/IP stack. Figure 8 gives message completion times measured in Azure Storage Network Protocol layer (Figure 4), which excludes the overhead of application processing. Compared to TCP, RDMA achieved obvious CPU saving and significantly accelerated network data transfer.

Storage frontend: Since we cannot perform large-scale A/B tests with customer traffic, we present results of an A/B test conducted in a test cluster in 2018. In this test, we used `DiskSpd` to generate read and write workloads at A IOPS and B IOPS (A < B). The I/O size was 8 KB. Figure 9 gives average CPU utilization of the host domain during the test period. Compared to TCP, RDMA could reduce the CPU utilization by up to 34.5%.

To understand the performance improvement introduced by RDMA, we leverage an always-on storage monitoring service. This service allocates some VMs in each region, uses them to periodically generate disk read and write workloads, and collects end-to-end performance results. The monitoring service covers different I/O sizes, types of disks, and transports for storage frontend traffic.

Figure 10 shows the overall average access latencies of a type of SSDs across all Azure public regions collected by the monitoring service for a year. Note that the RDMA and TCP in this figure only refer to the transport of frontend traffic generated by test VMs. We normalize RDMA results to corresponding TCP results. Compared to TCP, RDMA yielded better access latencies with every I/O size. In particular, 1 MB I/O requests benefited the most from RDMA with 23.8% and 15.6% latency reductions for read and write, respectively. This is due to the fact that large I/O requests are more sensitive to throughput than smaller I/O requests, and RDMA improves throughput drastically since it can run at line rate using a single connection without slow starts.

Congestion control: We ran stress tests in a test cluster to drive the DCQCN parameter setting that could achieve reasonable performance even under peak workloads. Figure 11 gives results of the 99th percentile message completion time, the key metric we used to guide our tuning. At the beginning,

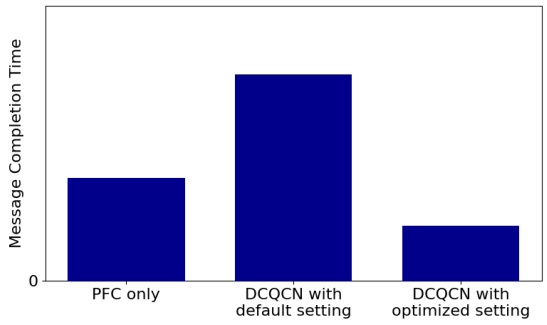

Figure 11: The 99th percentile message completion times of different schemes measured in a test cluster.

we disabled DCQCN and only tuned switch buffer parameters, e.g., the dynamic threshold of ingress lossless queues, to explore the best performance achieved by PFC only. After reaching the best performance of PFC only, we enabled DCQCN using the default parameter setting, which was derived on the lab testbed using synthetic traffic. While DCQCN reduced the number of PFC pause frames, it degraded the tail message completion time as the default setting reduced the sending rate too aggressively. Given this, we adjusted ECN marking parameters to improve DCQCN's throughput. With optimized setting, DCQCN performs better than using PFC alone. Our key takeaway from this tuning experience was that DCQCN and switch buffer should be jointly tuned to optimize the application performance, rather than PFC pause duration.

8.3 Problems Discovered and Fixed

During tests and deployments, we discovered and fixed a series of problems in NICs, switches and our RDMA applications.

FMR hidden fence: In sK-RDMA (§4.2), every I/O request from compute servers requires a `FMR` request followed by a `Send` request to the storage server, which contains the description of `FMR` registered memory and storage commands. Therefore, the send queue consists of many `FMR`/`Send` pairs.

When we deployed sK-RDMA in compute and storage clusters located in different datacenters, we found that the frontend traffic showed extremely low throughput, even though we kept many outstanding `FMR`/`Send` pairs in the send queue. To debug this problem, we used RDMA Estats to collect $T_5 - T_1$ latency for every `Send` request (§5). We found a strong correlation between $T_5 - T_1$ and inter-datacenter RTT, and noticed that there was only a single outstanding `Send` request per RTT. After we shared these findings with the NIC vendor, they identified the root cause: to simplify the implementation, NICs processed the `FMR` request only after the completions of previously posted requests. In sK-RDMA, the `FMR` request created a *hidden fence* between two `Send` requests, thus only allowing a single `Send` request in the air, which could not fill the large network pipe between datacenters. We have worked with the NIC vendor to fix this problem in the new NIC driver.

PFC and MACsec: After we enabled PFC on long-haul links between T2 and RH, many long-haul links reported high packet corruption rates, thus triggering many alerts and even auto-mitigation. When we debugged this problem, we found that there was a strong correlation between the number of PFC frames and the number of corrupted packets. Then we suspected this was probably due to unexpected behaviors in the interactions between PFC and MACsec, since MACsec was *only* enabled on long-haul links within the region. We found that MACSec standard [21] did not specify whether PFC frames should be encrypted. After we checked with vendors, we found that different switches had no agreement on whether PFC frames sent should be encrypted and what to do with arriving encrypted PFC frames. Some switches (*A*) did not encrypt outbound PFC frames. Other switches (*B*) encrypted outbound PFC frames and also expected inbound PFC frames to be encrypted. As a result, when *A* switches sent unencrypted PFC frames to *B* switches, *B* switches would treat those PFC frames as corrupted packets and report errors. We validated the above analysis by running interoperability tests in our testbed. We have worked with switch vendors to standardize how MACsec enabled switch ports encrypt outgoing PFC frames and process arriving PFC frames.

Congestion leaking: The problem was found in the testbed. When we enabled interoperability features (§7.2) on Gen2 NICs, we found that their throughput would be degraded. To dig into this problem, we used the water filling algorithm to calculate theoretical per-QP throughput results and compared them with actual throughput results measured from the testbed. We had two interesting observations when comparing the results. First, flows sent by a Gen2 NIC always had near identical sending rates regardless of their congestion degrees. Second, actual sending rates were very close to the theoretical sending rate of the slowest flow sent from the NIC. It seemed that all the flows from a Gen2 NIC were throttled by the slowest flow. We reported these observations to the NIC vendor, and they identified a head-of-line blocking in the NIC firmware. We have fixed this problem on all the NICs with interoperability features.

Slow receiver due to loopback RDMA: This problem was found in a test cluster. During stress tests, we found that a large number of servers sent PFC pause frames to T0 switches. However, unlike slow receivers found before, PFC watchdog was not triggered on any T0 switches. It seemed that those servers only gracefully slowed down the traffic coming from T0 switches, rather than completely blocking T0 switches for a long duration. In addition, where slow receivers were common at Azure's scale, it was very unlikely that a significant portion of servers in a cluster became "mad" simultaneously.

Based on the above observations, we suspected that these

slow receivers were caused by our applications. We found that each server actually ran multiple RDMA application instances. All the inter-instance traffic ran on RDMA, regardless of their locations. Therefore, loopback traffic and external traffic co-existed on every NIC, thus creating a 2:1 congestion on PCIe lanes of the NIC. Since the NIC could not mark ECN, it could only throttle loopback traffic and external traffic through PCIe back pressure and PFC pause frames. To validate the above analysis, we disabled RDMA for loopback traffic on some servers, then these servers stopped sending PFC frames. We notice that recent work [61, 70] also found this problem.

9 Lessons and Open Problems

In this section, we summarize the lessons learned from our experience and discuss open problems for future exploration.

Failovers are very expensive for RDMA. While we have implemented failover solutions in both sU-RDMA and sK-RDMA as the last resort, we find that failovers are particularly expensive for RDMA, and should be avoided as much as possible. Cloud providers adopt RDMA to save CPU cores and then use freed CPU cores for other purposes. To move traffic away from RDMA, we need to allocate *extra* CPU cores to carry these traffic. This increases CPU utilization, and even runs out of CPU cores at high loads. Hence, it is risky to perform large-scale RDMA failovers, which we treat as serious incidents in Azure. Given the risk, only after all the tests have passed, we gradually increase the RDMA deployment scale. During the rollout, we continuously monitor network performance and immediately stop the rolltout once anomalies are detected. After unavoidable failovers, we should aggressively switch back to RDMA when possible.

Host network and physical network should be converged. In 8.3, we present a new type of slow receivers, which is essentially due to congestion inside the host. Recent work [24] also presents evidence and characterization of host congestion in production clusters. We believe this problem is just a tip of the iceberg, while many problematic behaviors between host network and physical network remain unexposed. In conventional wisdom, host network and physical network are separated entities and NIC is their border. If we look into the host, it is essentially a network connecting heterogeneous nodes (e.g., CPU, GPU, DPU) with proprietary high speed links (e.g., PCIe link and NVLink) and switches (e.g., PCIe switch and NVSwitch). Inter-host traffic can be treated as north-south traffic for the host. With the increase of the datacenter link capacity and wide adoptions of hardware offloading and device direct access technologies (e.g., GPUDirect RDMA), inter-host traffic tends to consume larger and more various resources inside the host, thus resulting in more complex interactions with intra-host traffic.

We believe that host network and physical network should be converged in the future. And we envision this converged network will be an important step towards the dis-aggregated cloud. We look forward to operating this converged network in similar ways as we manage physical network today.

Switch buffer is increasingly important and needs more innovations. The conventional wisdom [26] suggests that low latency datacenter congestion control [26, 71, 82, 112] can alleviate the need of large switch buffers as they can preserve short queues. However, we find a strong correlation between switch buffers and RDMA performance problems in production. Clusters with smaller switch buffers tend to have more performance problems. And many performance problems can be mitigated by just tuning switch buffer parameters without touching DCQCN. This is why we always tune switch buffers before touching DCQCN (§8.2). The importance of switch buffer lies in the prevalence of bursty traffic and short-lived congestion events in datacenters [108]. Conventional congestion control solutions are ill-suited for such scenarios given their reactive nature. Instead, switch buffer plays as the first resort to absorb bursts and provide fast responses.

With the increase in datacenter link speed, we believe that switch buffer is increasingly important, thus deserving more efforts and innovations. First, the buffer size per port per Gbps on pizza box switches keeps decreasing in recent years [31]. Some switch ASICs even split the packet memory into multiple partitions, thus reducing effective buffer resource. We encourage more efforts to put into the development ASICs with deeper packet buffers and more unified architectures. Second, today's commodity switch ASICs only provide buffer management mechanisms [40] designed decades ago, thus limiting the scope of solutions to handle congestion. Following the trend of programmable data plane [32], we envision that future switch ASICs would provide more programmability on buffer models and interfaces, thus enabling the implementation of more effective buffer management solutions [22].

Cloud needs unified behavior models and interfaces for network devices. The diversity in software and hardware brings significant challenges to network operation at cloud scale. Different NICs from the same vendor can even have different behaviors that cause interoperability problems, not to mention devices from different vendors. In spite of all the efforts we put into the unified switch software (§6) and NIC congestion control (§7.2), we still experienced problems due to diversity, e.g., unexpected interactions between PFC and MACsec (§8.3). We envision that more unified models and interfaces will emerge to simplify operations and accelerate innovations in the cloud. Some key areas include chassis switches, smart network appliances, and RDMA NICs. We notice that there have been some efforts on standardizing congestion control for different data paths [85] and APIs for heterogeneous smart appliances [16].

Testing new network devices is crucial and challenging. From the day one of this project, we have been making large investments in building various testing tools and running rig-

orous tests in both testbeds and test clusters. Despite the significant number of problems discovered during tests, we still found some problems during deployments (§8.3), mostly due to micro-behaviors and corner cases that were overlooked. Some burning questions are given as follows:

- How to precisely capture micro-behaviors of RDMA NIC implementations in various scenarios?
- Despite many endeavors to measure switches' micro-behaviors (§6.3), we still rely on domain knowledge to design test cases. How to systematically test the correctness and performance of a switch?

These questions motivate us to rethink challenges and requirements of testing emerging network devices with more and more features. First, many features lack clear specifications, which is a prerequisite for systematic testing. Many seemingly simple features are actually entangled with complex interactions between software and hardware. We believe that unified behavior models and interfaces discussed above can help with this. Second, the test system should be able to interact with network devices at high speed, and precisely capture micro-behaviors. We believe programmable hardware can help on this [33, 37]. We note that there have been some recent progresses on testing RDMA NICs [69, 70] and programmable switches [37, 110].

10 Related Work

This paper focuses on RDMA for cloud storage. The literature of RDMA and storage systems is vast. Here we only discuss some closely related ideas.

Deployment experience of RDMA and storage networks: Before this project, we had deployed RDMA to support some Bing workloads and encountered many problems, such as PFC storms, PFC deadlocks, and slow receivers [50]. We learnt several lessons from this deployment. Gao et al. [46] summarized the experience of deploying intra-cluster RDMA to support storage backend traffic in Alibaba. Miao et al. [80] presented two generations of storage network stacks to carry Alibaba's storage frontend traffic: LUNA and SOLAR. LUNA is a high performance user-space TCP stack while SOLAR is a storage-oriented UDP stack implemented in proprietary DPU. Scalable Reliable Datagram (SRD) [96] is a cloud-optimized transport protocol implemented in AWS custom Nitro networking card, and used by HPC, ML, and storage applications [7]. In contrast, we use commodity hardware to enable intra-region RDMA to support both storage frontend and backend traffic.

Congestion control in datacenters: There is a large body of work on datacenter congestion control, including ECN-based [26, 27, 99, 112], delay-based [71, 72, 76, 82], INT-based [23, 75, 101], credit-based [34, 38, 45, 52, 55, 84, 86, 88] and packet scheduling [28, 30, 36, 49, 54]. Our work focuses on regional networks which have large RTT variations. We notice that some efforts [95, 107] target at similar scenarios.

Improve RDMA in datacenters: In addition to congestion control, there are many efforts to improve RDMA's reliability, security and performance in datacenters, such as deadlock mitigation [56, 92, 103], support of multi-path [77], resilience over lossy networks [78, 83, 102], security mechanisms [94, 98, 104], virtualization [53, 67, 89, 100], testing [69, 70], and performance isolation in multi-tenant environments [109]. Our work focuses on first party traffic in the trusted environment. Given the limited retransmission performance of our NICs, we enable RDMA over lossless networks (§2.4). We leverage storage stack and MACsec to encrypt data (§3.2) and use PFC watchdog to mitigate PFC storms and deadlocks (§3.1).

Accelerate storage systems using RDMA and other techniques: Many proposals [41, 62–66, 74, 93, 106, 111] leverage RDMA to accelerate storage systems or networked systems in general. Similar to some solutions [13, 47, 74, 90], our RDMA protocols (§4) provide socket-like interfaces to keep compatibility with legacy storage stack. In addition to RDMA, some recent proposals improve storage systems using new kernel designs [58, 59, 73] and SmartNIC [68, 81].

11 Conclusions and Future Work

In this paper, we summarize our experience in deploying intra-region RDMA to support storage workloads in Azure. The high complexity and heterogeneity of our infrastructure brings a series of new challenges. We have made several changes to our network infrastructure to address these challenges. Today, around 70% of traffic in Azure is RDMA and intra-region RDMA is supported in all Azure public regions. RDMA helps us achieve significant disk I/O performance improvements and CPU core savings.

In the future, we plan to further improve our storage systems through innovations on system architecture, hardware acceleration, and congestion control. We also plan to bring RDMA to more scenarios.

References

[1] Amazon ebs volume types. https://aws.amazon.com/ebs/volume-types/.

[2] Amazon web services region. https://aws.amazon.com/about-aws/global-infrastructure/regions_az/.

[3] Arista 7500r switch architecture ('a day in the life of a packet'). https://www.arista.com/assets/data/pdf/Whitepapers/Arista7500RSwitchArchitectureWP.pdf.

[4] Azure managed disk types. https://docs.microsoft.com/en-us/azure/virtual-machines/disks-types.

[5] Azure region. https://docs.microsoft.com/en-us/azure/availability-zones/az-overview.

[6] Cisco silicon one product family. https://www.cisco.com/c/dam/en/us/solutions/collateral/silicon-one/white-paper-sp-product-family.pdf.

[7] A decade of ever-increasing provisioned iops for amazon ebs. https://aws.amazon.com/blogs/aws/a-decade-of-ever-increasing-provisioned-iops-for-amazon-ebs/.

[8] Google cloud region. https://cloud.google.com/compute/docs/regions-zones.

[9] Keysight network test solutions. https://www.keysight.com/us/en/solutions/network-test.html.

[10] Packet testing framework (ptf). https://github.com/p4lang/ptf.

[11] Pfc watchdog in sonic. https://github.com/sonic-net/SONiC/wiki/PFC-Watchdog-Design.

[12] Priority flow control: Build reliable layer 2 infrastructure. https://e2e.ti.com/cfs-file/__key/communityserver-discussions-components-files/908/802.1q-Flow-Control-white_5F00_paper_5F00_c11_2D00_542809.pdf.

[13] rsocket(7) - linux man page. https://linux.die.net/man/7/rsocket.

[14] Smb direct. https://learn.microsoft.com/en-us/windows-server/storage/file-server/smb-direct.

[15] Software for open networking in the cloud (sonic). https://sonic-net.github.io/SONiC/.

[16] Sonic-dash - disaggregated api for sonic hosts. https://github.com/sonic-net/DASH.

[17] Sonic fast reboot. https://github.com/sonic-net/SONiC/blob/master/doc/fast-reboot/fastreboot.pdf.

[18] sonic-mgmt: Management and automation code used for sonic testbed deployment, tests and reporting. https://github.com/sonic-net/sonic-mgmt.

[19] Sonic warm reboot. https://github.com/sonic-net/SONiC/blob/master/doc/warm-reboot/SONiC_Warmboot.md.

[20] Switch abstraction interface (sai). https://github.com/opencomputeproject/SAI.

[21] Ieee standard for local and metropolitan area networks-media access control (mac) security. *IEEE Std 802.1AE-2018 (Revision of IEEE Std 802.1AE-2006)*, 2018.

[22] Vamsi Addanki, Maria Apostolaki, Manya Ghobadi, Stefan Schmid, and Laurent Vanbever. Abm: active buffer management in datacenters. In *SIGCOMM 2022*.

[23] Vamsi Addanki, Oliver Michel, and Stefan Schmid. Powertcp: Pushing the performance limits of datacenter networks. In *NSDI 2022*.

[24] Saksham Agarwal, Rachit Agarwal, Behnam Montazeri, Masoud Moshref, Khaled Elmeleegy, Luigi Rizzo, Marc Asher de Kruijf, Gautam Kumar, Sylvia Ratnasamy, David Culler, and Amin Vahdat. Understanding host interconnect congestion. In *HotNets 2022*.

[25] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM 2008*.

[26] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *SIGCOMM 2010*.

[27] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less is more: trading a little bandwidth for ultra-low latency in the data center. In *NSDI 2012*.

[28] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pfabric: Minimal near-optimal datacenter transport. In *SIGCOMM 2013*.

[29] InfiniBand Trade Association. Supplement to infiniband architecture specification volume 1 release 1.2. 1 annex a17: Rocev2, 2014.

[30] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. Information-agnostic flow scheduling for commodity data centers. In *NSDI 2015*.

[31] Wei Bai, Shuihai Hu, Kai Chen, Kun Tan, and Yongqiang Xiong. One more config is enough: Saving (dc) tcp for high-speed extremely shallow-buffered datacenters. In *INFOCOM 2020*.

[32] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David

Walker. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 2014.

[33] Pietro Bressana, Noa Zilberman, and Robert Soulé. Finding hard-to-find data plane bugs with a pta. In *CoNEXT 2020*.

[34] Qizhe Cai, Mina Tahmasbi Arashloo, and Rachit Agarwal. dcpim: Near-optimal proactive datacenter transport. In *SIGCOMM 2022*.

[35] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows azure storage: A highly available cloud storage service with strong consistency. In *SOSP 2011*.

[36] Li Chen, Kai Chen, Wei Bai, and Mohammad Alizadeh. Scheduling mix-flows in commodity datacenters with karuna. In *SIGCOMM 2016*.

[37] Yanqing Chen, Bingchuan Tian, Chen Tian, Li Dai, Yu Zhou, Mengjing Ma, Ming Tang, Hao Zheng, Zhewen Yang, Guihai Chen, Dennis Cai, and Ennan Zhai. Norma: Towards practical network load testing. In *NSDI 2023*.

[38] Inho Cho, Keon Jang, and Dongsu Han. Credit-scheduled delay-bounded congestion control for datacenters. In *SIGCOMM 2017*.

[39] Sean Choi, Boris Burkov, Alex Eckert, Tian Fang, Saman Kazemkhani, Rob Sherwood, Ying Zhang, and Hongyi Zeng. Fboss: building switch software at scale. In *SIGCOMM 2018*.

[40] Abhijit K. Choudhury and Ellen L. Hahne. Dynamic queue length thresholds for shared-memory packet switches. *IEEE/ACM Transactions on Networking*, 1998.

[41] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. Farm: Fast remote memory. In *NSDI 2014*.

[42] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure accelerated networking: SmartNICs in the public cloud. In *NSDI 2018*.

[43] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1993.

[44] Philip Werner Frey and Gustavo Alonso. Minimizing the hidden cost of rdma. In *ICDCS 2009*.

[45] Peter X. Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. phost: Distributed near-optimal datacenter transport over commodity network fabric. In *CoNEXT 2015*.

[46] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, Fei Feng, Yan Zhuang, Fan Liu, Pan Liu, Xingkui Liu, Zhongjie Wu, Junping Wu, Zheng Cao, Chen Tian, Jinbo Wu, Jiaji Zhu, Haiyong Wang, Dennis Cai, and Jiesheng Wu. When cloud storage meets RDMA. In *NSDI 2021*.

[47] Dror Goldenberg, Michael Kagan, Ran Ravid, and Michael S Tsirkin. Zero copy sockets direct protocol over infiniband-preliminary implementation and performance analysis. In *HOTI 2005*.

[48] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: a scalable and flexible data center network. In *SIGCOMM 2009*.

[49] Matthew P Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert NM Watson, Andrew W Moore, Steven Hand, and Jon Crowcroft. Queues don't matter when you can jump them! In *NSDI 2015*.

[50] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. Rdma over commodity ethernet at scale. In *SIGCOMM 2016*.

[51] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *SIGCOMM 2015*.

[52] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting datacenter networks

and stacks for low latency and high performance. In *SIGCOMM 2017*.

[53] Zhiqiang He, Dongyang Wang, Binzhang Fu, Kun Tan, Bei Hua, Zhi-Li Zhang, and Kai Zheng. Masq: Rdma for virtual private cloud. In *SIGCOMM 2020*.

[54] Chi-Yao Hong, Matthew Caesar, and P Godfrey. Finishing flows quickly with preemptive scheduling. In *SIGCOMM 2012*.

[55] Shuihai Hu, Wei Bai, Gaoxiong Zeng, Zilong Wang, Baochen Qiao, Kai Chen, Kun Tan, and Yi Wang. Aeolus: A building block for proactive transport in datacenters. In *SIGCOMM 2020*.

[56] Shuihai Hu, Yibo Zhu, Peng Cheng, Chuanxiong Guo, Kun Tan, Jitendra Padhye, and Kai Chen. Tagger: Practical pfc deadlock prevention in data center networks. In *CoNEXT 2017*.

[57] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. Erasure coding in windows azure storage. In *ATC 2012*.

[58] Jaehyun Hwang, Qizhe Cai, Ao Tang, and Rachit Agarwal. Tcp $\approx$ rdma: Cpu-efficient remote storage access with i10. In *NSDI 2020*.

[59] Jaehyun Hwang, Midhul Vuppalapati, Simon Peter, and Rachit Agarwal. Rearchitecting linux storage stack for μs latency and high throughput. In *OSDI 2021*.

[60] IEEE. 802.11 qbb. priority based flow control. 2008.

[61] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed dnn training in heterogeneous gpu/cpu clusters. In *OSDI 2020*.

[62] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter rpcs can be general and fast. In *NSDI 2019*.

[63] Anuj Kalia, Michael Kaminsky, and David G Andersen. Design guidelines for high performance rdma systems. In *ATC 2016*.

[64] Anuj Kalia, Michael Kaminsky, and David G Andersen. Fasst: Fast, scalable and simple distributed transactions with two-sided (rdma) datagram rpcs. In *OSDI 2016*.

[65] Anuj Kalia, Michael Kaminsky, and David G Andersen. Using rdma efficiently for key-value services. In *SIGCOMM 2014*.

[66] Daehyeok Kim, Amirsaman Memaripour, Anirudh Badam, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Shachar Raindel, Steven Swanson, Vyas Sekar, and Srinivasan Seshan. Hyperloop: group-based nic-offloading to accelerate replicated transactions in multi-tenant storage systems. In *SIGCOMM 2018*.

[67] Daehyeok Kim, Tianlong Yu, Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Shachar Raindel, Chuanxiong Guo, Vyas Sekar, and Srinivasan Seshan. Freeflow: Software-based virtual rdma networking for containerized clouds. In *NSDI 2019*.

[68] Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostić, Youngjin Kwon, Simon Peter, and Emmett Witchel. Linefs: Efficient smartnic offload of a distributed file system with pipeline parallelism. In *SOSP 2021*.

[69] Xinhao Kong, Jingrong Chen, Wei Bai, Yechen Xu, Mahmoud Elhaddad, Shachar Raindel, Jitendra Padhye, and Alvin R Lebeck Danyang Zhuo. Understanding rdma microarchitecture resources for performance isolation. In *NSDI 2023*.

[70] Xinhao Kong, Yibo Zhu, Huaping Zhou, Zhuo Jiang, Jianxi Ye, Chuanxiong Guo, and Danyang Zhuo. Collie: Finding performance anomalies in rdma subsystems. In *NSDI 2022*.

[71] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan M. G. Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. Swift: Delay is simple and effective for congestion control in the datacenter. In *SIGCOMM 2020*.

[72] Changhyun Lee, Chunjong Park, Keon Jang, Sue Moon, and Dongsu Han. Accurate latency-based congestion feedback for datacenters. In *ATC 2015*.

[73] Gyusun Lee, Seokha Shin, Wonsuk Song, Tae Jun Ham, Jae W. Lee, and Jinkyu Jeong. Asynchronous I/O stack: A low-latency kernel I/O stack for Ultra-Low latency SSDs. In *ATC 2019*.

[74] Bojie Li, Tianyi Cui, Zibo Wang, Wei Bai, and Lintao Zhang. Socksdirect: Datacenter sockets can be fast and compatible. In *SIGCOMM 2019*.

[75] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. Hpcc: High precision congestion control. In *SIGCOMM 2019*.

[76] Shiyu Liu, Ahmad Ghalayini, Mohammad Alizadeh, Balaji Prabhakar, Mendel Rosenblum, and Anirudh Sivaraman. Breaking the transience-equilibrium nexus: A new approach to datacenter packet transport. In *NSDI 2021*.

[77] Yuanwei Lu, Guo Chen, Bojie Li, Kun Tan, Yongqiang Xiong, Peng Cheng, Jiansong Zhang, Enhong Chen, and Thomas Moscibroda. Multi-path transport for rdma in datacenters. In *NSDI 2018*.

[78] Yuanwei Lu, Guo Chen, Zhenyuan Ruan, Wencong Xiao, Bojie Li, Jiansong Zhang, Yongqiang Xiong, Peng Cheng, and Enhong Chen. Memory efficient loss recovery for hardware-based transport in datacenter. In *APNet 2017*.

[79] Matt Mathis, John Heffner, and Rajiv Raghunarayan. Tcp extended statistics mib (rfc 4898). Technical report, 2007.

[80] Rui Miao, Lingjun Zhu, Shu Ma, Kun Qian, Shujun Zhuang, Bo Li, Shuguang Cheng, Jiaqi Gao, Yan Zhuang, Pengcheng Zhang, Rong Liu, Chao Shi, Binzhang Fu, Jiaji Zhu, Jiesheng Wu, Dennis Cai, and Hongqiang Harry Liu. From luna to solar: The evolutions of the compute-to-storage networks in alibaba cloud. In *SIGCOMM 2022*.

[81] Jaehong Min, Ming Liu, Tapan Chugh, Chenxingyu Zhao, Andrew Wei, In Hwan Doh, and Arvind Krishnamurthy. Gimbal: enabling multi-tenant storage disaggregation on smartnic jbofs. In *SIGCOMM 2021*.

[82] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. Timely: Rtt-based congestion control for the datacenter. In *SIGCOMM 2015*.

[83] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. Revisiting network support for rdma. In *SIGCOMM 2018*.

[84] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *SIGCOMM 2018*.

[85] Akshay Narayan, Frank Cangialosi, Deepti Raghavan, Prateesh Goyal, Srinivas Narayana, Radhika Mittal, Mohammad Alizadeh, and Hari Balakrishnan. Restructuring endpoint congestion control. In *SIGCOMM 2018*.

[86] Vladimir Olteanu, Haggai Eran, Dragos Dumitrescu, Adrian Popa, Cristi Baciu, Mark Silberstein, Georgios Nikolaidis, Mark Handley, and Costin Raiciu. An edge-queued datagram service for all datacenter traffic. In *NSDI 2022*.

[87] Madhav Himanshubhai Pandya, Aaron William Ogus, Zhong Deng, and Weixiang Sun. Transport protocol and interface for efficient data transfer over rdma fabric, August 2 2022. US Patent 11,403,253.

[88] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Deverat Shah, and Hans Fugal. Fastpass: A centralized "zero-queue" datacenter network. In *SIGCOMM 2014*.

[89] Jonas Pfefferle, Patrick Stuedi, Animesh Trivedi, Bernard Metzler, Ionnis Koltsidas, and Thomas R Gross. A hybrid i/o virtualization framework for rdma-capable network interfaces. *ACM SIGPLAN Notices*, 2015.

[90] Jim Pinkerton. Sockets direct protocol v1. 0 rdma consortium. 2003.

[91] Leon Poutievski, Omid Mashayekhi, Joon Ong, Arjun Singh, Mukarram Tariq, Rui Wang, Jianan Zhang, Virginia Beauregard, Patrick Conner, Steve Gribble, Rishi Kapoor, Stephen Kratzer, Nanfang Li, Hong Liu, Karthik Nagaraj, Jason Ornstein, Samir Sawhney, Ryohei Urata, Lorenzo Vicisano, Kevin Yasumura, Shidong Zhang, Junlan Zhou, and Amin Vahdat. Jupiter evolving: Transforming google's datacenter network via optical circuit switches and software-defined networking. In *SIGCOMM 2022*.

[92] Kun Qian, Wenxue Cheng, Tong Zhang, and Fengyuan Ren. Gentle flow control: avoiding deadlock in lossless networks. In *SIGCOMM 2019*.

[93] Waleed Reda, Marco Canini, Dejan Kostic, and Simon Peter. Rdma is turing complete, we just did not know it yet! In *NSDI 2022*.

[94] Benjamin Rothenberger, Konstantin Taranov, Adrian Perrig, and Torsten Hoefler. Redmark: Bypassing rdma security mechanisms. In *USENIX Security 2021*.

[95] Ahmed Saeed, Varun Gupta, Prateesh Goyal, Milad Sharif, Rong Pan, Mostafa Ammar, Ellen Zegura, Keon Jang, Mohammad Alizadeh, Abdul Kabbani, and Amin Vahdat. Annulus: A dual congestion control loop for datacenter and wan traffic aggregates. In *SIGCOMM 2020*.

[96] Leah Shalev, Hani Ayoub, Nafea Bshara, and Erez Sabbag. A cloud-optimized transport protocol for elastic and scalable hpc. *IEEE Micro*, 2020.

[97] Cheng Tan, Ze Jin, Chuanxiong Guo, Tianrong Zhang, Haitao Wu, Karl Deng, Dongming Bi, and Dong Xiang. Netbouncer: Active device and link failure localization in data center networks. In *NSDI 2019*.

[98] Konstantin Taranov, Benjamin Rothenberger, Adrian Perrig, and Torsten Hoefler. srdma: efficient nic-based authentication and encryption for remote direct memory access. In *ATC 2020*.

[99] Balajee Vamanan, Jahangir Hasan, and TN Vijaykumar. Deadline-aware datacenter tcp (d2tcp). In *SIGCOMM 2012*.

[100] Dongyang Wang, Binzhang Fu, Gang Lu, Kun Tan, and Bei Hua. vsocket: virtual socket interface for rdma in public clouds. In *VEE 2019*.

[101] Weitao Wang, Masoud Moshref, Yuliang Li, Gautam Kumar, TS Eugene Ng, Neal Cardwell, and Nandita Dukkipati. Poseidon: Efficient, robust, and practical datacenter cc via deployable int. In *NSDI 2023*.

[102] Zilong Wang, Layong Luo, Qingsong Ning, Chaoliang Zeng, Wenxue Li, Xinchen Wan, Peng Xie, Tao Feng, Ke Cheng, Xiongfei Geng, Tianhao Wang, Weicheng Ling, Kejia Huo, Pingbo An, Kui Ji, Shideng Zhang, Bin Xu, Ruiqing Feng, Tao Ding, Kai Chen, and Chuanxiong Guo. Srnic: A scalable architecture for rdma nics. In *NSDI 2023*.

[103] Xinyu Crystal Wu and TS Eugene Ng. Detecting and resolving pfc deadlocks with itsy entirely in the data plane. In *INFOCOM 2022*.

[104] Jiarong Xing, Kuo-Feng Hsu, Yiming Qiu, Ziyang Yang, Hongyi Liu, and Ang Chen. Bedrock: Programmable network support for secure rdma systems. In *USENIX Security 2022*.

[105] Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Tameesh Suri, Manu Awasthi, Zvika Guz, Anahita Shayesteh, and Vijay Balakrishnan. Performance analysis of nvme ssds and their implication on real world databases. In *SYSTOR 2015*.

[106] Jian Yang, Joseph Izraelevitz, and Steven Swanson. Orion: A distributed file system for non-volatile main memory and rdma-capable networks. In *FAST 2019*.

[107] Gaoxiong Zeng, Wei Bai, Ge Chen, Kai Chen, Dongsu Han, Yibo Zhu, and Lei Cui. Congestion control for cross-datacenter networks. In *ICNP 2019*.

[108] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. High-resolution measurement of data center microbursts. In *IMC 2017*.

[109] Yiwen Zhang, Yue Tan, Brent Stephens, and Mosharaf Chowdhury. Justitia: Software multi-tenancy in hardware kernel-bypass networks. In *NSDI 2022*.

[110] Naiqian Zheng, Mengqi Liu, Ennan Zhai, Hongqiang Harry Liu, Yifan Li, Kaicheng Yang, Xuanzhe Liu, and Xin Jin. Meissa: scalable network testing for programmable data planes. In *SIGCOMM 2022*.

[111] Bohong Zhu, Youmin Chen, Qing Wang, Youyou Lu, and Jiwu Shu. Octopus+: An rdma-enabled distributed persistent memory file system. *ACM Transactions on Storage*, 2021.

[112] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale rdma deployments. In *SIGCOMM 2015*.

[113] Yibo Zhu, Monia Ghobadi, Vishal Misra, and Jitendra Padhye. Ecn or delay: Lessons learnt from analysis of dcqcn and timely. In *CoNEXT 2016*.

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

A SONiC buffer analysis

```
"BUFFER_POOL": {
    "ingress_pool": {
        "size": "18000000",
        "type": "ingress",
        "mode": "dynamic",
        "xoff": "6000000"
    },
    "egress_lossy_pool": {
        "size": "14000000",
        "type": "egress",
        "mode": "dynamic"
    },
    "egress_lossless_pool": {
        "size": "24000000",
        "type": "egress",
        "mode": "static"
    }
}

"BUFFER_PROFILE": {
    "ingress_lossless_profile" : {
        "pool": "[BUFFER_POOL|ingress_pool]",
        "size": "1248",
        "dynamic_th": "-3",
        "xoff": "96928",
        "xon"   "1248",
        "xon_offset"    "2496"
    },
    "ingress_lossy_profile": {
        "pool": "[BUFFER_POOL|ingress_pool]",
        "size": "0",
        "static_th":"24000000"
    },
    "egress_lossless_profile": {
        "pool": "[BUFFER_POOL|egress_lossless_pool]",
        "size": "0",
        "static_th": "24000000"
    },
    "egress_lossy_profile": {
        "pool": "[BUFFER_POOL|egress_lossy_pool]",
        "size": "1664",
        "dynamic_th": "-1"
    }
}
```

Listing 1: SONiC Buffer Configuration Example

Listing 1 gives a buffer configuration example of a SONiC pizza box switch with 24 MB packet buffer. `ingress_pool` has 18 MB (`size`) shared buffer for all the ingress queues, and 6 MB (`xoff`) PFC headroom buffer exclusively for ingress lossless queues in the paused state. `egress_lossy_pool` and `egress_lossless_pool` have 14 MB and 24 MB shared buffer, respectively. It is worthwhile to notice that the sum of pool sizes can be larger than the physical buffer limit, as they are only virtual counters for admission control purposes.

Lossless packets are mapped to both ingress lossless queues (`ingress_lossless_profile`) and egress lossless queues (`egress_lossless_profile`). We use Dynamic Threshold (DT) algorithm [40] to manage the buffer occupancy of the ingress lossless queue in the 18 MB shared buffer space of `ingress_pool`. DT algorithm is controlled by a parameter called α, which is 1/8 ($2^{\texttt{dynamic\_th}}$) in Listing 1. Once the ingress lossless queue hits the dynamic threshold ($\alpha\times$ remaining buffer), it will enter the paused state (send PFC pause frames) and start to use PFC headroom. All the ingress lossless queues in the paused state share a 6 MB PFC headroom pool (`xoff` of `ingress_pool`). Each ingress lossless queue can use up to 96928 bytes buffer (`xoff` of `ingress_lossless_profile`) in the PFC headroom pool. We bypass the egress admission control for lossless traffic by setting the static threshold of the egress lossless queue (`static_th` of `egress_lossless_profile`) to 24 MB, which equals to the switch buffer size.

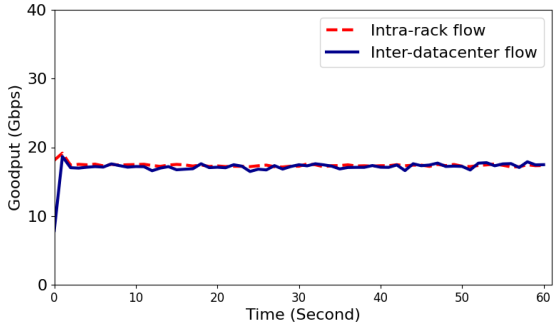

Figure 12: Goodput of two flows with different RTTs.

In contrast, we only want to apply egress admission control for lossy traffic. To bypass ingress admission control for lossy traffic, we configure a sky-high static threshold 24 MB (`static_th` of `ingress_lossy_profile`) for each ingress lossy queue. Since lossy traffic can only use 18 MB shared buffer space of `ingress_pool`, the size of `egress_lossy_pool` should be no larger than 18 MB (`size` of `ingress_pool`). In Listing 1, the `size` of `egress_lossy_pool` is 14 MB. This guarantees that ingress lossless queues can exclusively use 4 MB shared buffer (`size` of `ingress_pool` - `size` of `egress_lossy_pool`) in `ingress_pool` before entering the paused state. We use DT algorithm to manage the egress lossy queue length and set α to 1/2 ($2^{\texttt{dynamic\_th}}$). Once the egress lossy queue hits the dynamic threshold, its arriving packets will be dropped.

B DCQCN experiment results

We conduct an experiment in our lab testbed to demonstrate the RTT fairness of DCQCN. Our lab testbed uses a four-tier Clos topology like Figure 2. We use 80 km cables to interconnect T2 switches to a RH switch to emulate a region.

In this experiment, we use two hosts A and B as senders and a host C as the receiver. Each host is equipped with a Gen1 40 Gbps NIC. Host A and C are located within the same rack with $\sim$2 μs base RTT. In contrast, B is in another datacenter. The base RTT across the RH switch is $\sim$1.77 ms. On each sender, we use `ndperf` to create a QP with the receiver and keep posting 64 KB `Write` messages. Each QP can keep up to 160 in-flight `Write` messages, resulting in around 10 MB in-flight data, which is enough to saturate the large inter-datacenter pipe (40 Gbps $\times$ 1.77 ms = 8.85 MB). We set RED/ECN marking parameters K_{min}, K_{max} and P_{max} to 1 MB, 2 MB and 5%, respectively.

As shown in Figure 12, two DCQCN flows achieve similar goodput regardless of their RTTs. A flow can achieve around 17 Gbps goodput, which is close to half of the line rate. We also keep polling queue watermark counters at the congested switch and find queue watermarks oscillate around 1.36 MB,

which is smaller than K_{max}. This experiment demonstrates that DCQCN does not suffer from RTT unfairness.