Datacenter RPCs can be General and Fast

Anuj Kalia, ***Carnegie Mellon University;*** **Michael Kaminsky,** ***Intel Labs;***
David Andersen, ***Carnegie Mellon University***

https://www.usenix.org/conference/nsdi19/presentation/kalia

This paper is included in the Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19).

February 26–28, 2019 • Boston, MA, USA

ISBN 978-1-931971-49-2

Open access to the Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19) is sponsored by

NetApp®

Datacenter RPCs can be General and Fast

Anuj Kalia Michael Kaminsky[†] David G. Andersen
Carnegie Mellon University *[†]Intel Labs*

Abstract

It is commonly believed that datacenter networking software must sacrifice generality to attain high performance. The popularity of specialized distributed systems designed specifically for niche technologies such as RDMA, lossless networks, FPGAs, and programmable switches testifies to this belief. In this paper, we show that such specialization is not necessary. eRPC is a new general-purpose remote procedure call (RPC) library that offers performance comparable to specialized systems, while running on commodity CPUs in traditional datacenter networks based on either lossy Ethernet or lossless fabrics. eRPC performs well in three key metrics: message rate for small messages; bandwidth for large messages; and scalability to a large number of nodes and CPU cores. It handles packet loss, congestion, and background request execution. In microbenchmarks, one CPU core can handle up to 10 million small RPCs per second, or send large messages at 75 Gbps. We port a production-grade implementation of Raft state machine replication to eRPC without modifying the core Raft source code. We achieve 5.5 µs of replication latency on lossy Ethernet, which is faster than or comparable to specialized replication systems that use programmable switches, FPGAs, or RDMA.

1 Introduction

"Using performance to justify placing functions in a low-level subsystem must be done carefully. Sometimes, by examining the problem thoroughly, the same or better performance can be achieved at the high level."

— End-to-end Arguments in System Design

Squeezing the best performance out of modern, high-speed datacenter networks has meant painstaking specialization that breaks down the abstraction barriers between software and hardware layers. The result has been an explosion of co-designed distributed systems that depend on niche network technologies, including RDMA [18, 25, 26, 38, 50, 51, 58, 64, 66, 69], lossless networks [39, 47], FPGAs [33, 34], and programmable switches [37]. Add to that new distributed protocols with incomplete specifications, the inability to reuse existing software, hacks to enable consistent views of remote memory—and the typical developer is likely to give up and just use kernel-based TCP.

These specialized technologies were deployed with the belief that placing their functionality in the network will yield a large performance gain. In this paper, we show that a general-purpose RPC library called eRPC can provide state-of-the-art performance on commodity datacenter networks without additional network support. This helps inform the debate about the utility of additional in-network functionality vs purely end-to-end solutions for datacenter applications.

eRPC provides three key performance features: high message rate for small messages; high bandwidth for large messages; and scalability to a large number of nodes and CPU cores. It handles packet loss, node failures, congestion control, and long-running background requests. eRPC is *not* an RDMA-based system: it works well with only UDP packets over lossy Ethernet without Priority Flow Control (PFC), although it also supports InfiniBand. Our goal is to allow developers to use eRPC in unmodified systems. We use as test-cases two existing systems: a production-grade implementation of Raft [14, 54] that is used in Intel's distributed object store [11], and Masstree [49]. We successfully integrate eRPC support with both without sacrificing performance.

The need for eRPC arises because the communication software options available for datacenter networks leave much to be desired. The existing options offer an undesirable trade-off between performance and generality. Low-level interfaces such as DPDK [24] are fast, but lack features required by general applications (e.g., DPDK provides only unreliable packet I/O.) On the other hand, full-fledged networking stacks such as mTCP [35] leave significant performance on the table. Absent networking options that provide both high performance and generality, recent systems often choose to design and implement their own communication layer using low-level interfaces [18, 25, 26, 38, 39, 55, 58, 66].

The goal of our work is to answer the question: Can a general-purpose RPC library provide performance comparable to specialized systems? Our solution is based on two key insights. First, we optimize for the common case, i.e., when messages are small [16, 56], the network is congestion-free, and RPC handlers are short. Handling large messages, congestion, and long-running RPC handlers requires expensive code paths, which eRPC avoids whenever possible. Several eRPC components, including its API, message format, and wire protocol are optimized for the common case. Second, restricting each flow to at most one bandwidth-delay product (BDP) of outstanding data effectively prevents packet loss caused by switch buffer overflow for common traffic patterns. This is because datacenter switch buffers are much larger than the network's BDP. For example, in our two-

layer testbed that resembles real deployments, each switch has 12 MB of dynamic buffer, while the BDP is only 19 kB.

eRPC (*efficient* RPC) is available at https://github.com/efficient/eRPC. Our research contributions are:

1. We describe the design and implementation of a high-performance RPC library for datacenter networks. This includes (1) common-case optimizations that improve eRPC's performance for our target workloads by up to 66%; (2) techniques that enable zero-copy transmission in the presence of retransmissions, node failures, and rate limiting; and (3) a scalable implementation whose NIC memory footprint is independent of the number of nodes in the cluster.
2. We are the first to show experimentally that state-of-the-art networking performance can be achieved without lossless fabrics. We show that eRPC performs well in a 100-node cluster with lossy Ethernet without PFC. Our microbenchmarks on two lossy Ethernet clusters show that eRPC can: (1) provide 2.3 µs median RPC latency; (2) handle up to 10 million RPCs per second with one core; (3) transfer large messages at 75 Gbps with one core; (4) maintain low switch queueing during incast; and (5) maintain peak performance with 20000 connections per node (two million connections cluster-wide).
3. We show that eRPC can be used as a high-performance drop-in networking library for existing software. Notably, we implement a replicated in-memory key-value store with a production-grade version of Raft [14, 54] without modifying the Raft source code. Our three-way replication latency on lossy Ethernet is 5.5 µs, which is competitive with existing specialized systems that use programmable switches (NetChain [37]), FPGAs [33], and RDMA (DARE [58]).

2 Background and motivation

We first discuss aspects of modern datacenter networks relevant to eRPC. Next, we discuss limitations of existing networking software that underlie the need for eRPC.

2.1 High-speed datacenter networking

Modern datacenter networks provide tens of Gbps per-port bandwidth and a few microseconds round-trip latency [73, §2.1]. They support polling-based network I/O from userspace, eliminating interrupts and system call overhead from the datapath [28, 29]. eRPC uses userspace networking with polling, as in most prior high-performance networked systems [25, 37, 39, 56].

eRPC works well in commodity, lossy datacenter networks. We found that restricting each flow to one BDP of outstanding data prevents most packet drops even on lossy networks. We discuss these aspects below.

Lossless fabrics. Lossless packet delivery is a link-level feature that prevents congestion-based packet drops. For example, PFC for Ethernet prevents a link's sender from overflowing the receiver's buffer by using pause frames. Some datacenter operators, including Microsoft, have deployed PFC at scale. This was done primarily to support RDMA, since existing RDMA NICs perform poorly in the presence of packet loss [73, §1]. Lossless fabrics are useful even without RDMA: Some systems that do not use remote CPU bypass leverage losslessness to avoid the complexity and overhead of handling packet loss in software [38, 39, 47].

Unfortunately, PFC comes with a host of problems, including head-of-line blocking, deadlocks due to cyclic buffer dependencies, and complex switch configuration; Mittal et al. [53] discuss these problems in detail. In our experience, datacenter operators are often unwilling to deploy PFC due to these problems. Using simulations, Mittal et al. show that a new RDMA NIC architecture called IRN with improved packet loss handling can work well in lossy networks. Our BDP flow control is inspired by their work; the differences between eRPC's and IRN's transport are discussed in Section 5.2.3. Note that, unlike IRN, eRPC is a real system, and it does not require RDMA NIC support.

Switch buffer ≫ BDP. The increase in datacenter bandwidth has been accompanied by a corresponding decrease in round-trip time (RTT), resulting in a small BDP. Switch buffers have grown in size, to the point where "shallow-buffered" switches that use SRAM for buffering now provide tens of megabytes of shared buffer. Much of this buffer is dynamic, i.e., it can be dedicated to an incast's target port, preventing packet drops from buffer overflow. For example, in our two-layer 25 GbE testbed that resembles real datacenters (Table 1), the RTT between two nodes connected to different top-of-rack (ToR) switches is 6 µs, so the BDP is 19 kB. This is unsurprising: for example, the BDP of the two-tier 10 GbE datacenter used in pFabric is 18 kB [15].

In contrast to the small BDP, the Mellanox Spectrum switches in our cluster have 12 MB in their dynamic buffer pool [13]. Therefore, the switch can ideally tolerate a 640-way incast. The popular Broadcom Trident-II chip used in datacenters at Microsoft and Facebook has a 9 MB dynamic buffer [9, 73]. Zhang et al. [70] have made a similar observation (i.e., buffer ≫ BDP) for gigabit Ethernet.

In practice, we wish to support approximately 50-way incasts: congestion control protocols deployed in real datacenters are tested against comparable incast degrees. For example, DCQCN and Timely use up to 20- and 40-way incasts, respectively [52, 73]. This is much smaller than 640, allowing substantial tolerance to technology variations, i.e., we expect the switch buffer to be large enough to prevent most packet drops in datacenters with different BDPs and switch buffer sizes. Nevertheless, it is unlikely that the BDP-to-buffer ratio will grow substantially in the near future: newer 100 GbE switches have even larger buffers (42 MB in Mellanox's Spectrum-2 and 32 MB in Broadcom's Trident-III), and NIC-added latency is continuously decreasing. For ex-

ample, we measured InfiniBand's RTT between nodes under different ToR's to be only 3.1 µs, and Ethernet has historically caught up with InfiniBand's performance.

2.2 Limitations of existing options

Two reasons underlie our choice to design a new general-purpose RPC system for datacenter networks: First, existing datacenter networking software options sacrifice performance or generality, preventing unmodified applications from using the network efficiently. Second, co-designing storage software with the network is increasingly popular, and is largely seen as necessary to achieve maximum performance. However, such specialization has well-known drawbacks, which can be avoided with a general-purpose communication layer that also provides high performance. We describe a representative set of currently available options and their limitations below, roughly in order of increasing performance and decreasing generality.

Fully-general networking stacks such as mTCP [35] and IX [17] allow legacy sockets-based applications to run unmodified. Unfortunately, they leave substantial performance on the table, especially for small messages. For example, one server core can handle around 1.5 million and 10 million 64 B RPC requests per second with IX [17] and eRPC, respectively.

Some recent RPC systems can perform better, but are designed for specific use cases. For example, RAMCloud RPCs [56] are designed for low latency, but not high throughput. In RAMCloud, a single dispatch thread handles all network I/O, and request processing is done by other worker threads. This requires inter-thread communication for every request, and limits the system's network throughput to one core. FaRM RPCs [25] use RDMA writes over connection-based hardware transports, which limits scalability and prevents use in non-RDMA environments.

Like eRPC, our prior work on FaSST RPCs [39] uses only datagram packet I/O, but requires a lossless fabric. FaSST RPCs do not handle packet loss, large messages, congestion, long-running request handlers, or node failure; researchers have believed that supporting these features in software (instead of NIC hardware) would substantially degrade performance [27]. We show that with careful design, we can support all these features and still match FaSST's performance, while running on a lossy network. This upends conventional wisdom that losslessness or NIC support is necessary for high performance.

2.3 Drawbacks of specialization

Co-designing distributed systems with network hardware is a well-known technique to improve performance. Co-design with RDMA is popular, with numerous examples from key-value stores [25, 38, 50, 65, 66], state machine replication [58], and transaction processing systems [21, 26, 41, 66]. Programmable switches allow in-network optimizations such as reducing network round trips for distributed protocols [37, 43, 44], and in-network caching [36]. Co-design with FPGAs is an emerging technique [33].

While there are advantages of co-design, such specialized systems are unfortunately very difficult to design, implement, and deploy. Specialization breaks abstraction boundaries between components, which prevents reuse of components and increases software complexity. Building distributed storage systems requires tremendous programmer effort, and co-design typically mandates starting from scratch, with new data structures, consensus protocols, or transaction protocols. Co-designed systems often cannot reuse existing codebases or protocols, tests, formal specifications, programmer hours, and feature sets. Co-design also imposes deployment challenges beyond needing custom hardware: for example, using programmable switches requires user control over shared network switches, which may not be allowed by datacenter operators; and, RDMA-based systems are unusable with current NICs in datacenters that do not support PFC.

In several cases, specialization does not provide even a performance advantage. Our prior work shows that RPCs outperform RDMA-based designs for applications like key-value stores and distributed transactions, with the same amount of CPU [38, 39]. This is primarily because operations in these systems often require multiple remote memory accesses that can be done with one RPC, but require multiple RDMAs. In this paper (§ 7.1), we show that RPCs perform comparably with switch- and FPGA-based systems for replication, too.

3 eRPC overview

We provide an overview of eRPC's API and threading model below. In these aspects, eRPC is similar to existing high-performance RPC systems like Mellanox's Accelio [4] and FaRM. eRPC's threading model differs in how we sometimes run long-running RPC handlers in "worker" threads (§ 3.2).

eRPC implements RPCs on top of a transport layer that provides basic unreliable packet I/O, such as UDP or InfiniBand's Unreliable Datagram transport. A userspace NIC driver is required for good performance. Our primary contribution is the design and implementation of end-host mechanisms and a network transport (e.g., wire protocol and congestion control) for the commonly-used RPC API.

3.1 RPC API

RPCs execute at most once, and are asynchronous to avoid stalling on network round trips; intra-thread concurrency is provided using an event loop. RPC servers register request handler functions with unique request types; clients use these request types when issuing RPCs, and get continuation callbacks on RPC completion. Users store RPC messages in opaque, DMA-capable buffers provided by eRPC, called msgbufs; a library that provides marshalling and unmarshalling can be used as a layer on top of eRPC.

Each user thread that sends or receives RPCs creates an exclusive `Rpc` endpoint (a C++ object). Each `Rpc` endpoint contains an RX and TX queue for packet I/O, an event loop, and several *sessions*. A session is a one-to-one connection between two `Rpc` endpoints, i.e., two user threads. The client endpoint of a session is used to send requests to the user thread at the other end. A user thread may participate in multiple sessions, possibly playing different roles (i.e., client or server) in different sessions.

User threads act as "dispatch" threads: they must periodically run their `Rpc` endpoint's event loop to make progress. The event loop performs the bulk of eRPC's work, including packet I/O, congestion control, and management functions. It invokes request handlers and continuations, and dispatches long-running request handlers to worker threads (§ 3.2).

Client control flow: `rpc->enqueue_request()` queues a request msgbuf on a session, which is transmitted when the user runs `rpc`'s event loop. On receiving the response, the event loop copies it to the client's response msgbuf and invokes the continuation callback.

Server control flow: The event loop of the `Rpc` that owns the server session invokes (or dispatches) a request handler on receiving a request. We allow *nested* RPCs, i.e., the handler need not enqueue a response before returning. It may issue its own RPCs and call `enqueue_response()` for the first request later when all dependencies complete.

3.2 Worker threads

A key design decision for an RPC system is which thread runs an RPC handler. Some RPC systems such as RAMCloud use dispatch threads for only network I/O. RAMCloud's dispatch threads communicate with *worker* threads that run request handlers. At datacenter network speeds, however, inter-thread communication is expensive: it reduces throughput and adds up to 400 ns to request latency [56]. Other RPC systems such as Accelio and FaRM avoid this overhead by running all request handlers directly in dispatch threads [25, 38]. This latter approach suffers from two drawbacks when executing long request handlers: First, such handlers block other dispatch processing, increasing tail latency. Second, they prevent rapid server-to-client congestion feedback, since the server might not send packets while running user code.

Striking a balance, eRPC allows running request handlers in both dispatch threads and worker threads: When registering a request handler, the programmer specifies whether the handler should run in a dispatch thread. This is the only additional user input required in eRPC. In typical use cases, handlers that require up to a few hundred nanoseconds use dispatch threads, and longer handlers use worker threads.

3.3 Evaluation clusters

Table 1 shows the clusters used in this paper. They include two types of networks (lossy Ethernet, and lossless InfiniBand), and three generations of NICs released between 2011 (CX3) and 2017 (CX5). eRPC works well on all three clusters, showing that our design is robust to NIC and network technology changes. We use traditional UDP on the Ethernet clusters (i.e., we do not use RoCE), and InfiniBand's Unreliable Datagram transport on the InfiniBand cluster.

Currently, eRPC is primarily optimized for Mellanox NICs. eRPC also works with DPDK-capable NICs that support flow steering. For Mellanox Ethernet NICs, we generate UDP packets directly with `libibverbs` instead of going through DPDK, which internally uses `libibverbs` for these NICs.

Our evaluation primarily uses the large CX4 cluster, which resembles real-world datacenters. The ConnectX-4 NICs used in CX4 are widely deployed in datacenters at Microsoft and Facebook [3, 73], and its Mellanox Spectrum switches perform similarly to Broadcom's Trident switches used in these datacenters (i.e., both switches provide dynamic buffering, cut-through switching, and less than 500 ns port-to-port latency.) We use 100 nodes out of the 200 nodes in the shared CloudLab cluster. The six switches in the CX4 cluster are organized as five ToRs with 40 25 GbE downlinks and five 100 GbE uplinks, for a 2:1 oversubscription.

4 eRPC design

Achieving eRPC's performance goals requires careful design and implementation. We discuss three aspects of eRPC's design in this section: scalability of our networking primitives, the challenges involved in supporting zero-copy, and the design of sessions. The next section discusses eRPC's wire protocol and congestion control. A recurring theme in eRPC's design is that we optimize for the common case, i.e., when request handlers run in dispatch threads, RPCs are small, and the network is congestion-free.

4.1 Scalability considerations

We chose plain packet I/O instead of RDMA writes [25, 66, 69] to send messages in eRPC. This decision is based on prior insights from our design of FaSST: First, packet I/O provides completion queues that can scalably detect received packets. Second, RDMA caches connection state in NICs, which does not scale to large clusters. We next discuss *new* observations about NIC hardware trends that support this design.

4.1.1 Packet I/O scales well

RPC systems that use RDMA writes have a *fundamental* scalability limitation. In these systems, clients write requests directly to per-client circular buffers in the server's memory; the server must poll these buffers to detect new requests. The number of circular buffers grows with the number of clients, limiting scalability.

With traditional userspace packet I/O, the NIC writes an incoming packet's payload to a buffer specified by a descriptor pre-posted to the NIC's RX queue (RQ) by the receiver host; the packet is dropped if the RQ is empty. Then, the

Name	Nodes	Network type	Mellanox NIC	Switches	Intel Xeon E5 CPU code
CX3	11	InfiniBand	56 Gbps ConnectX-3	One SX6036	2650 (8 cores)
CX4	100	Lossy Ethernet	25 Gbps ConnectX-4 Lx	5x SN2410, 1x SN2100	2640 v4 (10 cores)
CX5	8	Lossy Ethernet	Dual-port 40 Gbps ConnectX-5	One SX1036	2697 v3 (14 c) or 2683 v4 (16 c)

Table 1: Measurement clusters. CX4 and CX3 are CloudLab [59] and Emulab [68] clusters, respectively.

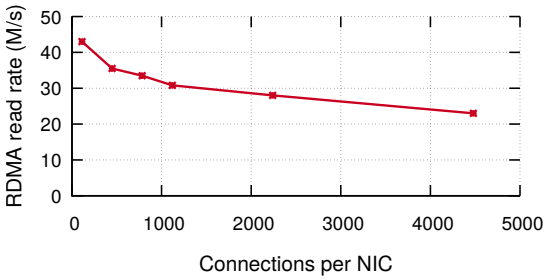

Figure 1: Connection scalability of ConnectX-5 NICs

NIC writes an entry to the host's RX completion queue. The receiver host can then check for received packets in constant time by examining the head of the completion queue.

To avoid dropping packets due to an empty RQ with no descriptors, RQs must be sized proportionally to the number of independent connected RPC endpoints (§ 4.3.1). Older NICs experience cache thrashing with large RQs, thus limiting scalability, but we find that newer NICs fare better: While a Connect-IB NIC could support only 14 2K-entry RQs before thrashing [39], we find that ConnectX-5 NICs do not thrash even with 28 64K-entry RQs. This improvement is due to more intelligent prefetching and caching of RQ descriptors, instead of a massive 64x increase in NIC cache.

We use features of current NICs (e.g., multi-packet RQ descriptors that identify several contiguous packet buffers) in novel ways to guarantee a *constant* NIC memory footprint per CPU core, i.e., it does not depend on the number of nodes in the cluster. This result can simplify the design of future NICs (e.g., RQ descriptor caching is unneeded), but its current value is limited to performance improvements because current NICs support very large RQs, and are perhaps overly complex as a result. We discuss this in detail in Appendix A.

4.1.2 Scalability limits of RDMA

RDMA requires NIC-managed connection state. This limits scalability because NICs have limited SRAM to cache connection state. The number of in-NIC connections may be reduced by sharing them among CPU cores, but doing so reduces performance by up to 80% [39].

Some researchers have hypothesized that improvements in NIC hardware will allow using connected transports at large scale [27, 69]. To show that this is unlikely, we measure the connection scalability of state-of-the-art ConnectX-5 NICs, released in 2017. We repeat the connection scalability experiment from FaSST, which was used to evaluate the older Connect-IB NICs from 2012. We enable PFC on CX5 for this

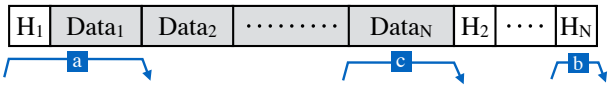

Figure 2: Layout of packet headers and data for an N-packet msgbuf. Blue arrows show NIC DMAs; the letters show the order in which the DMAs are performed for packets 1 and N.

experiment since it uses RDMA; PFC is disabled in all experiments that use eRPC. In the experiment, each node creates a tunable number of connections to other nodes and issues 16-byte RDMA reads on randomly-chosen connections. Figure 1 shows that as the number of connections increases, RDMA throughput decreases, losing ≈50% throughput with 5000 connections. This happens because NICs can cache only a few connections, and cache misses require expensive DMA reads [25]. In contrast, eRPC maintains its peak throughput with 20000 connections (§ 6.3).

ConnectX-5's connection scalability is, surprisingly, not substantially better than Connect-IB despite the five-year advancement. A simple calculation shows why this is hard to improve: In Mellanox's implementation, each connection requires ≈375 B of in-NIC connection state, and the NICs have ≈2 MB of SRAM to store connection state as well as other data structures and buffers [1]. 5000 connections require 1.8 MB, so cache misses are unavoidable.

NIC vendors have been trying to improve RDMA's scalability for a decade [22, 42]. Unfortunately, these techniques do not map well to RPC workloads [39]. Vendors have not put more memory in NICs, probably because of cost and power overheads, and market factors. The scalability issue of RDMA is exacerbated by the popularity of *multihost* NICs, which allow sharing a powerful NIC among 2–4 CPUs [3, 7].

eRPC replaces NIC-managed connection state with CPU-managed connection state. This is an explicit design choice, based upon fundamental differences between the CPU and NIC architectures. NICs and CPUs will both cache recently-used connection state. CPU cache misses are served from DRAM, whereas NIC cache misses are served from the CPU's memory subsystem over the slow PCIe bus. The CPU's miss penalty is therefore much lower. Second, CPUs have substantially larger caches than the ~2 MB available on a modern NIC, so the cache miss *frequency* is also lower.

4.2 Challenges in zero-copy transmission

eRPC uses zero-copy packet I/O to provide performance comparable to low-level interfaces such as DPDK and RDMA. This section describes the challenges involved in doing so.

4.2.1 Message buffer layout

eRPC provides DMA-capable message buffers to applications for zero-copy transfers. A msgbuf holds one, possibly multi-packet message. It consists of per-packet headers and data, arranged in a fashion optimized for small single-packet messages (Figure 2). Each eRPC packet has a header that contains the transport header, and eRPC metadata such as the request handler type and sequence numbers. We designed a msgbuf layout that satisfies two requirements.

1. The data region is contiguous to allow its use in applications as an opaque buffer.
2. The first packet's data and header are contiguous. This allows the NIC to fetch small messages with one DMA read; using multiple DMAs for small messages would substantially increase NIC processing and PCIe use, reducing message rate by up to 20% [40].

For multi-packet messages, headers for subsequent packets are at the end of the message: placing header 2 immediately after the first data packet would violate our first requirement. Non-first packets require two DMAs (header and data); this is reasonable because the overhead for DMA-reading small headers is amortized over the large data DMA.

4.2.2 Message buffer ownership

Since eRPC transfers packets directly from application-owned msgbufs, msgbuf references must never be used by eRPC after msgbuf ownership is returned to the application. In this paper, we discuss msgbuf ownership issues for only clients; the process is similar but simpler for the server, since eRPC's servers are passive (§ 5). At clients, we must ensure the following invariant: *no eRPC transmission queue contains a reference to the request msgbuf when the response is processed.* Processing the response includes invoking the continuation, which permits the application to reuse the request msgbuf. In eRPC, a request reference may be queued in the NIC's hardware DMA queue, or in our software rate limiter (§ 5.2).

This invariant is maintained trivially when there are no retransmissions or node failures, since the request must exit all transmission queues before the response is received. The following **example** demonstrates the problem with retransmissions. Consider a client that falsely suspects packet loss and retransmits its request. The server, however, received the first copy of the request, and its response reaches the client before the retransmitted request is dequeued. Before processing the response and invoking the continuation, we must ensure that there are no queued references to the request msgbuf. We discuss our solution for the NIC DMA queue next, and for the rate limiter in Appendix C.

The conventional approach to ensure DMA completion is to use "signaled" packet transmission, in which the NIC writes completion entries to the TX completion queue. Unfortunately, doing so reduces message rates by up to 25% by using more NIC and PCIe resources [38], so we use unsignaled packet transmission in eRPC.

Our method of ensuring DMA completion with unsignaled transmission is in line with our design philosophy: we choose to make the common case (no retransmission) fast, at the expense of invoking a more-expensive mechanism to handle the rare cases. We flush the TX DMA queue after queueing a retransmitted packet, which blocks until all queued packets are DMA-ed. This ensures the required invariant: when a response is processed, there are no references to the request in the DMA queue. This flush is moderately expensive (≈2 µs), but it is called during rare retransmission or node failure events, and it allows eRPC to retain the 25% throughput increase from unsignaled transmission.

During server node failures, eRPC invokes continuations with error codes, which also yield request msgbuf ownership. It is possible, although extremely unlikely, that server failure is suspected while a request (not necessarily a retransmission) is in the DMA queue or the rate limiter. Handling node failures requires similar care as discussed above, and is discussed in detail in Appendix B.

4.2.3 Zero-copy request processing

Zero-copy reception is harder than transmission: To provide a contiguous request msgbuf to the request handler at the server, we must strip headers from received packets, and copy only application data to the target msgbuf. However, we were able to provide zero-copy reception for our common-case workload consisting of single-packet requests and dispatch-mode request handlers as follows. eRPC owns the packet buffers DMA-ed by the NIC until it re-adds the descriptors for these packets back to the receive queue (i.e., the NIC cannot modify the packet buffers for this period.) This ownership guarantee allows running dispatch-mode handlers without copying the DMA-ed request packet to a dynamically-allocated msgbuf. Doing so improves eRPC's message rate by up to 16% (§ 6.2).

4.3 Sessions

Each session maintains multiple outstanding requests to keep the network pipe full. Concurrently requests on a session can complete *out-of-order* with respect to each other. This avoids blocking dispatch-mode RPCs behind a long-running worker-mode RPC. We support a constant number of concurrent requests (default = 8) per session; additional requests are transparently queued by eRPC. This is inspired by how RDMA connections allow a constant number of operations [10]. A session uses an array of *slots* to track RPC metadata for outstanding requests.

Slots in server-mode sessions have an MTU-size preallocated msgbuf for use by request handlers that issue short responses. Using the preallocated msgbuf does not require user input: eRPC chooses it automatically at run time by examining the handler's desired response size. This opti-

mization avoids the overhead of dynamic memory allocation, and improves eRPC's message rate by up to 13% (§ 6.2).

4.3.1 Session credits

eRPC limits the number of unacknowledged packets on a session for two reasons. First, to avoid dropping packets due to an empty RQ with no descriptors, the number of packets that may be sent to an Rpc must not exceed the size of its RQ ($|RQ|$). Because each session sends packets independently of others, we first limit the number of sessions that an Rpc can participate in. Each session then uses *session credits* to implement packet-level flow control: we limit the number of packets that a client may send on a session before receiving a reply, allowing the server Rpc to replenish used RQ descriptors before sending more packets.

Second, session credits automatically implement end-to-end flow control, which reduces switch queueing (§ 5.2). Allowing BDP/MTU credits per session ensures that each session can achieve line rate. Mittal et al. [53] have proposed similar flow control for RDMA NICs (§ 5.2.3).

A client session starts with a quota of C packets. Sending a packet to the server consumes a credit, and receiving a packet replenishes a credit. An Rpc can therefore participate in up to $|RQ|/C$ sessions, counting both server-mode and client-mode sessions; session creation fails after this limit is reached. We plan to explore statistical multiplexing in the future.

4.3.2 Session scalability

eRPC's scalability depends on the user's desired value of C, and the number and size of RQs that the NIC and host can effectively support. Lowering C increases scalability, but reduces session throughput by restricting the session's packet window. Small values of C (e.g., $C = 1$) should be used in applications that (a) require only low latency and small messages, or (b) whose threads participate in many sessions. Large values (e.g., BDP/MTU) should be used by applications whose sessions individually require high throughput.

Modern NICs can support several very large RQs, so NIC RQ capacity limits scalability only on older NICs. In our evaluation, we show that eRPC can handle 20000 sessions with 32 credits per session on the widely-used ConnectX-4 NICs. However, since each RQ entry requires allocating a packet buffer in host memory, needlessly large RQs waste host memory and should be avoided.

5 Wire protocol

We designed a wire protocol for eRPC that is optimized for small RPCs and accounts for per-session credit limits. For simplicity, we chose a simple *client-driven* protocol, meaning that each packet sent by the server is in response to a client packet. A client-driven protocol has fewer "moving parts" than a protocol in which both the server and client can independently send packets. Only the client maintains wire protocol state that is rolled back during retransmission. This removes the need for client-server coordination before rollback, reducing complexity. A client-driven protocol also shifts the overhead of rate limiting entirely to clients, freeing server CPU that is often more valuable.

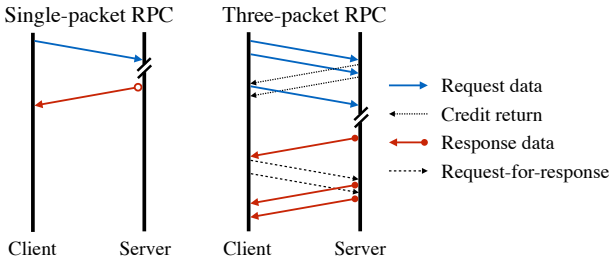

Figure 3: Examples of eRPC's wire protocol, with 2 credits/session.

5.1 Protocol messages

Figure 3 shows the packets sent with $C = 2$ for a small single-packet RPC, and for an RPC whose request and response require three packets each. Single-packet RPCs use the fewest packets possible. The client begins by sending a window of up to C request data packets. For each request packet except the last, the server sends back an explicit *credit return* (CR) packet; the credit used by the last request packet is implicitly returned by the first response packet.

Since the protocol is client-driven, the server cannot immediately send response packets after the first. Subsequent response packets are triggered by *request-for-response* (RFR) packets that the client sends after receiving the first response packet. This increases the latency of multi-packet responses by up to one RTT. This is a fundamental drawback of client-driven protocols; in practice, we found that the added latency is less than 20% for responses with four or more packets.

CRs and RFRs are tiny 16 B packets, and are sent only for large multi-packet RPCs. The additional overhead of sending these tiny packets is small with userspace networking that our protocol is designed for, so we do not attempt complex optimizations such as cumulative CRs or RFRs. These optimizations may be worthwhile for kernel-based networking stacks, where sending a 16 B packet and an MTU-sized packet often have comparable CPU cost.

5.2 Congestion control

Congestion control for datacenter networks aims to reduce switch queueing, thereby preventing packet drops and reducing RTT. Prior high-performance RPC implementations such as FaSST do not implement congestion control, and some researchers have hypothesized that doing so will substantially reduce performance [27]. Can effective congestion control be implemented efficiently in software? We show that optimizing for uncongested networks, and recent advances in software rate limiting allow congestion control with only 9% overhead (§ 6.2).

5.2.1 Available options

Congestion control for high-speed datacenter networks is an evolving area of research, with two major approaches for commodity hardware: RTT-based approaches such as Timely [52], and ECN-based approaches such as DCQCN [73]. Timely and DCQCN have been deployed at Google and Microsoft, respectively. We wish to use these protocols since they have been shown to work at scale.

Both Timely and DCQCN are rate-based: client use the congestion signals to adjust per-session sending rates. We implement Carousel's rate limiter [61], which is designed to efficiently handle a large number of sessions. Carousel's design works well for us as-is, so we omit the details.

eRPC includes the hooks and mechanisms to easily implement either Timely or DCQCN. Unfortunately, we are unable to implement DCQCN because none of our clusters performs ECN marking[1]. Timely can be implemented entirely in software, which made it our favored approach. eRPC runs all three Timely components—per-packet RTT measurement, rate computation using the RTT measurements, and rate limiting—at client session endpoints. For `Rpc`'s that host only server-mode endpoints, there is no overhead due to congestion control.

5.2.2 Common-case optimizations

We use three optimizations for our common-case workloads. Our evaluation shows that these optimizations reduce the overhead of congestion control from 20% to 9%, and that they do not reduce the effectiveness of congestion control. The first two are based on the observation that datacenter networks are typically uncongested. Recent studies of Facebook's datacenters support this claim: Roy et al. [60] report that 99% of all datacenter links are less than 10% utilized at one-minute timescales. Zhang et al. [71, Fig. 6] report that for Web and Cache traffic, 90% of top-of-rack switch links, which are the most congested switches, are less than 10% utilized at 25 µs timescales.

When a session is uncongested, RTTs are low and Timely's computed rate for the session stays at the link's maximum rate; we refer to such sessions as *uncongested*.

1. **Timely bypass.** If the RTT of a packet received on an uncongested session is smaller than Timely's low threshold, below which it performs additive increase, we do not perform a rate update. We use the recommended value of 50 µs for the low threshold [52, 74].
2. **Rate limiter bypass.** For uncongested sessions, we transmit packets directly instead of placing them in the rate limiter.
3. **Batched timestamps for RTT measurement.** Calling `rdtsc()` costs 8 ns on our hardware, which is substantial when processing millions of small packets per second. We reduce timer overhead by sampling it once per RX or TX batch instead of once per packet.

[1]The Ethernet switch in our private CX5 cluster does not support ECN marking [5, p. 839]; we do not have admin access to the shared CloudLab switches in the public CX4 cluster; and InfiniBand NICs in the CX3 cluster do not relay ECN marks to software.

5.2.3 Comparison with IRN

IRN [53] is a new RDMA NIC architecture designed for lossy networks, with two key improvements. First, it uses BDP flow control to limit the outstanding data per RDMA connection to one BDP. Second, it uses efficient selective acks instead of simple go-back-N for packet loss recovery.

IRN was evaluated with simulated switches that have small (60–480 kB) static, per-port buffers. In this buffer-deficient setting, they found SACKs necessary for good performance. However, dynamic-buffer switches are the de-facto standard in current datacenters. As a result, packet losses are very rare with only BDP flow control, so we currently do not implement SACKs, primarily due to engineering complexity. eRPC's dependence on dynamic switch buffers can be reduced by implementing SACK.

With small per-port switch buffers, IRN's maximum RTT is a few hundred microseconds, allowing a ∼300 µs retransmission timeout (RTO). However, the 12 MB dynamic buffer in our main CX4 cluster (25 Gbps) can add up to 3.8 ms of queueing delay. Therefore, we use a conservative 5 ms RTO.

5.3 Handling packet loss

For simplicity, eRPC treats reordered packets as losses by dropping them. This is not a major deficiency because datacenter networks typically use ECMP for load balancing, which preserves intra-flow ordering [30, 71, 72] except during rare route churn events. Note that current RDMA NICs also drop reordered packets [53].

On suspecting a lost packet, the client rolls back the request's wire protocol state using a simple go-back-N mechanism. It then reclaims credits used for the rolled-back transmissions, and retransmits from the updated state. The server never runs the request handler for a request twice, guaranteeing at-most-once RPC semantics.

In case of a false positive, a client may violate the credit agreement by having more packets outstanding to the server than its credit limit. In the extremely rare case that such an erroneous loss detection occurs *and* the server's RQ is out of descriptors, eRPC will have "induced" a real packet loss. We allow this possibility and handle the induced loss like a real packet loss.

6 Microbenchmarks

eRPC is implemented in 6200 SLOC of C++, excluding tests and benchmarks. We use static polymorphism to create an `Rpc` class that works with multiple transport types without the overhead of virtual function calls. In this section, we evaluate eRPC's latency, message rate, scalability, and bandwidth using microbenchmarks. To understand eRPC's performance in commodity datacenters, we primarily use the large CX4

Cluster	CX3 (InfiniBand)	CX4 (Eth)	CX5 (Eth)
RDMA read	1.7 µs	2.9 µs	2.0 µs
eRPC	2.1 µs	3.7 µs	2.3 µs

Table 2: Comparison of median latency with eRPC and RDMA

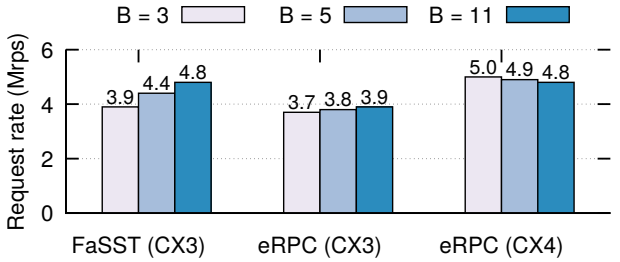

Figure 4: Single-core small-RPC rate with B requests per batch

cluster. We use CX5 and CX3 for their more powerful NICs and low-latency InfiniBand, respectively. eRPC's congestion control is enabled by default.

6.1 Small RPC latency

How much latency does eRPC add? Table 2 compares the median latency of 32 B RPCs and RDMA reads between two nodes connected to the same ToR switch. Across all clusters, eRPC is at most 800 ns slower than RDMA reads.

eRPC's median latency on CX5 is only 2.3 µs, showing that latency with commodity Ethernet NICs and software networking is much lower than the widely-believed value of 10–100 µs [37, 57]. CX5's switch adds 300 ns to every layer-3 packet [12], meaning that end-host networking adds only ≈850 ns each at the client and server. This is comparable to switch-added latency. We discuss this further in § 7.1.

6.2 Small RPC rate

What is the CPU cost of providing generality in an RPC system? We compare eRPC's small message performance against FaSST RPCs, which outperform other RPC systems such as FaRM [39]. FaSST RPCs are *specialized* for single-packet RPCs in a lossless network, and they do not handle congestion.

We mimic FaSST's experiment setting: one thread per node in an 11-node cluster, each of which acts each acts as both RPC server and client. Each thread issues batches of B requests, keeping multiple request batches in flight to hide network latency. Each request in a batch is sent to a randomly-chosen remote thread. Such batching is common in key-value stores and distributed online transaction processing. Each thread keeps up to 60 requests in flight, spread across all sessions. RPCs are 32 B in size. We compare eRPC's performance on CX3 (InfiniBand) against FaSST's reported numbers on the same cluster. We also present eRPC's performance on the CX4 Ethernet cluster. We omit CX5 since it has only 8 nodes.

Figure 4 shows that eRPC's per-thread request issue rate is at most 18% lower than FaSST across all batch sizes, and only 5% lower for $B = 3$. This performance drop is acceptable since eRPC is a full-fledged RPC system, whereas FaSST is highly specialized. On CX4, each thread issues 5 million requests per second (Mrps) for $B = 3$; due to the experiment's symmetry, it simultaneously also handles incoming requests from remote threads at 5 Mrps. Therefore, each thread processes 10 million RPCs per second.

Disabling congestion control increases eRPC's request rate on CX4 ($B = 3$) from 4.96 Mrps to 5.44 Mrps. This shows that the overhead of our optimized congestion control is only 9%.

Action	RPC rate	% loss
Baseline (with congestion control)	4.96 M/s	–
Disable batched RTT timestamps (§5.2)	4.84 M/s	2.4%
Disable Timely bypass (§5.2)	4.52 M/s	6.6%
Disable rate limiter bypass (§5.2)	4.30 M/s	4.8%
Disable multi-packet RQ (§4.1.1)	4.06 M/s	5.6%
Disable preallocated responses (§4.3)	3.55 M/s	12.6%
Disable 0-copy request processing (§4.2.3)	3.05 M/s	14.0%

Table 3: Impact of disabling optimizations on small RPC rate (CX4)

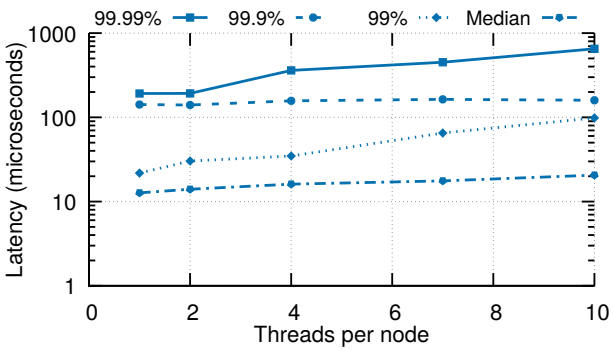

Figure 5: Latency with increasing threads on 100 CX4 nodes

Factor analysis. How important are eRPC's common-case optimizations? Table 3 shows the performance impact of *disabling* some of eRPC's common-case optimizations on CX4; other optimizations such as our single-DMA msgbuf format and unsignaled transmissions cannot be disabled easily. For our baseline, we use $B = 3$ and enable congestion control. Disabling all three congestion control optimizations (§ 5.2.2) reduces throughput to 4.3 Mrps, increasing the overhead of congestion control from 9% to 20%. Further disabling preallocated responses and zero-copy request processing reduces throughput to 3 Mrps, which is 40% lower than eRPC's peak throughput. *We therefore conclude that optimizing for the common case is both necessary and sufficient for high-performance RPCs.*

6.3 Session scalability

We evaluate eRPC's scalability on CX4 by increasing the number of nodes in the previous experiment ($B = 3$) to 100. The five ToR switches in CX4 were assigned between 14 and

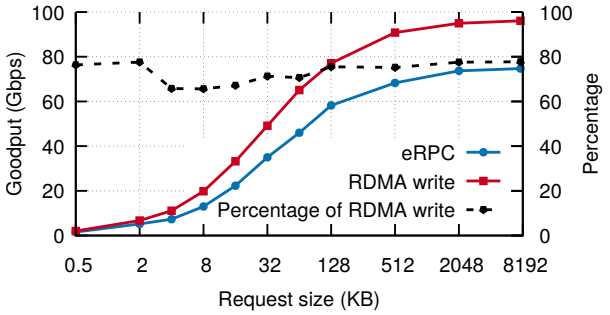

Figure 6: Throughput of large transfers over 100 Gbps InfiniBand

27 nodes each by CloudLab. Next, we increase the number of threads per node: With T threads per node, there are $100T$ threads in the cluster; each thread creates a client-mode session to $100T - 1$ threads. Therefore, each node hosts $T * (100T - 1)$ client-mode sessions, and an equal number of server-mode sessions. Since CX4 nodes have 10 cores, each node handles up to 19980 sessions. This is a challenging traffic pattern that resembles distributed online transaction processing (OLTP) workloads, which operate on small data items [26, 39, 66, 69].

With 10 threads/node, each node achieves 12.3 Mrps on average. At 12.3 Mrps, each node sends and receives 24.6 million packets per second (packet size = 92 B), corresponding to 18.1 Gbps. This is close to the link's achievable bandwidth (23 Gbps out of 25 Gbps), but is somewhat smaller because of oversubscription. We observe retransmissions with more than two threads per node, but the retransmission rate stays below 1700 packets per second per node.

Figure 5 shows the RPC latency statistics. The median latency with one thread per node is 12.7 µs. This is higher than the 3.7 µs for CX4 in Table 2 because most RPCs now go across multiple switches, and each thread keeps 60 RPCs in flight, which adds processing delay. Even with 10 threads per node, eRPC's 99.99th percentile latency stays below 700 µs.

These results show that eRPC can achieve high message rate, bandwidth, and scalability, and low latency in a large cluster with lossy Ethernet. Distributed OLTP has been a key application for lossless RDMA fabrics; our results show that it can also perform well on lossy Ethernet.

6.4 Large RPC bandwidth

We evaluate eRPC's bandwidth using a client thread that sends large messages to a remote server thread. The client sends R-byte requests and keeps one request outstanding; the server replies with a small 32 B response. We use up to 8 MB requests, which is the largest message size supported by eRPC. We use 32 credits per session. To understand how eRPC performs relative to hardware limits, we compare against R-byte RDMA writes, measured using `perftest`.

On the clusters in Table 1, eRPC gets bottlenecked by network bandwidth in this experiment setup. To understand eRPC's performance limits, we connect two nodes in the CX5 cluster to a 100 Gbps switch via ConnectX-5 InfiniBand NICs. (CX5 is used as a 40 GbE cluster in the rest of this paper.) Figure 6 shows that eRPC achieves up to 75 Gbps with one core. eRPC's throughput is at least 70% of RDMA write throughput for 32 kB or larger requests.

Loss rate	10^{-7}	10^{-6}	10^{-5}	10^{-4}	10^{-3}
Bandwidth (Gbps)	73	71	57	18	2.5

Table 4: eRPC's 8 MB request throughput with packet loss

Incast degree	**Total bw**	**50% RTT**	**99% RTT**
20	21.8 Gbps	39 µs	67 µs
20 (no cc)	23.1 Gbps	202 µs	204 µs
50	18.4 Gbps	34 µs	174 µs
50 (no cc)	23.0 Gbps	524 µs	524 µs
100	22.8 Gbps	349 µs	969 µs
100 (no cc)	23.0 Gbps	1056 µs	1060 µs

Table 5: Effectiveness of congestion control (cc) during incast

In the future, eRPC's bandwidth can be improved by freeing-up CPU cycles. First, on-die memory copy accelerators can speed up copying data from RX ring buffers to request or response msgbufs [2, 28]. Commenting out the memory copies at the server increases eRPC's bandwidth to 92 Gbps, showing that copying has substantial overhead. Second, cumulative credit return and request-for-response (§ 5.1) can reduce packet processing overhead.

Table 4 shows the throughput with R = 8 MB (the largest size supported by eRPC), and varying, artificially-injected packet loss rates. With the current 5 ms RTO, eRPC is usable while the loss probability is up to .01%, beyond which throughput degrades rapidly. We believe that this is sufficient to handle packet corruptions. RDMA NICs can handle a somewhat higher loss rate (.1%) [73].

6.5 Effectiveness of congestion control

We evaluate if our congestion control is successful at reducing switch queueing. We create an incast traffic pattern by increasing the number of client nodes in the previous setup (R = 8 MB). The one server node acts as the incast victim. During an incast, queuing primarily happens at the victim's ToR switch. We use per-packet RTTs measured at the clients as a proxy for switch queue length [52].

Table 5 shows the total bandwidth achieved by all flows and per-packet RTT statistics on CX4, for 20, 50, and 100-way incasts (one flow per client node). We use two configurations: first with eRPC's optimized congestion control, and second with no congestion control. Disabling our common-case congestion control optimizations does not substantially affect the RTT statistics, indicating that these optimizations do not reduce the quality of congestion control.

Congestion control successfully handles our target workloads of up to 50-way incasts, reducing median and 99th percentile queuing by over 5x and 3x, respectively. For 100-way incasts, our implementation reduces median queueing by 3x, but fails to substantially reduce 99th percentile queueing. This is in line with Zhu et al. [74, § 4.3]'s analysis, which shows that Timely-like protocols work well with up to approximately 40 incast flows.

The combined incast throughput with congestion control is within 20% of the achievable 23 Gbps. We believe that this small gap can be further reduced with better tuning of Timely's many parameters. Note that we can also support ECN-based congestion control in eRPC, which may be a better congestion indicator than RTT [74].

Incast with background traffic. Next, we augment the setup above to mimic an experiment from Timely [52, Fig 22]: we create one additional thread at each node that is not the incast victim. These threads exchange latency-sensitive RPCs (64 kB request and response), keeping one RPC outstanding. During a 100-way incast, the 99th percentile latency of these RPCs is 274 µs. This is similar to Timely's latency (≈200-300 µs) with a 40-way incast over a 20 GbE lossless RDMA fabric. Although the two results cannot be directly compared, this experiment shows that the latency achievable with software-only networking in commodity, lossy datacenters is comparable to lossless RDMA fabrics, even with challenging traffic patterns.

7 Full-system benchmarks

In this section, we evaluate whether eRPC can be used in real applications with unmodified existing storage software: We build a state machine replication system using an open-source implementation of Raft [54], and a networked ordered key-value store using Masstree [49].

7.1 Raft over eRPC

State machine replication (SMR) is used to build fault-tolerant services. An SMR service consists of a group of server nodes that receive commands from clients. SMR protocols ensure that each server executes the same sequence of commands, and that the service remains available if servers fail. Raft [54] is such a protocol that takes a *leader*-based approach: Absent failures, the Raft replicas have a stable leader to which clients send commands; if the leader fails, the remaining Raft servers elect a new one. The leader appends the command to replicas' logs, and it replies to the client after receiving acks from a majority of replicas.

SMR is difficult to design and implement correctly [31]: the protocol must have a specification and a proof (e.g., in TLA+), and the implementation must adhere to the specification. We avoid this difficulty by using an existing implementation of Raft [14]. (It had no distinct name, so we term it LibRaft.) We did not write LibRaft ourselves; we found it on GitHub and used it as-is. LibRaft is well-tested with fuzzing over a network simulator and 150+ unit tests. Its only requirement is that the user provide callbacks for sending and handling RPCs—which we implement using eRPC. Porting to eRPC required no changes to LibRaft's code.

Measurement	System	Median	99%
Measured at client	NetChain	9.7 µs	N/A
	eRPC	5.5 µs	6.3 µs
Measured at leader	ZabFPGA	3.0 µs	3.0 µs
	eRPC	3.1 µs	3.4 µs

Table 6: Latency comparison for replicated PUTs

We compare against recent consistent replication systems that are built from scratch for two specialized hardware types. First, NetChain [37] implements chain replication over programmable switches. Other replication protocols such as conventional primary-backup and Raft are too complex to implement over programmable switches [37]. Therefore, despite the protocol-level differences between LibRaft-over-eRPC and NetChain, our comparison helps understand the relative performance of end-to-end CPU-based designs and switch-based designs for in-memory replication. Second, Consensus in a Box [33] (called ZabFPGA here), implements ZooKeeper's atomic broadcast protocol [32] on FPGAs. eRPC also outperforms DARE [58], which implements SMR over RDMA; we omit the results for brevity.

Workloads. We mimic NetChain and ZabFPGA's experiment setups for latency measurement: we implement a 3-way replicated in-memory key-value store, and use one client to issue PUT requests. The replicas' command logs and key-value store are stored in DRAM. NetChain and ZabFPGA use 16 B keys, and 16–64 B values; we use 16 B keys and 64 B values. The client chooses PUT keys uniformly at random from one million keys. While NetChain and ZabFPGA also implement their key-value stores from scratch, we reuse existing code from MICA [45]. We compare eRPC's performance on CX5 against their published numbers because we do not have the hardware to run NetChain or ZabFPGA. Table 6 compares the latencies of the three systems.

7.1.1 Comparison with NetChain

NetChain's key assumption is that software networking adds 1–2 orders of magnitude more latency than switches [37]. However, we have shown that eRPC adds 850 ns, which is only around 2x higher than latency added by current programmable switches (400 ns [8]).

Raft's latency over eRPC is 5.5 µs, which is substantially lower than NetChain's 9.7 µs. This result must be taken with a grain of salt: On the one hand, NetChain uses NICs that have higher latency than CX5's NICs. On the other hand, it has numerous limitations, including key-value size and capacity constraints, serial chain replication whose latency increases linearly with the number of replicas, absence of

congestion control, and reliance on a complex and external failure detector. The main takeaway is that microsecond-scale consistent replication is achievable in commodity Ethernet datacenters with a general-purpose networking library.

7.1.2 Comparison with ZabFPGA

Although ZabFPGA's SMR servers are FPGAs, the clients are commodity workstations that communicate with the FPGAs over slow kernel-based TCP. For a challenging comparison, we compare against ZabFPGA's commit latency measured at the leader, which involves only FPGAs. In addition, we consider its "direct connect" mode, where FPGAs communicate over point-to-point links (i.e., without a switch) via a custom protocol. Even so, eRPC's median leader commit latency is only 3% worse.

An advantage of specialized, dedicated hardware is low jitter. This is highlighted by ZabFPGA's negligible leader latency variance. This advantage does not carry over directly to end-to-end latency [33] because storage systems built with specialized hardware are eventually accessed by clients running on commodity workstations.

7.2 Masstree over eRPC

Masstree [49] is an ordered in-memory key-value store. We use it to implement a single-node database index that supports low-latency point queries in the presence of less performance-critical longer-running scans. This requires running scans in worker threads. We use CX3 for this experiment to show that eRPC works well on InfiniBand.

We populate a Masstree server on CX3 with one million random 8 B keys mapped to 8 B values. The server has 16 Hyper-Threads, which we divide between 14 dispatch threads and 2 worker threads. We run 64 client threads spread over 8 client nodes to generate the workload. The workload consists of 99% GET(key) requests that fetch a key-value item, and 1% SCAN(key) requests that sum up the values of 128 keys succeeding the key. Keys are chosen uniformly at random from the inserted keys. Two outstanding requests per client was sufficient to saturate our server.

We achieve 14.3 million GETs/s on CX3, with 12 µs 99th percentile GET latency. If the server is configured to run only dispatch threads, the 99th percentile GET latency rises to 26 µs. eRPC's median GET latency under low load is 2.7 µs. This is around 10x faster than Cell's single-node B-Tree that uses multiple RDMA reads [51]. Despite Cell's larger key/-value sizes (64 B/256 B), the latency differences are mostly from RTTs: At 40 Gbps, an additional 248 B takes only 50 ns more time to transmit.

8 Related work

RPCs. There is a vast amount of literature on RPCs. The practice of optimizing an RPC wire protocol for small RPCs originates with Birrell and Nelson [19], who introduce the idea of an implicit-ACK. Similar to eRPC, the Sprite RPC system [67] directly uses raw datagrams and performs retransmissions only at clients. The Direct Access File System [23] was one of the first to use RDMA in RPCs. It uses SEND/RECV messaging over a connected transport to initiate an RPC, and RDMA reads or writes to transfer the bulk of large RPC messages. This design is widely used in other systems such as NFS's RPCs [20] and some MPI implementations [48]. In eRPC, we chose to transfer all data over datagram messaging to avoid the scalability limits of RDMA. Other RPC systems that use RDMA include Mellanox's Accelio [4] and RFP [63]. These systems perform comparably to FaRM's RPCs, which are slower than eRPC at scale by an order of magnitude.

Co-design. There is a rapidly-growing list of projects that co-design distributed systems with the network. This includes key-value stores [38, 46, 50, 65], distributed databases and transaction processing systems [21, 25, 66, 69], state machine replication [33, 58], and graph-processing systems [62]. We believe the availability of eRPC will motivate researchers to investigate how much performance these systems can achieve without sacrificing the networking abstraction. On the other hand, there is a smaller set of recent projects that also prefer RPCs over co-design, including RAMCloud, FaSST, and the distributed data shuffler by Liu et al. [47]. However, their RPCs lack either performance (RAMCloud) or generality (FaSST), whereas eRPC provides both.

9 Conclusion

eRPC is a fast, general-purpose RPC system that provides an attractive alternative to putting more functions in network hardware, and specialized system designs that depend on these functions. eRPC's speed comes from prioritizing common-case performance, carefully combining a wide range of old and new optimizations, and the observation that switch buffer capacity far exceeds datacenter BDP. eRPC delivers performance that was until now believed possible only with lossless RDMA fabrics or specialized network hardware. It allows unmodified applications to perform close to the hardware limits. Our ported versions of LibRaft and Masstree are, to our knowledge, the fastest replicated key-value store and networked database index in the academic literature, while operating end-to-end without additional network support.

Acknowledgments We received valueable insights from our SIGCOMM and NSDI reviewers, Miguel Castro, John Ousterhout, and Yibo Zhu. Sol Boucher, Jack Kosaian, and Hyeontaek Lim helped improve the writing. CloudLab [59] and Emulab [68] resources were used in our experiments. This work was supported by funding from the National Science Foundation under awards CCF-1535821 and CNS-1700521, and by Intel via the Intel Science and Technology Center for Visual Cloud Systems (ISTC-VCS). Anuj Kalia is supported by the Facebook Fellowship.

Appendix A. eRPC's NIC memory footprint

Primarily, four on-NIC structures contribute to eRPC's NIC memory footprint: the TX and RX queues, and their corresponding completion queues. The TX queue must allow sufficient pipelining to hide PCIe latency; we found that 64 entries are sufficient in all cases. eRPC's TX queue and TX completion queue have 64 entries by default, so their footprint does not depend on cluster size. The footprint of on-NIC page table entries required for eRPC is negligible because we use 2 MB hugepages [25].

As discussed in Section 4.3.1, eRPC's RQs must have sufficient descriptors for all connected sessions. If traditional RQs are used, their footprint grows with the number of connected sessions supported. Modern NICs (e.g., ConnectX-4 and newer NICs from Mellanox) support *multi-packet* RQ descriptors that specify multiple contiguous packet buffers using base address, buffer size, and number of buffers. With eRPC's default configuration of 512-way RQ descriptors, RQ size is reduced by 512x, making it negligible. This optimization has the added advantage of almost eliminating RX descriptor DMA, which is now needed only once every 512 packets. While multi-packet RQs were originally designed for large receive offload of one message [6], we use this feature to receive packets of independent messages.

What about the RX completion queue (CQ)? By default, NICs expect the RX CQ to have sufficient space for each received packet, so using multi-packet RQ descriptors does not reduce CQ size. However, eRPC does not need the information that the NIC DMA-writes to the RX CQ entries. It needs only the number of new packets received. Therefore, we shrink the CQ by allowing it to *overrun*, i.e., we allow the NIC to overwrite existing entries in the CQ in a round-robin fashion. We poll the overrunning CQ to check for received packets. It is possible to use a RX CQ with only one entry, but we found that doing so causes cache line contention between eRPC's threads and the CPU's on-die PCIe controller. We solve this issue by using 8-entry CQs, which makes the contention negligible.

Appendix B. Handling node failures

eRPC launches a session management thread that handles sockets-based management messaging for creating and destroying sessions, and detects failure of remote nodes with timeouts. When the management thread suspects a remote node failure, each dispatch thread with sessions to the remote node acts as follows. First, it flushes the TX DMA queue to release msgbuf references held by the NIC. For client sessions, it waits for the rate limiter to transmit any queued packets for the session, and then invokes continuations for pending requests with an error code. For server-mode sessions, it frees session resources after waiting (non-blocking) for request handlers that have not enqueued a response.

Appendix C. Rate limiting with zero-copy

Recall the request retransmission example discussed in § 4.2.2: On receiving the response for the first copy of a retransmitted request, we wish to ensure that the rate limiter does not contain a reference to the retransmitted copy. Unlike eRPC's NIC DMA queue that holds only a few tens of packets, the rate limiter tracks up to milliseconds worth of transmissions during congestion. As a result, flushing it like the DMA queue is too slow. Deleting references from the rate limiter turned out to be too complex: Carousel requires a bounded difference between the current time and a packet's scheduled transmission time for correctness, so deletions require rolling back Timely's internal rate computation state. Each Timely instance is shared by all slots in a session (§ 4.3), which complicates rollback.

We solve this problem by dropping response packets received while a retransmitted request is in the rate limiter. Each such response indicates a false positive in our retransmission mechanism, so they are rare. This solution does not work for the NIC DMA queue: since we use unsignaled transmission, it is generally impossible for software to know whether a request is in the DMA queue without flushing it.

References

[1] Private communication with Mellanox.

[2] Fast memcpy with SPDK and Intel I/OAT DMA Engine. https://software.intel.com/en-us/articles/fast-memcpy-using-spdk-and-ioat-dma-engine.

[3] A peek inside Facebook's server fleet upgrade. https://www.nextplatform.com/2017/03/13/peek-inside-facebooks-server-fleet-upgrade/, 2017.

[4] Mellanox Accelio. http://www.accelio.org, 2017.

[5] Mellanox MLNX-OS user manual for Ethernet. http://www.mellanox.com/related-docs/prod_management_software/MLNX-OS_ETH_v3_6_3508_UM.pdf, 2017.

[6] Mellanox OFED for Linux release notes. http://www.mellanox.com/related-docs/prod_software/Mellanox_OFED_Linux_Release_Notes_3_2-1_0_1_1.pdf, 2017.

[7] Oak Ridge leadership computing facility - Summit. https://www.olcf.ornl.gov/summit/, 2017.

[8] Aurora 710 based on Barefoot Tofino switching silicon. https://netbergtw.com/products/aurora-710/, 2018.

[9] Facebook open switching system FBOSS and Wedge in the open. https://code.facebook.com/posts/843620439027582/facebook-open-switching-system-fboss-and-wedge-in-the-open/, 2018.

[10] RDMAmojo - blog on RDMA technology and programming by Dotan Barak. http://www.rdmamojo.com/2013/01/12/ibv_modify_qp/, 2018.

[11] Distributed asynchronous object storage stack. https://github.com/daos-stack, 2018.

[12] Tolly report: Mellanox SX1016 and SX1036 10/40GbE switches. http://www.mellanox.com/related-docs/prod_eth_switches/Tolly212113MellanoxSwitchSXPerformance.pdf, 2018.

[13] Jim Warner's switch buffer page. https://people.ucsc.edu/~warner/buffer.html, 2018.

[14] C implementation of the Raft consensus protocol. https://github.com/willemt/raft, 2018.

[15] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pfabric: Minimal near-optimal datacenter transport. In *Proc. ACM SIGCOMM*, Hong Kong, China, Aug. 2013.

[16] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the SIGMETRICS'12*, June 2012.

[17] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014.

[18] C. Binnig, A. Crotty, A. Galakatos, T. Kraska, and E. Zamanian. The end of slow networks: It's time for a redesign. In *Proc. VLDB*, New Delhi, India, Aug. 2016.

[19] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 1984.

[20] B. Callaghan, T. Lingutla-Raj, A. Chiu, P. Staubach, and O. Asad. NFS over RDMA. In *Proceedings of the ACM SIGCOMM Workshop on Network-I/O Convergence: Experience, Lessons, Implications*, 2003.

[21] Y. Chen, X. Wei, J. Shi, R. Chen, and H. Chen. Fast and general distributed transactions using RDMA and HTM. In *Proc. 11th ACM European Conference on Computer Systems (EuroSys)*, Apr. 2016.

[22] D. Crupnicoff, M. Kagan, A. Shahar, N. Bloch, and H. Chapman. Dynamically-connected transport service, May 19 2011. URL https://www.google.com/patents/US20110116512. US Patent App. 12/621,523.

[23] M. DeBergalis, P. Corbett, S. Kleiman, A. Lent, D. Noveck, T. Talpey, and M. Wittle. The direct access file system. In *Proceedings of the 2Nd USENIX Conference on File and Storage Technologies*, 2003.

[24] DPDK. Data Plane Development Kit (DPDK). http://dpdk.org/, 2017.

[25] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast remote memory. In *Proc. 11th USENIX NSDI*, Seattle, WA, Apr. 2014.

[26] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *Proc. 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, Oct. 2015.

[27] A. Dragojevic, D. Narayanan, and M. Castro. RDMA reads: To use or not to use? *IEEE Data Eng. Bull.*, 2017.

[28] M. D. et al. Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In *Proc. 15th USENIX NSDI*, Renton, WA, Apr. 2018.

[29] D. Firestone et al. Azure accelerated networking: SmartNICs in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, Renton, WA, Apr. 2018.

[30] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. A. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, Z. Lin, and V. Kurien. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proc. ACM SIGCOMM*, London, UK, Aug. 2015.

[31] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill. IronFleet: Proving practical distributed systems correct. In *Proc. 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, Oct. 2015.

[32] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: wait-free coordination for internet-scale systems. In *Proc. USENIX Annual Technical Conference*, Boston, MA, June 2010.

[33] Z. István, D. Sidler, G. Alonso, and M. Vukolic. Consensus in a box: Inexpensive coordination in hardware. In *Proc. 13th USENIX NSDI*, Santa Clara, CA, May 2016.

[34] Z. István, D. Sidler, and G. Alonso. Caribou: Intelligent distributed storage. Aug. 2017.

[35] E. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mTCP: A highly scalable user-level TCP stack for multicore systems. In *Proc. 11th USENIX NSDI*, Seattle, WA, Apr. 2014.

[36] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica. NetCache: Balancing key-value stores with fast in-network caching. In *Proc. 26th ACM Symposium on Operating Systems Principles (SOSP)*, Shanghai, China, Oct. 2017.

[37] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica. NetChain: Scale-free sub-RTT coordination. In *Proc. 15th USENIX NSDI*, Renton, WA, Apr. 2018.

[38] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA efficiently for key-value services. In *Proc. ACM*

SIGCOMM, Chicago, IL, Aug. 2014.

[39] A. Kalia, M. Kaminsky, and D. G. Andersen. FaSST: Fast, scalable and simple distributed transactions with two-sided RDMA datagram RPCs. In *Proc. 12th USENIX OSDI*, Savannah, GA, Nov. 2016.

[40] A. Kalia, M. Kaminsky, and D. G. Andersen. Design guidelines for high-performance RDMA systems. In *Proc. USENIX Annual Technical Conference*, Denver, CO, June 2016.

[41] D. Kim, A. Memaripour, A. Badam, Y. Zhu, H. H. Liu, J. Padhye, S. Raindel, S. Swanson, V. Sekar, and S. Seshan. HyperLoop: Group-based NIC-offloading to accelerate replicated transactions in multi-tenant storage systems. In *Proc. ACM SIGCOMM*, Budapest, Hungary, Aug. 2018.

[42] M. J. Koop, J. K. Sridhar, and D. K. Panda. Scalable MPI design over InfiniBand using eXtended Reliable Connection. In *2008 IEEE International Conference on Cluster Computing*, 2008.

[43] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. K. Ports. Just say no to Paxos overhead: Replacing consensus with network ordering. In *Proc. 12th USENIX OSDI*, Savannah, GA, Nov. 2016.

[44] J. Li, E. Michael, and D. R. K. Ports. Eris: Coordination-free consistent transactions using in-network concurrency control. In *Proc. 26th ACM Symposium on Operating Systems Principles (SOSP)*, Shanghai, China, Oct. 2017.

[45] S. Li, H. Lim, V. W. Lee, J. H. Ahn, A. Kalia, M. Kaminsky, D. G. Andersen, O. Seongil, S. Lee, and P. Dubey. Architecting to achieve a billion requests per second throughput on a single key-value store server platform. In *ISCA*, 2015.

[46] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *Proc. 11th USENIX NSDI*, Seattle, WA, Apr. 2014.

[47] F. Liu, L. Yin, and S. Blanas. Design and evaluation of an RDMA-aware data shuffling operator for parallel database systems. In *Proc. 12th ACM European Conference on Computer Systems (EuroSys)*, Apr. 2017.

[48] J. Liu, J. Wu, and D. K. Panda. High performance RDMA-based MPI implementation over InfiniBand. *International Journal of Parallel Programming*, 2004.

[49] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *Proc. 7th ACM European Conference on Computer Systems (EuroSys)*, Bern, Switzerland, Apr. 2012.

[50] C. Mitchell, Y. Geng, and J. Li. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *Proc. USENIX Annual Technical Conference*, San Jose, CA, June 2013.

[51] C. Mitchell, K. Montgomery, L. Nelson, S. Sen, and J. Li. Balancing CPU and network in the Cell distributed B-Tree store. In *Proc. USENIX Annual Technical Conference*, Denver, CO, June 2016.

[52] R. Mittal, T. Lam, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, and D. Zats. TIMELY: RTT-based congestion control for the datacenter. In *Proc. ACM SIGCOMM*, London, UK, Aug. 2015.

[53] R. Mittal, A. Shpiner, A. Panda, E. Zahavi, A. Krishnamurthy, S. Ratnasamy, and S. Shenker. Revisiting network support for RDMA. In *Proc. ACM SIGCOMM*, Budapest, Hungary, Aug. 2018.

[54] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *Proc. USENIX Annual Technical Conference*, Philadelphia, PA, June 2014.

[55] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in RAMCloud. In *Proc. 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, Oct. 2011.

[56] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang. The RAMCloud storage system. *ACM TOCS*, 2015.

[57] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker. NetBricks: Taking the V out of NFV. In *Proc. 12th USENIX OSDI*, Savannah, GA, Nov. 2016.

[58] M. Poke and T. Hoefler. DARE: High-performance state machine replication on RDMA networks. In *HPDC*, 2015.

[59] R. Ricci, E. Eide, and The CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login:*, 2014.

[60] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the social network's (datacenter) network. In *Proc. ACM SIGCOMM*, London, UK, Aug. 2015.

[61] A. Saeed, N. Dukkipati, V. Valancius, V. The Lam, C. Contavalli, and A. Vahdat. Carousel: Scalable traffic shaping at end hosts. In *Proc. ACM SIGCOMM*, Los Angeles, CA, Aug. 2017.

[62] J. Shi, Y. Yao, R. Chen, H. Chen, and F. Li. Fast and concurrent RDF queries with RDMA-based distributed graph exploration. In *Proc. 12th USENIX OSDI*, Savannah, GA, Nov. 2016.

[63] M. Su, M. Zhang, K. Chen, Z. Guo, and Y. Wu. RFP: When RPC is faster than server-bypass with RDMA. In *Proc. 12th ACM European Conference on Computer Systems (EuroSys)*, Apr. 2017.

[64] Y. Wang, X. Meng, L. Zhang, and J. Tan. C-hint: An effective and reliable cache management for RDMA-accelerated key-value stores. In *Proc. 5th ACM Symposium on Cloud Computing (SOCC)*, Seattle, WA, Nov.

2014.

[65] Y. Wang, L. Zhang, J. Tan, M. Li, Y. Gao, X. Guerin, X. Meng, and S. Meng. Hydradb: A resilient RDMA-driven key-value middleware for in-memory cluster computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015.

[66] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. Fast in-memory transaction processing using RDMA and HTM. In *Proc. 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, Oct. 2015.

[67] B. B. Welch. The Sprite remote procedure call system. Technical report, Berkeley, CA, USA, 1986.

[68] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. 5th USENIX OSDI*, pages 255–270, Boston, MA, Dec. 2002.

[69] E. Zamanian, C. Binnig, T. Harris, and T. Kraska. The end of a myth: Distributed transactions can scale. In *Proc. VLDB*, Munich, Germany, Aug. 2017.

[70] J. Zhang, F. Ren, X. Yue, R. Shu, and C. Lin. Sharing bandwidth by allocating switch buffer in data center networks. *IEEE Journal on Selected Areas in Communications*, 2014.

[71] Q. Zhang, V. Liu, H. Zeng, and A. Krishnamurthy. High-resolution measurement of data center microbursts. In *Proceedings of the 2017 Internet Measurement Conference*, IMC '17, 2017.

[72] J. Zhou, M. Tewari, M. Zhu, A. Kabbani, L. Poutievski, A. Singh, and A. Vahdat. WCMP: Weighted cost multipathing for improved fairness in data centers. In *Proc. 9th ACM European Conference on Computer Systems (EuroSys)*, Apr. 2014.

[73] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang. Congestion control for large-scale RDMA deployments. In *Proc. ACM SIGCOMM*, London, UK, Aug. 2015.

[74] Y. Zhu, M. Ghobadi, V. Misra, and J. Padhye. ECN or delay: Lessons learnt from analysis of DCQCN and TIMELY. In *Proc. CoNEXT*, Dec. 2016.