PROGRAMMING

Datacenter RPCs Can Be General and Fast

ANUJ KALIA, MICHAEL KAMINSKY, AND DAVID G. ANDERSEN

Anuj Kalia is a PhD student in the Computer Science Department at Carnegie Mellon University, advised by David Andersen and Michael Kaminsky. He is interested in high-performance computer systems and networks. akalia@cs.cmu.edu

David G. Andersen is an Associate Professor of Computer Science at Carnegie Mellon University. He completed his MS and PhD degrees at MIT, and holds BS degrees in biology and computer science from the University of Utah. In 1995, he co-founded an Internet service provider in Salt Lake City, Utah. dga@cs.cmu.edu

Michael Kaminsky is a Senior Research Scientist at Intel Labs and an adjunct faculty member of the Computer Science Department at Carnegie Mellon University. He is part of the Intel Science and Technology Center for Visual Cloud Systems (ISTC-VCS), based in Pittsburgh, PA. His research interests include distributed systems, operating systems, and networking. michael.e.kaminsky@intel.com

"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," J. H. Saltzer, D. P. Reed, and D. D. Clark, 1984

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 article, 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.

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, FPGAs, and programmable switches. 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 would yield a large performance gain. Our work shows that a general-purpose RPC library called eRPC can provide state-of-the-art performance on commodity Ethernet datacenter networks without additional network support. This helps inform the debate about the utility of additional in-network functionality versus purely end-to-end solutions for datacenter applications.

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, 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://erpc.io.

Background and Motivation

We first discuss aspects of modern datacenter networks relevant to eRPC and the limitations of existing networking options that underlie the need for eRPC.

High-Speed Datacenter Networking

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

Lossless fabrics. Lossless packet delivery is a link-level feature that prevents congestion-based packet drops. For example, priority-based flow control (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. Unfortunately, PFC comes with a host of problems, including head-of-line blocking, deadlocks due to cyclic buffer dependencies, and complex switch configuration. In our experience, datacenter operators are often unwilling to deploy PFC due to these problems.

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, the RTT between two nodes connected to different top-of-rack (ToR) switches is 6 μs, so the BDP is 19 kB. In contrast to the small BDP, the Mellanox Spectrum switches in our cluster have 12 MB in their dynamic buffer pool. 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.

In practice, we wish to support approximately 50-way incasts: congestion control protocols deployed in real datacenters are tested against comparable incast degrees. 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.

Limitations of Existing Options

Software options. Existing datacenter networking software options sacrifice performance or generality, preventing unmodified applications from using the network efficiently. On the one hand, fully general networking stacks such as mTCP [4] allow legacy sockets-based applications to run unmodified. Unfortunately, they leave substantial performance on the table, especially for small messages. On the other extreme, fast packet I/O libraries such as DPDK provide only unreliable packet delivery.

Our prior RPC design—FaSST RPCs [6]—was the precursor to eRPC. Like eRPC, FaSST RPCs use datagram packet I/O, but they assume a lossless network and lack several features such as multi-packet messages and congestion control. eRPC's main contribution is a design that performs well in lossy networks and supports the aforementioned features with low overhead.

Hardware options. Numerous recent research proposals co-design distributed systems with special network hardware technologies like RDMA, FPGAs, and programmable switches for fast communication. 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 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.

eRPC Overview

eRPC implements RPCs on top of a transport layer that provides basic unreliable packet I/O, such as UDP over Ethernet networks 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 RPCs.

eRPC's requests 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; a library that provides marshalling and unmarshalling can be used as a layer on top of eRPC.

Each user thread that sends or receives requests creates an exclusive RPC endpoint. Each 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 endpoint's event loop to make progress. The event loop performs the bulk of eRPC's work, including packet I/O, invoking request handlers and continuations, congestion control, and management functions. To avoid blocking on a long-running request handler, eRPC provides a pool of background threads to handle request types that are annotated by the user as long-running, typically over a few microseconds.

Client control flow. `rpc->enqueue_request()` queues a request 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 buffer 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.

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 transfers, 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.

Scalability considerations. We chose plain packet I/O instead of RDMA writes to send messages in eRPC. eRPC holds connection state in large CPU caches, which allows scaling to a large number of connections. In contrast, RDMA requires maintaining per-connection in much smaller (~2 MB) on-NIC caches, which does not scale well to large clusters. Our experiments show that whereas RDMA performance drops by up to 50% with 5000 connections, eRPC's performance remains constant with even 20,000 connections. In addition, eRPC uses modern NIC features (e.g., multi-packet receive queues) to guarantee a constant NIC memory footprint per local CPU core.

Zero-copy challenges. eRPC supports zero-copy transfers from DMA-capable buffers provided to applications. Supporting zero-copy along with eRPC's feature set required solving several challenges, such as reasoning about DMA buffer ownership in the presence of retransmissions. Since eRPC transfers packets directly from application-owned buffers, care must be taken so that buffer references are never used by eRPC after buffer ownership is returned to the application. The following example demonstrates the problem: 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 sent out by the client's NIC. Before processing the response and invoking the continuation, we must ensure that there are no references to the request buffer in the client's NIC DMA queue.

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 increases NIC and PCIe resource use, 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 flush is moderately expensive ($\approx 2\ \mu s$), but it is called only during rare retransmissions.

Sessions. Each session maintains multiple outstanding requests to keep the network pipe full. Concurrent requests on a session can complete *out-of-order* with respect to each other. This avoids blocking dispatch-mode RPCs behind a long-running background RPC. We support a constant number of concurrent requests (default = 8) per session; additional requests are transparently queued by eRPC.

eRPC limits the number of unacknowledged packets on a session to implement end-to-end flow control, which reduces switch queueing. Allowing *BDP/MTU* unacknowledged packets per session ensures that each session can achieve line rate.

Transport Layer

One of eRPC's main contributions is the design of low-overhead transport layer components, including end-to-end reliability and congestion control, discussed next. eRPC uses a *client-driven* protocol, meaning that each packet sent by the server is in response to a client packet. This shifts most transport complexity to clients, freeing server CPU that is often more valuable.

End-to-end reliability. For simplicity, eRPC treats reordered packets as losses by dropping them. Datacenter networks typically preserve intra-flow ordering even with network load balancing (e.g., ECMP), except during rare route churn events. On suspecting a lost packet, the client rolls back the request's wire protocol state using a simple Go-Back-N mechanism, and retransmits from the updated state. The server never runs the request handler for a request twice, guaranteeing at-most-once RPC semantics.

Congestion control. Congestion control for datacenter networks aims to reduce switch queueing, thereby preventing packet drops and reducing RTT. While software-based congestion control has been considered to be slow in the past, we show that optimizing for uncongested networks, and recent advances in software rate limiting allow congestion control with little overhead.

eRPC uses a congestion control algorithm for high-speed datacenter networks called Timely [7], although other algorithms may also be supported in the future. Timely uses packet RTT as the congestion signal, and it updates session transmission rates based on RTT statistics. We use a software rate limiter for enforcing the transmission rate suggested by Timely.

Datacenter networks are typically uncongested, so we optimize congestion control for uncongested networks. Recent datacenter studies support this claim. For example, Roy et al. [9] report that 99% of all Facebook datacenter links are less than 10% utilized at one-minute 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. If the RTT of a packet received on an uncongested session is smaller than Timely's low threshold (~50 μs), below which it performs additive increase, we do not perform a rate update. For uncongested sessions, we transmit packets directly instead of placing them in the rate limiter.

Evaluation

eRPC is implemented in 6200 SLOC of C++, excluding tests and benchmarks. We evaluated eRPC's performance both in microbenchmarks and real applications. The numbers presented here were measured on an eight-node cluster with Intel Xeon servers, with Mellanox ConnectX-5 NICs connected to a 40 GbE switch.

Microbenchmarks

Latency. For small 32-byte RPCs, eRPC's median latency is 2.3 μs, which is only 300 ns more than 32-byte RDMA reads.

Bandwidth. To measure eRPC's bandwidth for large messages, we use a client that sends large requests to a server thread, which replies with small, 32-byte responses. With 8 MB requests, eRPC saturates the network's 40 Gbps with one client thread. On a faster 100 Gbps InfiniBand network, we measured that eRPC can achieve 75 Gbps in the same experiment.

In addition, our microbenchmarks showed that eRPC also provides:

- **High scalability**. On a large 100-node cluster, eRPC's performance scales to 20,000 connections per-node.
- **Incast tolerance**. eRPC's congestion control successfully reduces switch queueing with up to 50-way incasts.
- **Packet loss tolerance**. eRPC delivers good bandwidth with a packet loss rate of up to 10^{-5}.

Raft over eRPC

To evaluate whether eRPC can be used in real applications with unmodified existing storage software, we built a state machine replication system using an open-source implementation of Raft [8].

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 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: 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 [1]. (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** (μs)	**99%** (μs)
Measured at client	NetChain	9.7	N/A
	eRPC	5.5	6.3
Measured at leader	ZabFPGA	3.0	3.0
	eRPC	3.1	3.4

Table 1: Latency comparison for replicated PUTs

We compare against recent consistent replication systems that are built from scratch for two specialized hardware types. First, NetChain [5] implements chain replication over programmable switches. Second, Consensus in a Box [3] (called ZabFPGA here) implements ZooKeeper's atomic broadcast protocol [2] on FPGAs.

Workloads. We mimic NetChain and ZabFPGA's experiment setups for latency measurement: we implement a three-way replicated in-memory key-value store with 16-byte keys and 64-byte values, and use one client to issue PUT requests. The replicas' command logs and key-value store are stored in DRAM. We compare eRPC's performance on CX5 against their published numbers because we do not have the hardware to run NetChain or ZabFPGA. Table 1 compares the latencies of the three systems.

Comparison with NetChain. NetChain's key assumption is that software networking adds 1–2 orders of magnitude more latency than switches [5]. However, our experiments show that eRPC adds 850 ns, which is comparable to latency added by current programmable switches (~800 ns).

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 our ConnectX-5 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.

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.

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, and it allows unmodified applications to perform close to the hardware limits.

References

[1] C Implementation of the Raft Consensus Protocol, 2019: https://github.com/willemt/raft.

[2] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-Free Coordination for Internet-Scale Systems," in *Proceedings of the USENIX Annual Technical Conference (USENIX ATC '10)*, June 2010: https://www.usenix.org/legacy/event/usenix10/tech/full_papers/Hunt.pdf.

[3] Z. István, D. Sidler, G. Alonso, and M. Vukolic, "Consensus in a Box: Inexpensive Coordination in Hardware," in *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI '16)*, May 2016: https://www.usenix.org/system/files/conference/nsdi16/nsdi16-paper-istvan.pdf.

[4] 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 *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14)*, April 2014: https://www.usenix.org/system/files/conference/nsdi14/nsdi14-paper-jeong.pdf.

[5] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica, "NetChain: Scale-Free Sub-RTT Coordination," in *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18)*, April 2018: https://www.usenix.org/system/files/conference/nsdi18/nsdi18-jin.pdf.

[6] A. Kalia, M. Kaminsky, and D. G. Andersen, "FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided RDMA Datagram RPCs," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, November 2016: https://www.usenix.org/system/files/conference/osdi16/osdi16-kalia.pdf.

[7] 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 *Proceedings of the ACM SIGCOMM*, August 2015.

[8] D. Ongaro and J. Ousterhout, "In Search of an Understandable Consensus Algorithm," in *Proceedings of the USENIX Annual Technical Conference (USENIX ATC '14)*, June 2014: https://www.usenix.org/system/files/conference/atc14/atc14-paper-ongaro.pdf.

[9] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the Social Network's (Datacenter) Network," in *Proceedings of the ACM SIGCOMM*, August 2015.