Efficient Remote Procedure Calls for Datacenters

Anuj Kalia

CMU-CS-19-126

September 2019

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

*Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.*

This research was sponsored by the National Science Foundation under grant numbers CNS-1314721 and CNS-1700521, Intel ISTC-CC, Intel ISTC-VCC, and a Facebook Fellowship. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: datacenter networks, distributed systems, Remote Procedure Calls, Remote Direct Memory Access, eRPC

To my father, Darshan Lal Kalia,
for getting me here

Abstract

Datacenter network latencies are approaching their microsecond-scale speed-of-light limit, and network bandwidths continue to grow beyond 100 Gbps. These improvements bear rethinking the design of communication-intensive distributed systems for datacenters, whose performance has historically been limited by slow networks. With the slowing down of Moore's law, a popular approach is to redesign distributed systems to use network hardware devices and technologies that offload communication or data access from commodity CPUs, such as smart network cards (NICs), lossless networks, programmable NICs, and programmable switches.

In this dissertation, we show that we can continue to use end-to-end software-only communication mechanisms to build high-performance distributed systems, i.e., we bring the speed of fast networks to distributed systems without an expensive redesign with in-network hardware offloads. We show that the ubiquitous Remote Procedure Call (RPC) communication mechanism, when rearchitected specially for the capabilities of modern commodity datacenter hardware, is a fast, scalable, flexible, and simple communication choice for distributed systems. We make three contributions. First, we present a detailed analysis of datacenter communication hardware—ranging from the peripheral bus that connects CPUs to NICs, to the datacenter's switched network—that informs our choice of the communication mechanism. Second, we lay out the advantages of RPCs over network hardware offloads through the design and evaluation of two new systems, a key-value store called HERD, and a distributed transaction processing system called FaSST. Third, we combine the lessons learned from the first two steps with new insights about datacenter packet loss and congestion control to create a new RPC library called eRPC, and show how existing distributed system codebases perform well over eRPC. In many cases, these systems substantially outperform offloads because they use less communication, and their end-to-end design provides flexibility and simplicity.

Acknowledgments

Around this time six years ago, I burned the midnight oil learning Paxos in an attempt to convince David Andersen and Michael Kaminsky to take me under their wing as a student. I still don't understand Paxos, but my effort paid off immensely. I have learned a tremendous amount from Dave, from executing a research vision, to ethics and the value of doing the right thing, to correctly formatting `i++`. Dave gave me freedom to chart my own course, while still providing close guidance and supporting my decisions throughout. Thanks, Dave, for making this the amazing journey that it was.

I am grateful to Michael for his consistency in mentorship, collaboration and support. Michael taught me by example that the perfect is the enemy of the good, and that the answer to many questions in research and in life is "it depends." At a time when I was looking for the perfect thesis topic, Michael encouraged me to commit to datacenter networks research. I am fortunate to have followed his advice. Thanks for everything, Michael.

My debt to Miguel Castro and Justine Sherry goes beyond them serving on my thesis committee. Miguel's feedback on my work was crucial to the FaSST and eRPC projects. Justine provided invaluable help during my job search, especially by single-handedly rescuing my job talk.

I am also grateful to Garth Gibson, Dushyanth Narayanan, Vyas Sekar, Amar Phanishayee, Nathan Beckmann, Brandon Lucia, and Rashmi Vinayak for their help and guidance at various phases in my PhD. Joe Moore's generosity in providing access to a cluster at NetApp was invaluable. The administrative aspects of the PhD program were easy thanks to Deb Cavlovich, Angy Malloy, and Karen Lindenfelser.

Members and friends of the FAWN group—Iulian Moraru, Hyeontaek Lim, Dong Zhou, Huanchen Zhang, Conglong Li, Sol Boucher, Thomas Kim, Angela Jiang, Chris Canel, Giulio Zhou, and Daniel Wong—created a fun, cooperative, and helpful research environment. In particular, I learned the art of building and evaluating systems from Hyeontaek.

I have been lucky to have many close friends. Alok, Rohan, Ojasvi and Rishika, and Deepak and Shikha: thanks for providing a home away from home, and for the many adventures. Deepak Vasisht: I would not have done a PhD if random luck had not made us roommates ten years ago. Thanks for the friendship, help, and competition, and I hope our paths will cross once again. My friends in Pittsburgh made this journey so much more enjoyable. Thanks to Yang Jiao for all the support, and for introducing me to good food. Jack Kosaian's magnanimity as a person, friend and officemate made me finally stop working from home. Angela Jiang changed my worldview in the short span of a few months. Thanks, Angela, for being the coolest person ever.

I am grateful for the constant love and support of my family. My greatest debt is towards my mother and father, who made many great sacrifices to help me succeed. My father made it a mission in his life to see me get the best education, and towards this goal he left no stone unturned. For all the times he went to bat for me, this thesis is dedicated to him.

Contents

1 **Introduction** **1**
1.1 Thesis contributions and outline . . . 2
1.2 Distributed systems performance from a speed-of-light perspective . . . 3
1.3 An end-to-end design for high flexibility . . . 5
1.4 Scalability: A silicon power consumption argument . . . 6
1.5 A fast and general-purpose design for simplicity . . . 7
1.6 Evolution of our RPC designs . . . 8

2 **Background** **9**
2.1 Modern datacenter networks . . . 9
2.1.1 Datacenter network hardware . . . 10
2.1.2 Userspace networking . . . 12
2.1.3 In-network offloads for distributed systems . . . 13
2.2 Communication-intensive distributed applications . . . 18
2.2.1 Main-memory key-value stores . . . 18
2.2.2 Distributed transaction processing . . . 19
2.2.3 State machine replication . . . 19
2.2.4 Common application workload characteristics . . . 20
2.3 Evaluation clusters . . . 20
2.4 Open-source code . . . 21

3 **Guidelines for use of modern high-speed NICs** **22**
3.1 A review of PCI Express . . . 23
3.1.1 PCIe headers . . . 23
3.1.2 Memory-mapped I/O and Direct Memory Access . . . 24
3.2 How modern NICs work . . . 25
3.3 RDMA terminology . . . 26
3.3.1 RDMA verbs . . . 26
3.3.2 RDMA queue pairs . . . 27
3.3.3 RDMA transport types . . . 27
3.4 Preface to the guidelines . . . 28
3.5 Guidelines for NICs with transport-layer offload . . . 29
3.5.1 Prefer application-level ACKs over transport-level ACKs . . . 29
3.5.2 Avoid storing connection state on NICs . . . 30

3.6 Reduce PCIe traffic . . . 32
3.6.1 Measurement method: PCIe counters on commodity CPUs . . . 32
3.6.2 Reduce CPU-initiated MMIOs . . . 33
3.6.3 Reduce NIC-initiated DMAs . . . 36
3.7 Guidelines based on NIC architecture . . . 38
3.7.1 Engage multiple NIC processing units . . . 39
3.7.2 Avoid contention among NIC processing units . . . 40
3.7.3 Avoid NIC cache misses . . . 41
3.8 Related work . . . 42
3.9 Conclusion . . . 43

4 Case study 1: HERD – An RPC-based key-value store 44
4.1 Introduction . . . 44
4.2 Background . . . 46
4.2.1 Recent research on key-value stores . . . 46
4.2.2 One-sided RDMA–based key-value stores . . . 46
4.3 Design decisions . . . 47
4.3.1 Notation and experimental setup . . . 48
4.3.2 Constructing a fast RPC primitive . . . 48
4.3.3 Using datagram transport for responses . . . 52
4.4 Design of HERD . . . 54
4.4.1 HERD's key-value data structure . . . 54
4.4.2 Masking DRAM latency with prefetching . . . 54
4.4.3 Request format and handling . . . 56
4.4.4 Response format and handling . . . 57
4.5 Evaluation . . . 57
4.5.1 Experimental setup . . . 57
4.5.2 Workloads . . . 59
4.5.3 Throughput comparison with one-sided RDMA approaches . . . 59
4.5.4 Latency comparison with one-sided RDMA approaches . . . 62
4.5.5 Cluster scalability . . . 62
4.5.6 CPU use . . . 63
4.5.7 Resistance to workload skew . . . 64
4.6 Revisiting HERD's design for faster NICs . . . 65
4.6.1 Applying Doorbell batching to HERD . . . 65
4.6.2 Comparison with key-value stores that use RDMA atomics . . . 66
4.7 A networked sequencer with HERD RPCs . . . 67
4.7.1 Specializing HERD RPCs for the sequencer's workload . . . 68
4.7.2 Comparison with atomics-based sequencers . . . 70
4.8 Related work . . . 70
4.9 Conclusion . . . 71

5 Case study 2: FaSST – Fast, Scalable, and Simple Distributed Transactions 73
5.1 Distributed transactions background . . . 74

5.2 Choosing networking primitives . . . 75
5.2.1 Advantages of RPCs for transactions . . . 75
5.2.2 Advantages of datagram transport . . . 76
5.2.3 Performance considerations . . . 77
5.2.4 Reliability considerations . . . 81
5.3 FaSST RPCs . . . 82
5.3.1 Coroutines for network latency hiding . . . 82
5.3.2 RPC interface and optimizations . . . 83
5.3.3 Detecting packet loss . . . 84
5.3.4 Limitations of FaSST RPCs . . . 84
5.3.5 Single-core RPC performance . . . 85
5.4 Transactions . . . 86
5.4.1 Handling failures and packet loss . . . 88
5.4.2 Implementation . . . 89
5.5 Evaluation . . . 90
5.5.1 Performance for an object store . . . 91
5.5.2 Performance on the TATP benchmark . . . 93
5.5.3 Performance on the SmallBank benchmark . . . 94
5.5.4 Transaction latency on TATP . . . 94
5.6 Discussion . . . 95
5.6.1 Dynamically Connected Transport . . . 95
5.6.2 Comparison with hybrid RPC-RDMA approaches . . . 96
5.6.3 Advanced one-sided RDMA . . . 96
5.7 Related work on distributed transactions . . . 97
5.8 Conclusion . . . 97

6 eRPC: A Fast and General-purpose RPC Library 98
6.1 Understanding packet loss in datacenter networks . . . 99
6.2 eRPC overview . . . 100
6.2.1 RPC API . . . 100
6.2.2 Worker threads . . . 101
6.2.3 Evaluation clusters . . . 101
6.3 eRPC design . . . 102
6.3.1 Scalability considerations . . . 102
6.3.2 Challenges in zero-copy transmission . . . 103
6.3.3 Sessions . . . 106
6.4 Wire protocol . . . 107
6.4.1 Protocol messages . . . 108
6.4.2 Congestion control . . . 108
6.4.3 Handling packet loss . . . 110
6.5 Microbenchmarks . . . 110
6.5.1 Small RPC latency . . . 111
6.5.2 Small RPC rate . . . 111
6.5.3 Session scalability . . . 112

6.5.4 Large RPC bandwidth . . . 113
6.5.5 Effectiveness of congestion control . . . 115
6.6 Full-system evaluations . . . 115
6.6.1 Raft over eRPC . . . 116
6.6.2 Masstree over eRPC . . . 117
6.7 Conclusion . . . 118

7 Lessons learned, limitations, and looking forward 119
7.1 Lessons learned . . . 119
7.2 Limitations . . . 120
7.3 Future work . . . 121
7.3.1 Protocol and API design for datacenter networks . . . 121
7.3.2 Towards a full-fledged networking library . . . 122

Bibliography 123

"Remote procedure calls (RPC) appear to be a useful paradigm."

Birrell & Nelson, 1984

Chapter 1

Introduction

In the past, datacenter networks were slow, and distributed systems for datacenters were designed to minimize network communication. Datacenter networks have become fast over the past decade, and their speed continues to rise. As a result, communication-intensive distributed systems (e.g., networked storage systems, and coordination services) can now achieve high performance. The improvement in datacenter network speed is perhaps best illustrated by comparing the speed of the network between two hosts to the speed of local main memory (DRAM) at one host. Table 1.1 shows DRAM and network speeds for technologies deployed in large-scale Internet datacenters in 2009 and 2019; the network's speed is measured between two machines in one rack.

	2009			**2019**		
	DRAM	*Network*	*DRAM : Net*	*DRAM*	*Network*	*DRAM : Net*
Latency (nanoseconds)	100	300,000	1 : 3000	80	2,000	1 : 25
Bandwidth (GB/s)	20	0.1	200 : 1	100	12.5	8 : 1

Table 1.1: Comparison of datacenter network speed to main memory speed today and a decade ago. The performance numbers for 2009 are reproduced from Dean [32].

A decade ago, networks were comparatively much slower than DRAM, but the difference has decreased by over an order of magnitude. While network latency used to be 3000x higher than DRAM, it is now only 25x higher. Network bandwidth used to be 200x lower than DRAM, but it is now only 8x lower.

The large improvement in network performance requires revisiting the design and implementation of communication-intensive distributed systems in modern datacenters. Such systems, including key-value stores, online transaction processing systems, and highly-available replicated storage systems, are core services in datacenters. These services run on millions of computers and tens of billions of dollars of installed hardware worldwide, so improving them can save substantial cost and energy. Unmodified communication software, such as the operating system's TCP stack, performs poorly in such communication-intensive systems [38, 97, 113, 129], necessitating new communication approaches for achieving high performance.

Moore's law appears poised to end based on current technological trends, so it is conceivable that software-only communication approaches running on general-purpose CPUs might not be able to keep up with rapid advances in network speed, especially bandwidth. Indeed, a popular approach today is to redesign distributed systems to use in-network hardware devices and technologies that offload network communication, and data access or storage from commodity CPUs, such as special network cards (NICs) [26, 38, 39, 83, 113, 114, 129, 152, 153, 154, 155, 159], lossless networks [81, 99, 165], programmable NICs [72, 73, 90, 102], and programmable switches [76, 92, 93]. Co-designing distributed systems with such in-network offloads is expensive and complex: these hardware devices or technologies are often more expensive than their commodity counterparts, and they impose design and deployment challenges. For example, network devices have a restricted and complicated programming model, which exacerbates the already-complex task of designing distributed systems. Using these devices in applications requires user control over shared network infrastructure, which datacenter operators may not allow.

In contrast, *end-to-end* [140] distributed systems treat the network as a simple lossy pipe, and implement communication and data access in end-host software. This dissertation seeks to answer the following question: *can we continue to use end-to-end software-based communication mechanisms that do not rely on in-network devices to build fast distributed systems for modern datacenters?* We answer the question in the affirmative. We show that we *can* in fact provide good performance, without an expensive redesign with in-network offloads. Counter-intuitively, we also show that in many cases, the speed and scalability of our end-to-end software-based systems exceeds that of systems built with in-network devices. One of our key insights is that, in order to achieve high performance, we must optimize the communication software for the capabilities of datacenter hardware. Because of the ubiquity of Remote Procedure Call (RPC)–based communication, and its potential performance advantages that we summarize in this chapter, we apply this insight to RPCs.

Thesis: *Remote Procedure Calls, rearchitected for the capabilities of modern commodity datacenter hardware, are well-suited for building fast, flexible, and scalable distributed systems.*

1.1 Thesis contributions and outline

We make three contributions in this dissertation.

1. Modern NICs offer the potential for exceptional performance, but design choices including which NIC primitives to use and how to use them determine application performance. In Chapter 3, we present a set of guidelines to help designers of high-performance distributed systems navigate the large design space of NIC primitives and knobs. These guidelines are based on a detailed analysis of the low-level details of modern NICs, such as their interaction with the CPU, and their hardware architecture.

2. We present two case studies demonstrating the performance and scalability benefits of RPC-based systems optimized for datacenter hardware capabilities, over approaches that

use in-network devices. We design, implement, and evaluate two RPC-based distributed systems: the HERD key-value store (Chapter 4), and the FaSST distributed transaction processing system (Chapter 5).

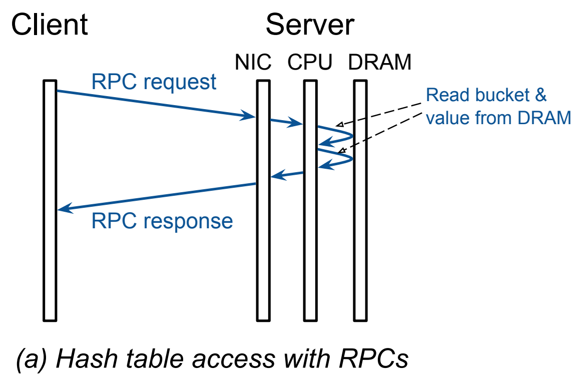
3. The RPC subsystems in HERD and FaSST are simple prototypes designed specifically for the respective systems, and they lack the functionality and ease-of-use of a general-purpose RPC library. We combine the lessons learned from the first two contributions with new insights about datacenter packet loss and congestion control to create a general-purpose RPC library called eRPC (Chapter 6). To demonstrate eRPC's speed and generality, we show how existing distributed system codebases, including a production-grade implementation of Raft state machine replication [23, 121], are easily ported to run over eRPC, and that they perform well.

Systems built with eRPC have four primary advantages over those built with in-network hardware: performance, flexibility, scalability, and simplicity. Our two RPC designs that were precursors to eRPC—HERD RPCs and FaSST RPCs—also possess some of these advantages. Higher performance and scalability arises from the hardware-aware design of our RPCs, and, as discussed next, from the fact that RPC-based distributed systems can often complete operations in fewer round trips than offload-based systems. Higher flexibility arises from the end-to-end design [140] of eRPC, which makes minimal assumptions about in-network capabilities. eRPC provides a clean communication abstraction that is both fast and general-purpose, simplifying the design and implementation of distributed systems that use eRPC. We summarize these four advantages next.

1.2 Distributed systems performance from a speed-of-light perspective

Although the special circuitry of in-network offload devices makes them well-suited for simple communication and data access tasks, there is a gap between the limited computational and memory-access capabilities of these devices, and the capabilities needed for distributed system operations. Compensating for this gap often increases the number of round trips required for an operation compared to an RPC-based approach that uses the general-purpose compute and memory capabilities of CPUs.

A design that uses fewer network round trips gains a *fundamental* performance advantage because network round trip latency is lower-bounded by the speed of light. For instance, in a small datacenter 50 meters in size, the propagation delay of light in the network's optic cables alone is 500 nanoseconds. In addition, the round trip time includes the latency of electric components like NICs and switches, which each add 300–500 ns every time a packet goes across them. These components are already highly optimized for low latency, so we do not expect their latency to go down substantially. Therefore, from a speed-of-light perspective, RPC-based designs are often a better choice than offload-based designs.



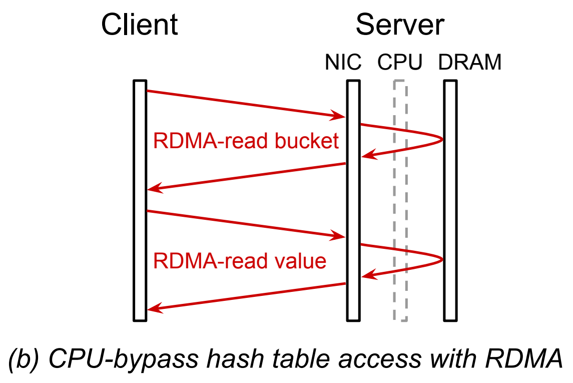


Figure 1.1: Network round trips for accessing a remote hash table (a) with RPCs and (b) with RDMA

As an example, consider the task of designing a networked main memory key-value store. For concreteness, assume that the data is organized as a hash table; a similar argument applies to other data structures such as trees. We assume a typical hash table data structure with one level of pointer indirection: the table consists of an array of buckets that maps keys to pointers, and the values for keys are stored at the pointer's address.

To access a remote key-value store with RPCs, a client sends a request message to the key-value store server (e.g., over TCP or UDP). The server's NIC relays the received request to the server's CPU, which handles the request by retrieving the key's bucket from local DRAM, and dereferencing the pointer in the bucket to get the key's value. Then the server's CPU sends a reply to the client. With RPCs, key-value store accesses complete in one round trip, as shown in Figure 1.1 (a).

Early work in designing networked key-value stores for modern datacenters, exemplified by Pilaf [113] and FaRM [38], found RPC-based approaches too slow, primarily due to the high cost of their software-based communication libraries. To avoid involving CPUs, they proposed using NICs that provide Remote Direct Memory Access (RDMA), an offload feature that allows clients to directly access the server's memory without involving the server's CPU cores. RDMA NICs, however, have limited capabilities, allowing clients to read or write only one contiguous chunk of server memory in one RDMA operation. Data structures typically use pointer-based indirection, so clients must use multiple RDMA round trips to chase pointers in remote memory. For example, retrieving the value for a key from a hash table (one level of indirection) with RDMA requires at least two network round trips: The client first issues an RDMA read to fetch a hash table bucket. After this remote read completes, the client knows the address of the value, which it fetches in another RDMA read. Figure 1.1 (b) shows how this RDMA-based approach compares to the RPC approach.

Our evaluation shows that fewer round trips in RPC-based designs gives a substantial performance advantage in practice. The RTT advantage is the primary reason why (a) HERD outperforms prior RDMA-based key-value stores (e.g., Pilaf and FaRM's key-value store) by around 2x in *both* throughput and latency, and (b) why Raft-over-eRPC outperforms state machine replication approaches that use RDMA or programmable switches. The RTT advantage is also one of the reasons why FaSST outperforms RDMA-based transaction processing systems.

To emphasize that round trip amplification is a common drawback of offload-based approaches, we list below more some examples of round trip amplification in common distributed system operations.

- **Reading and locking a remote object**, a primitive used in applications such as distributed transaction processing, requires two round trips with RDMA: one RDMA atomic operation to acquire a lock, and another to read the object data [154, 159]. In our FaSST transaction processing system (Chapter 5), completing both steps requires one RTT with an RPC.
- **Appending an entry to a remote log** with RDMA may require two round trips: one to write an entry to the log, and one to increment the log's tail pointer [129]. Such logging requires one RPC to an active server, as in our Raft-over-eRPC system (Section 6.6.1).
- **Replicating an object** in the memory of a group of programmable network switches often necessitates using serial, chain replication [151] due to the limited compute and memory resources on such switches [76]. RPC-based approaches can instead use parallel primary-backup replication, which has lower latency with four or more replicas.

1.3 An end-to-end design for high flexibility

The most advanced RPC library presented in this dissertation, eRPC (Chapter 6), follows an end-to-end design, meaning that the network provides only lossy datagram forwarding, and all other functionality is contained in software running on end-host CPUs. The network need not provide more sophisticated capabilities such as losslessness, in-order packet delivery, and congestion control, or in-network offload devices. Our decision to not rely on special in-network capabilities has its roots in classic networks and systems design principles.

David Clark noted in 1988 [27] that making minimal assumptions about the capabilities of the underlying network helps maximize the range of network designs across which the Internet protocols can function. Similar benefits arise in datacenter networks; these benefits, however, must be weighed against the potential cost of eliding possible optimization from placing additional capabilities in network hardware. Unlike the Internet, datacenters run in a more homogeneous context. The large number of recent systems that depend on in-network offloads, such as RDMA, lossless networks, programmable NICs, and programmable switches might suggest that, in datacenters, the performance advantages outweigh the reduction in flexibility.

The end-to-end arguments by Saltzer et al. [140] offer guidance about whether to implement functionality at a low level or a high level. They note that resisting function inclusion in the low-level core of a system is often the correct design choice [135], and that *"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."* Our experience with designing communication software for modern datacenters is in line with this quote: end-to-end systems built with eRPC are faster than or competitive with systems that rely on in-network capabilities, while being more flexible.

The higher flexibility of eRPC's end-to-end design manifests in three broad ways.

1. **Variety of networks.** eRPC requires only basic datagram forwarding, so it works in all datacenter networks. In contrast, designs that rely on special in-network capabilities are limited to datacenters that support such capabilities. At the time of writing, such capabilities are deployed in only a small fraction of datacenters. Adopting designs that depend critically on network functionality can also tie developers' hands in the future. For example, current RDMA deployments require the network's link layer to prevent packet loss, which precludes transport-level mechanisms that depend on lost packets as a signal (e.g., packet loss–based congestion control).

2. **Data access.** Applications often require data access in ways that are handled easily by a CPU, but are inefficient, cumbersome, or unsupported in in-network devices. A CPU may perform any required sequence of memory reads and writes, use cheap atomic instructions to manage concurrent access to memory, or use instruction fences and cache line flushes to provide ordering or durability in hosts with non-volatile memory. In-network devices restrict memory access patterns (e.g., RDMA allows only one contiguous access per round trip). These devices typically attach to the host over the PCI Express (PCIe) bus, and, as a result, lack first-class support for concurrency control, memory ordering, and durability.

3. **Network protocol innovation.** Network transports for datacenters evolve rapidly. For example, new and improved schemes for congestion control and packet retransmission are invented frequently. Implementing these transport functionalities in software allows adapting to new innovations quickly, because it is much easier to change end-host software than modifying network devices. Transport layer functionalities are sometimes implemented in NICs or switches to get high performance and to free up CPU cycles, but modifications might require support from the network vendor [55, 165]. eRPC uses new insights about datacenter congestion control and packet loss to show that a full-fledged software transport can be fast, disproving the commonly-held belief that offloading the transport to network hardware is necessary for performance.

1.4 Scalability: A silicon power consumption argument

Distributed systems in modern datacenters may run on thousands of hosts. The large scale of communication imposes a scalability challenge at every host that is best handled by CPUs, whose high-capacity and sophisticated memory subsystem (i.e., CPU caches and DRAM) provides fast access to the large amount of communication state in such settings. In contrast, network offload devices such as NICs and FPGAs have less capable memory subsystems for economic reasons, reducing their scalability.

Providing a full-fledged network transport typically requires a connection-oriented design, in which the host maintains per-connection state for every pair of communicating peers. This is because connections are a natural way to providing reliable packet delivery, congestion control, and encryption; the state size for each connection is typically a few hundred bytes. For example, in Mellanox's current implementation of RDMA transport in their NICs, each connection requires around 375 B of state.

The combined memory footprint of connection state at one host can be several tens of megabytes. A high-end host in a modern datacenter today has around 100 CPU cores. To communicate with 1000 peers in the datacenter without sharing connections among CPU cores, the host creates $100 \times 1000 = 100{,}000$ connections. The combined state for these connections is around 100000×375 bytes, which equals 37.5 MB. This memory footprint is manageable in CPUs, which have tens of megabytes of cache, and fast DRAM to serve cache misses. In addition, memory latency hiding techniques in hardware and software can reduce the penalty of cache misses. Transport layer offload devices like NICs and FPGAs have only a few megabytes of cache, resulting in frequent cache misses at large scale. These cache misses are expensive because they are typically served from their host CPU's memory subsystem over the slow PCIe bus.

Caches on offload devices are unlikely to become much larger in the near future because these devices are expected to consume little power. In order to stay power-efficient, they dedicate little die area to caches, which are power-hungry. In general, the utility of accelerators often comes from their low cost and power efficiency. Transport accelerators with more cache will be more expensive and power-hungry, and therefore might be less useful. Compared to more powerful CPUs, making transport accelerators much more powerful appeals to a smaller demographic of users and applications, and pins resources (e.g., cost and energy) an inflexible way.

1.5 A fast and general-purpose design for simplicity

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. This is because existing datacenter networking software options sacrifice performance or generality, preventing unmodified applications from using the network efficiently. On the one extreme, low-level high-speed packet I/O libraries such as DPDK [37] are fast, but lack features required by real applications, such as support for multi-packet messages, packet retransmission, and congestion control. Fully-general networking stacks such as mTCP [74] and IX [15] allow legacy sockets-based applications to run unmodified. Unfortunately, they leave substantial performance on the table, especially for small messages. For example, with 64 B RPC requests, one server core running can handle only 1.5 million requests per second with IX, and over 10 million requests per second with eRPC.

By providing both high speed and a general-purpose feature set sufficient for real applications, eRPC allows existing distributed system codebases to run at near-network speeds, with little or no modifications to their source code. The explosion of co-designed distributed systems that depend on niche network technologies stemmed from the lack of networking software that provides both speed and generality. eRPC preserves a modular networking abstraction, permitting reuse of existing software and developer effort, including data structures, distributed protocols, programmer hours, tests, formal specifications, and feature sets. Co-designing distributed systems with the network breaks abstraction boundaries between components, and prevents such reuse.

Our work helps achieve more robust designs by making existing systems faster in the confines of current hardware, instead of breaking abstraction boundaries (e.g., the hardware-software boundary, or the boundary between systems software and application software) in pursuit of performance. Our designs aid datacenter-level simplicity, because they don't have much entanglement between modules.

1.6 Evolution of our RPC designs

The research that went into this thesis was conducted over six years, in which we developed a series of RPC designs—HERD RPCs, FaSST RPCs, and eRPC. Our initial designs targeted specific applications, and they had special network requirements, until we arrived at eRPC, which is the focus of this thesis. HERD RPCs require an RDMA-capable network, and they work best in asymmetric client-server applications with many clients and few servers. FaSST RPCs require a network with a lossless link layer, which, in theory, is a less stringent requirement than an RDMA-capable network. (In practice, however, datacenters with lossless networks usually also support RDMA.) FaSST RPCs are specialized for symmetric online transaction processing applications, where every host acts as both a client and server.

Our main contribution in eRPC is an efficient design that does not depend on network features, nor is specialized for a particular application. HERD RPCs and FaSST RPCs help provide context for evaluating eRPC's performance relative to more specialized designs: we show that eRPC provides the flexibility and simplicity of an end-to-end, general-purpose design for a small performance penalty.

The evolution from HERD RPCs, to FaSST RPCs, and finally to eRPC is marked by a decreasing reliance on special network features, made possible by our own improved understanding of datacenter network hardware capabilities. For example, we used RDMA writes for request messages in HERD because we believed that using traditional packet I/O for receiving requests necessarily adds PCIe overhead at the server. In eRPC, we show how modern NIC features allow avoiding much of this PCIe overhead for traditional packet I/O. Similarly, we used a lossless network in FaSST because we were concerned that packet loss in lossy networks would cripple RPC performance. In eRPC, we show how with a little assistance from the software transport layer, switch buffers in lossy datacenters are sufficient to prevent most packet loss.

Chapter 2

Background

In this chapter, we discuss relevant background for high-performance distributed systems in modern datacenter networks. We first describe state-of-the-art datacenter networking hardware and software. We review recent datacenter technology advances that permit inter-host communication with only a few microseconds of latency, and close to a hundred gigabits per second of bandwidth. Next, we discuss the class of *communication-intensive* applications that we target in our work, which spend a large portion of their time in the communication subsystem. We review examples and use cases of such applications, their cluster-level scale, and their communication and computation requirements.

2.1 Modern datacenter networks

A decade ago, fast networks were exclusive to High-Performance Computing (HPC) cluster interconnects, such as InfiniBand, or Cray's Aries and Gemini interconnects. Commodity Ethernet-based datacenter networks were slow, which made it challenging to build fast communication-intensive distributed systems. For example, a replicated database that provides strong consistency guarantees requires that a host coordinate with remote replicas before committing a database transaction. Before the recent adoption of fast Ethernet networks in datacenters, such coordination was often prohibitively expensive, leading database designers to adopt weak consistency guarantees.

Over the last decade, commodity Ethernet-based datacenter networks have caught up with HPC interconnects. Their latency improved from hundreds of microseconds to single-digit microseconds, and bandwidth improved from 1 Gbps to 100 Gbps. A combination of faster networking hardware and software drove this improvement. On the hardware side, vendors created Ethernet switches and NICs that add lower latency and handle higher-bandwidth links. On the software side, datacenter operators deployed userspace networking stacks that bypass the operating system kernel's heavyweight network stack, and eliminate expensive system calls and interrupts from the performance-critical datapath.

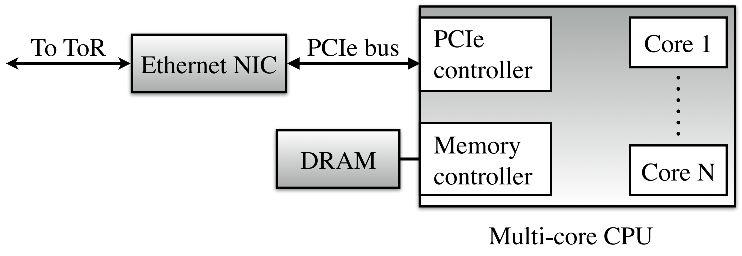


Figure 2.1: Hardware components of a host in a datacenter

We cannot simply re-purpose an existing HPC communication library for datacenter workloads and get high performance. HPC clusters typically use an implementation of the Message Passing Interface (MPI) specification for communication, which are currently ill-suited for datacenter workloads, for three main reasons. First, on commodity datacenter networks that lack RDMA support, MPI implementations use the kernel's TCP stack for reliability and congestion control, and are therefore slower than even kernel TCP. Second, MPI designers typically target workloads with different requirements than datacenter workloads. For example, the message size in HPC workloads is much larger: the average point-to-point MPI message size in HPC workloads ranges from a few kilobytes to hundreds of kilobytes [158]. In contrast, as we discuss later in Section 2.2.4, datacenter applications require high performance for messages as small as a few tens of bytes. MPI implementations perform poorly for such workloads. Third, MPI is usually slow even when HPC interconnects are available. Quoting a recent paper that seeks to remedy MPI's high overhead [131], with authors from several supercomputing centers, and NIC and MPI vendors: *"MPI is often criticized as being a heavyweight runtime system that can add significant overhead, particularly for applications that need very fine-grained communication on fast networks, ..."*

2.1.1 Datacenter network hardware

Datacenter networks connect tens of thousands of hosts in a datacenter. Figure 2.1 shows the high-level hardware architecture of a host in an Internet datacenter, deployed in companies such as Microsoft, Google, and Facebook. Each host contains one or a few multicore CPUs, each with ~20 CPU cores, and a few hundreds of gigabytes of DRAM. The CPUs in one host are attached to one 10–100 gigabit Ethernet NIC at the host using a 32—128 Gbps PCIe link. Hosts dedicated to particular applications such as machine learning may have application-specific accelerators like GPUs (not shown in the figure). A few tens of such hosts are organized in a rack, and a few tens of racks comprise a pod. A typical large datacenter contains many such pods.

Figure 2.2 shows how the hosts in a datacenter are typically interconnected using a fat-tree topology [5]. The NICs of all hosts in one rack connect to a *top-of-rack* (ToR) switch. ToR switches in one pod connect to leaf switches in the pod. Leaf switches from all pods in the datacenter connect to spine switches.

Component	PCIe bus	Network card	Switch
Approx. one-way latency	200 ns	200 ns	300 ns

Table 2.1: State-of-the-art network hardware components add a few hundred seconds of latency

Network latency. State-of-the-art datacenter round trip time is typically on the order of a few microseconds, especially within a rack or a pod. Several electric components contribute substantial fractions to the RTT, i.e., there is no one huge contributor that dwarves the rest and is sufficient to optimize for. Building fast distributed systems requires accounting for all these major sources.

A packet sent from one host to another crosses three types of hardware components whose circuitry adds substantial delay, on the order of a few hundred nanoseconds (Table 2.1). At the first host, the CPU writes the packet contents to the NIC over the PCIe bus, whose current generation (PCIe 3.0) has a one-way latency of approximately 200 ns. The NIC takes around 200 ns to process a packet before placing it on the wire. The packet crosses one or more switches on its path to the destination host, each of which adds around 300 ns of one-way port-to-port latency, in addition to any queueing delay due to congestion in the switch.

In addition, the cables used to interconnect NICs and switches add propagation delay. The signal speed in these cables (copper, or fiber-optic) is around two-thirds the speed of light, so one meter of cable adds 5 ns of propagation delay. Within a rack, the cables used to connect hosts to the ToR switch are short (2–3 meters), with a one-way propagation delay of around 10 ns. A round trip between two hosts in a rack includes four PCIe bus crossings, four NIC crossings, four short cable crossings, and two switch crossings. The total latency is therefore $(4 \times 200) + (4 \times 200) + (4 \times 10) + (2 \times 300) = 2240$ ns, or around 2.2 µs, which is close to the latency that we measure in practice (Chapter 6). Propagation delay is higher if the communicating hosts

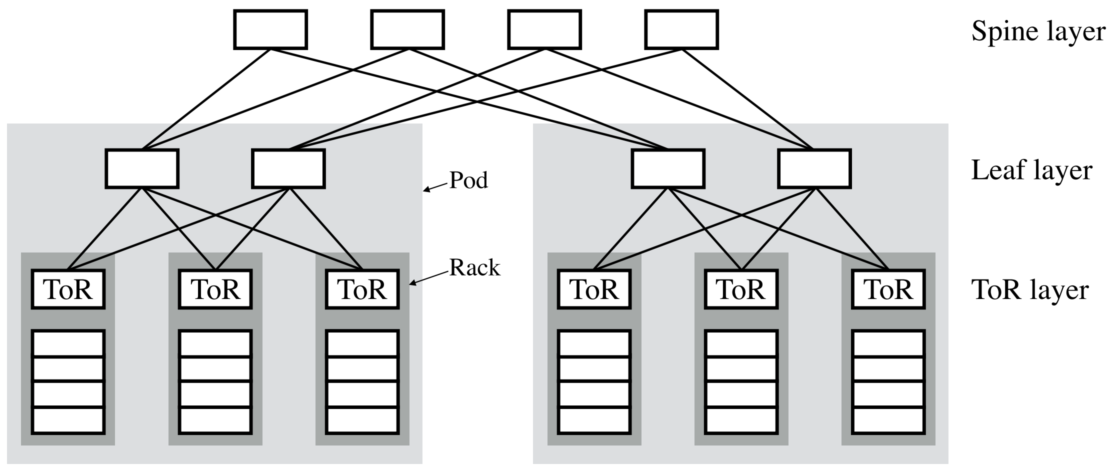


Figure 2.2: Layout of switches in a fat-tree datacenter, reproduced from Guo et al. [55]. The dark grey boxes are racks, containing several hosts and one top-of-rack (ToR) switch. The light grey boxes are pods.

are in different pods. The links between leaf switches and spine switches are ~200-meter fiber optic cables [55], which add around 1 µs of one-way propagation delay.

Network bandwidth and message rate. Hosts within a rack can communicate at the full bandwidth of their ToR links. To keep network costs low, operators typically oversubscribe ToR switches, meaning that these switches have higher cumulative bandwidth to the servers in their rack than to the leaf switches. Oversubscription ratios ranging from as 2:1 and 6:1 are commonly used. Each host can send and receive tens of millions of packets per second. A 100 Gbps Ethernet link can theoretically transmit ~150 million packets per second (Mpps), and such message rates are attainable in practice. Therefore, the end-host networking subsystem should ideally be capable of handling over a hundred million packets per second.

2.1.2 Userspace networking

State-of-the-art datacenter network hardware allows hosts to communicate at 100 Gbps, send over a hundred million packets per second, and complete round trips in a few microseconds. Bringing the advantages of fast network hardware to application software, however, required changing the way in which applications access the network. Historically, the kernel managed applications' access to the network via the sockets system call application programming interface (API) that uses the kernel's heavyweight network stack. Despite continual efforts to improve OS network stacks, the improvements did not keep up with improvements in network speed.

As 10 Gbps Ethernet gained widespread adoption in Internet datacenters, the overhead of heavyweight kernel stacks started limiting application performance. Rizzo [137] showed that, in 2012, an application running on the FreeBSD kernel took around 1000 ns to transmit one UDP packet using the `sendto()` system call. Communication-intensive applications that rely on kernel stacks spend a large percentage of their CPU cycles in the kernel, not executing useful application logic, e.g., 80% in the popular lighttp HTTP server [74], and 75% in a popular memcached in-memory key-value store [15].

For more CPU-efficient networking, kernel-bypass networking—a widely-used approach in HPC—is gaining traction in Internet datacenters. Kernel-bypass network stacks run entirely in userspace, reducing or eliminating three sources of overhead that are present in kernel stacks:

1. **System call overhead.** With kernel stacks, applications must use system calls to access the NIC. System calls require expensive user-to-kernel context switches, whose previously-high ~50 ns cost recently increased to ~200 ns due to kernel page table isolation patches for the Meltdown [98] security vulnerability [91].

   With userspace stacks, applications can send and receive packets without kernel involvement, except during initial setup. The initial setup consists of mapping the NIC's packet I/O queues and registers into application memory. After this is done, packets can be sent or received with cheap memory-mapped I/O instructions. Modern NICs typically support multiple isolated queues, allowing safe access from multiple processes.

2. **Interrupt overhead.** With kernel stacks, the NIC generates interrupts on receiving packets. Although interrupts can be coalesced, in the worst case, one interrupt is generated per received packet. The CPU cycle cost of handling an interrupt with current kernels is approximately 2500 ns [78].

   Userspace stacks typically disable interrupts, and check for received packets by polling the NIC's memory-mapped packet receive queue. This results in high performance when packets are being regularly received, at the expense of wasted CPU cycles during periods when few packets are received. Although our work relies on busy-polling, recent work by Kaffes et al. [78] and Ousterhout et al. [123] shows that it may be possible to achieve the best of both worlds, i.e., the performance of busy-polling during high load, and the efficiency of interrupts during low load.

3. **Hardware-generality.** General-purpose OS stacks support a wide variety of network types (e.g., WiFi, wide-area networks, Bluetooth, etc) with vastly different bandwidth, latency, routing, and packet loss characteristics. They run on a range of computers, from embedded devices with limited CPU and memory, to high-end datacenter servers. This generality comes at the cost of performance.

   Userspace stacks typically aim to support only high-speed datacenter networks, so they may sacrifice hardware-generality for performance. For example, it is safe to assume that datacenter servers have sufficient DRAM to pre-allocate memory for packet buffers, whereas this assumption might not work well in embedded devices.

The end result from these three classes of optimization is that userspace stacks such as DPDK [37] can handle 20–30 million packets per second with one CPU core, over an order of magnitude higher than OS stacks. With a few CPU cores, multi-threaded applications can handle the 150 million packets per second supported by a 100 Gbps link.

High-speed packet I/O alone is rarely sufficient for building real applications. A full transport layer provides a more convenient abstraction by hiding the details of message fragmentation, packet retransmission, congestion control, and buffer memory management. Applications that use plain packets sometimes choose to embed some transport-related functionality at the application level [119], but doing so breaks modularity, thereby preventing code reuse, and increasing complexity. eRPC provides a modular transport layer with all these features, leaving few reasons for re-implementing transport functionality at the application level.

2.1.3 In-network offloads for distributed systems

Several types of devices or technologies may be deployed in datacenter networks to speed up distributed systems by shifting data processing or transport-related tasks away from CPUs. We next provide an overview of four such technologies that are currently seeing use—RDMA, lossless networks, programmable NICs, and programmable switches—along with their strengths and shortcomings when used in distributed systems.

2.1.3.1 Remote Direct Memory Access

RDMA, by definition, is a NIC hardware feature that allows a client host to directly read or write the memory of a remote server host without involving the CPU at the remote host. RDMA has been a common feature in HPC clusters for decades, starting from the Virtual Interface Architecture (VIA) [41]. Today, supercomputers typically use special RDMA-capable interconnects (e.g., InfiniBand [64], or Cray's Aries interconnect [7]), which have their own suite of layer-1 through layer-4 protocols. In Internet datacenters, however, IP and Ethernet are the de-facto layer-2 and layer-3 protocols. To allow using RDMA in such datacenters, network vendors created a new protocol called RDMA over Converged Ethernet (RoCE) derived from InfiniBand to run on IP-routed Ethernet.

To distinguish from RoCE, we refer to non-RDMA providing Ethernet networks as *classical* Ethernet. At the time of writing, RDMA NICs and classical Ethernet NICs are priced similarly.

RDMA NICs, including RoCE NICs from vendors such as Mellanox [108] and Broadcom [22], typically provide two features in addition to server CPU bypass that reduce CPU cycles spent in communication at both the client and the server host. They provide kernel bypass, meaning that the operating system kernel at either host is involved only during initial RDMA setup. The NICs also implement the transport layer in hardware, including reliable and ordered packet delivery, and congestion control.

To expose a region of server memory for RDMA from clients, an application at the server registers a region of memory with the RDMA NIC. A client that wishes to access this region establishes an RDMA connection with the server over an out-of-band communication channel. During this handshake, the client application receives the remote region's virtual address and a security key from the server. In addition, NICs at both the server and the client allocate and initialize transport contexts for the connection.

To write a local buffer into the server's memory, the client application issues an RDMA-write work request to its NIC, specifying the connection, the local buffer's address, and the destination address the server. The client's NIC fetches the local buffer over PCIe using Direct Memory Access (DMA) and sends it to the server. On receiving the buffer, the server's NIC DMA-writes it to the destination address. On completing the transfer, the client's NIC DMA-writes a completion entry to the application. Transport engines in the client and server NIC handle message fragmentation and reassembly, acknowledgments, retransmission, and congestion control. Only the client's CPU spends cycles throughout the RDMA write: first for writing a small work request to the NIC, and then for processing the completion. No unnecessary copies of the buffer are made, resulting in a *zero-copy* transfer.

We provide a more detailed description of RDMA's different transports and capabilities in

Chapter 3. The InfiniBand architecture specification [64] is a comprehensive reference.

Strengths. RDMA improves speed and efficiency primarily when used for *regular* memory reads and writes that access one contiguous region of remote memory per round trip. For example, virtual machine migration requires transferring gigabytes of data between DRAM of two hosts. RDMA allows such a transfer at peak network bandwidth and near-zero CPU cycles. Another example is memory disaggregation, wherein hosts expose parts of their DRAM over the network. Remote hosts whose memory requirement exceeds their memory capacity can use the exposed DRAM as slow memory or swap space [4, 52].

Shortcomings. RDMA has three primary shortcomings that reduce its usefulness in distributed systems. First, the limited flexibility of RDMA forces designers to add inefficiency or complexity to the system's design in order to compensate for the lack of rich operations. For example, as discussed in Section 1.2, because RDMA is limited to one remote memory access per round trip, using RDMA for remote data structure access necessitates additional round trips per access. Similarly, RDMA's lack of first-class support for concurrency control and persistence forces a cumbersome design in systems that deal with concurrent data modifications and non-volatile memory, respectively.

Second, the per-connection RDMA communication state is kept in small on-NIC caches, reducing scalability (Section 1.4). Third, current RDMA NICs require a lossless link layer (discussed next), which is challenging to deploy at datacenter scale, limiting RDMA's deployment to only a few companies, such as Microsoft [55].

2.1.3.2 Lossless networks

Commodity datacenter networks are lossy, meaning that switches regularly drop packets due to buffer overflow caused by congestion. Lossless packet delivery is a link-layer feature that prevents congestion-based packet drops. On Ethernet networks, if Priority Flow Control (PFC) [63] is enabled, a link's receiver prevents its buffer from overflowing by sending a pause frame to the link's sender before the buffer overflows. HPC interconnects such as InfiniBand typically use credit-based link-level flow control [64], in which the link's sender transmits data only if the receiver has previously indicated sufficient buffer room, i.e., the sender has sufficient credits.

Strengths. Lossless link layers have two primary advantages. First, they simplify the transport layer's implementation. Since packets are lost extremely rarely (e.g., during network hardware failures), the transport layer may keep its packet loss recovery logic on a slow path, simplifying the fast path; such simplification can be necessary for an efficient hardware implementation. For example, RDMA NICs often exclude packet loss recovery logic from their transport engine ASIC circuits, instead implementing the recovery logic in slow firmware [165].

Second, lossless link layers potentially reduce tail latency for applications by reducing retransmission timeouts experienced by end hosts, which add a large amount of tail latency. End hosts typically must wait for tens of milliseconds before retransmitting a packet that they suspect to be lost. This is because datacenter switches can add several milliseconds of queueing delay. For example, a 100 GbE Broadcom Trident 3 switch can queue up to 32 MB of data behind a congested egress port, which takes 2.6 ms to drain. Because there may be several such congested switches in the network, avoiding spurious retransmissions that increase congestion requires a 10–100 millisecond retransmission timeout. With a lossless link layer, there are no retransmissions in the absence of hardware failures, avoiding the retransmission timeout problem altogether.

Shortcomings. Link layer losslessness comes with several fundamental problems. Cyclic buffer dependencies can cause deadlocks, causing the network to come to a halt. Because entire links are paused, packets taking an uncongested path may get stuck due to a paused link, causing head-of-line blocking. Switch configuration becomes more complex, due to buffer reservation and congestion marking parameters that depend on buffer reservation [165]. Mittal et al. [116] discuss these problems in detail. In our experience, most Ethernet datacenter operators are unwilling to deploy PFC due to these problems. Since current RDMA NICs depend on a lossless network for good performance, these operators also do not deploy RDMA. Some datacenter operators, including Microsoft [55], have deployed PFC at scale to support RDMA.

2.1.3.3 Programmable NICs

Adding new capabilities into NICs is one path towards high-speed distributed systems. Current NICs span a wide spectrum of functionality. One the one extreme, the average NIC in a datacenter is a simple interface for sending and receiving datagrams on the network; our work shows that it is possible to build fast distributed systems with such NICs alone. In the middle of the spectrum, RDMA-capable NICs provide a fixed set of additional capabilities: one remote memory access per round trip, and a hardware transport layer. Programmable NICs are at the other extreme of high functionality, providing near general-purpose processing inside the NIC.

Programmable NICs add some DRAM and a programmable chip—either an FPGA or a multicore CPU with simple, low-power cores—to the NIC board, in addition to the fixed-function NIC ASIC. The NIC's programmable chip can handle requests received at the host using on-NIC processing and memory, and send a response without involving the CPU at the host. Examples of FPGA-based programmable NICs include Microsoft's Azure SmartNIC [47] and Mellanox's Innova [109]. Examples of multicore CPU–based programmable NICs include Amazon's Nitro [11], Mellanox's BlueField [107], Marvell's LiquidIO [106], and Netronome's Agilio [118]. The CPUs in these NICs use either commodity ARM or MIPS cores, or proprietary cores specialized for packet processing.

Strengths. The deployment of programmable NICs accelerated because of their applicability to fast network virtualization in cloud providers such as Microsoft Azure and Amazon Web Services [11, 47]. In cloud datacenters, virtual machines (VMs) need fast networking, but their access to the physical network must be mediated by a programmable layer that is trusted by the datacenter operator. Traditionally, this was done in software by the hypervisor. Pushing network access mediation to NIC hardware improves performance by allowing VMs direct network access without expensive VM exits to the hypervisor.

When used as devices to handle distributed system operations (e.g., in-memory hash table accesses), programmable NICs typically provision dedicated hardware resources for the application. As a result, they may provide more predictable latency than handling the operations on a host CPU, which is typically shared by many applications. However, the benefit of latency predictability is reduced by other sources of latency variation. Request latency will be higher during network congestion, regardless of whether the end-host device handling the operation is a programmable NIC or a CPU. In addition, the client issuing the operation is typically software running on a CPU, so handling the operation in the server's NIC eliminates only part of the latency variation caused by using CPUs.

Shortcomings. For remote data structure access, programmable NICs offer a seemingly better alternative to RDMA because they can perform multiple memory accesses while processing a request. However, host memory accesses from a programmable NIC go over the slow PCIe bus, which has higher latency (~500 ns) than the host CPU's connection to its DRAM (~80 ns).[1] As a result, programmable NICs suffer from latency amplification due to multiple PCIe round trips, similar to RDMA's latency amplification over the datacenter network. In addition, similar to RDMA NICs, programmable NICs lack first-class support for concurrency, memory ordering, and durability.

2.1.3.4 Programmable switches

Historically, network switches were fixed-function: they examined a fixed set of header fields (e.g., IP address), and implemented a fixed set of actions on packets (e.g., forwarding or dropping a packet, or decrementing a packet's time-to-live field). The use of Software-Defined Networking in datacenters demanded *programmable* switches that can process arbitrary header fields, and perform more general-purpose actions [20]. Such switches, such as Barefoot's Tofino switches [146] that are programmed in the P4 switch programming language [125], are currently being deployed in some datacenters.

Packets arriving at a programmable switch's port are queued into the port's ingress pipeline in the switch ASIC. The pipeline's programmable parser extracts the packet's header, and passes it through a series of programmable match-action tables. Both the parser and the match-action tables are specified by the switch program (e.g., the P4 program). Each match-action stage examines a fixed set of fields from the packet header. If the fields match an entry in the stage's table, the switch takes a simple action specified by the entry (e.g., modify a header field, or increment an in-switch counter). After ingress processing, the header is routed to an egress port by the switch's crossbar. The egress port's pipeline may include more match-action stages before the packet is transmitted.

Strengths. Currently, the primary use cases of programmable switches in industry are in network management, e.g., improving network visibility through telemetry and real-time measurements, and improving network reliability by eliminating unneeded features that are typically included in fixed-function switches [13]. These use cases are orthogonal to the class of

[1] Although programmable NICs have some on-board DRAM, most of the host's DRAM is connected to the CPU.

distributed systems studied in this thesis.

By virtue of their placement in the network, programmable switches offer a unique ability to reduce the number of network hops in a distributed protocol, improving latency [92, 93, 130]. For example, in Paxos and other related leader-based state machine replication protocols [85], completing a replication request requires four network hops when replicas are failure-free. The client first sends its request to the leader replica, which assigns a sequence number to the request, and forwards it to follower replicas. Each follower replies to the leader, which replies to the client after collecting replies from followers. Li et al. [92] show how using a programmable switch as the leader can reduce the number of effective network hops to two.

Shortcomings. The limited computational and memory flexibility of the match-action programming model leads to complex designs that shoehorn application logic into switches. There are also open challenges that must be solved before developers can safely use programmable switches in applications beyond network management. First, it is unclear how stateful algorithms in recent in-switch applications [76, 92, 93] handle in-switch parallelism. Today's programmable switches use multiple independent match-action pipelines to satisfy bandwidth demands. There is no support for inter-pipeline state sharing or concurrency control because the need to run pipelines at line rate precludes such heavyweight functionalities. Second, how can we safely provide control over shared network switches to application developers? For instance, how should we safely and effectively share switch SRAM among packet buffering, and memory allocated to different in-switch applications?

2.2 Communication-intensive distributed applications

Core services in datacenters are typically distributed systems, which run on anywhere between a few hosts to thousands of hosts. This section first gives an overview of three examples of the types of communication-intensive applications that we target in this dissertation. Then, we describe two common workload characteristics across the three applications that challenge the communication subsystem: small messages, and short per-message processing.

2.2.1 Main-memory key-value stores

Main memory–based key-value stores and caches are widespread in large-scale Internet services. They are used both as primary stores (e.g., Redis [134]), and as caches in front of backend, persistent databases (e.g., Memcached [112]). At their most basic level, these systems export the traditional key-value interface, with `GET`, `PUT`, and `DELETE` calls. The key-value items are partitioned across hosts, typically using a scheme such as consistent hashing [82] with key hashes. Internally, they use a variety of data structures to provide fast, memory-efficient access to their underlying data (e.g., hash table or tree-based indexes).

Distributed in-memory key-value stores are an important building block for large-scale web services. For example, Facebook's Memcached deployment consists of thousands of machines and acts as an object cache for trillions of data items [119].

2.2.2 Distributed transaction processing

Distributed transactional data stores with ACID guarantees (Atomicity, Consistency, Isolation, and Durability) are the primary persistent data stores in datacenters. Examples include MySQL at Facebook, Google's BigTable, and Microsoft's SQL server [25, 119]. A transaction is a sequence of reads and writes to items in the data store, delimited by transaction begin and end commands. A distributed transaction processing system with ACID guarantees provides a powerful programming abstraction for designing distributed systems such as object stores and online transaction processing (OLTP) systems. Such a system provides the illusion of one centralized database, in which transactions commit durably in sequence.

In the past, with slow networks and storage technologies, distributed databases sacrificed either transaction support, or weakened transaction consistency guarantees [39]. For example, BigTable provides only single-row transactions, and Amazon's Dynamo [34] provides only eventual consistency instead of strong consistency. Recently, Dragojević et al. [39] showed that with the availability of fast networks and non-volatile memory in datacenters, distributed transactions with strong consistency guarantees can achieve good performance, supporting millions of distributed transactions per second with around 100 hosts.

Modern Internet services handle such transaction rates in practice. For example, Amazon reports that their Aurora database handled 148 billion transactions over two days worldwide [8]. This transaction rate corresponds to 0.85 million transactions per second on average. Due to skew in transaction rate, it is reasonable to expect that their database handled many millions of transactions per second during some periods of high activity.

2.2.3 State machine replication

State machine replication (SMR) is used to build highly available services, such as metadata stores or lock servers. An SMR service consists of a group of server hosts 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. Paxos [85] and Raft [121] are widely-used SMR protocols that take a *leader*-based approach: Absent failures, the SMR 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 acknowledgments from a majority of replicas.

Despite its seeming simplicity, SMR is difficult to design and implement correctly [59]: the protocol must have a specification and a proof (e.g., in TLA+), and the implementation must adhere to the specification. A fast and general-purpose communication library like eRPC allows using an existing well-tested SMR implementation, avoiding the high complexity and cost of re-implementing or redesigning an SMR system from scratch for new in-network hardware technologies.

2.2.4 Common application workload characteristics

Communication-intensive applications, including the three examples discussed above, impose two challenging requirements on the communication subsystem.

1. **Small messages.** These systems handle primarily small messages, ranging from tens to a few hundred bytes. In Facebook's Memcached deployment [9], the most frequently accessed cache pool ("USR"), has keys up to 21 B, and virtually all values are 2 B. In addition, 90% of all cache space is allocated to values smaller than 500 B. In the industry-standard TPC-C benchmark for transactional databases, the entries in each database table are smaller than 320 B [148]. SMR systems are typically used for serving small locks and metadata objects.
2. **Short per-message application processing.** Messages in communication-intensive systems require tens of nanoseconds to a few microseconds of application-level processing. Accessing a main-memory hash table or a tree-based data structure takes only tens to hundreds of nanoseconds [89]. An RPC in a distributed database or SMR system typically requires either a cheap access to a main-memory key-value store, or persisting a transaction's update log record. With the availability of non-volatile memory technologies such as Intel's DC Persistent Memory [67], the latter requires only hundreds of nanoseconds to a few microseconds.

It is challenging to design a communication library that performs well for small messages and short per-message application-level processing. Doing so requires attention to numerous subtle factors, such as user-kernel crossings, PCIe overheads, NIC architecture details, CPU cache misses, buffer management, etc. These factors are less important for workloads with large messages or high per-message processing. For these workloads, performance depends primarily on simpler factors. They are likely to be bottlenecked by network bandwidth or application-level processing, or the number of times (typically zero or one) large packet buffers are copied,

2.3 Evaluation clusters

To demonstrate that the techniques, optimizations, and system designs presented in this dissertation are general and not dependent on particular hardware platforms, we evaluate them on several different clusters spanning a wide range of hardware technologies. Table 2.2 lists the specifications of these clusters. Notably, the NICs used in our evaluation span almost a decade of hardware, and they include both InfiniBand and Ethernet. We name each cluster using an abbreviation of the cluster's NIC model.

Cluster name	**CX**	**CX3**	**CIB**	**CX4**	**CX5**
Number of nodes	10	200	11	100	8
Network type	InfiniBand	InfiniBand	InfiniBand	Ethernet	Ethernet
Mellanox switch	InfiniScale IV	SX6036	SX6036	SN2410/SN2100	SX1036
CPU model	Opteron 8354	E5-2450	E5-2683 v3	E5-2640 v4	E5-2697 v3
Core count, frequency	4c, 2.2 GHz	8c, 2.1 GHz	14c, 2.0 GHz	10c, 2.4 GHz	14c, 2.6 GHz
NIC release date	2008	2011	2012	2014	2016
Mellanox NIC	ConnectX	ConnectX-3	Connect-IB	ConnectX-4 Lx	ConnectX-5
NIC ports and speed	1x 20 Gbps	1x 56 Gbps	2x 56 Gbps	1x 25 Gbps	2x 40 Gbps
PCIe link	PCIe 2.0 x8[2]	PCIe 3.0 x8	PCIe 3.0 x16	PCIe 3.0 x8	PCIe 3.0 x16

Table 2.2: Evaluation clusters used in our work. CX, CX3, and CX4 are public clusters that are part of NSF PRObE, Emulab, and CloudLab, respectively [51, 136, 156]. CIB and CX5 are private clusters at NetApp and Carnegie Mellon University, respectively. CPU models with names starting with "E5" are Intel Xeon CPUs with two-way Hyper-threading.

2.4 Open-source code

The source code for all systems and experiments presented in this dissertation is available online.

- Chapter 3: `https://github.com/efficient/rdma_bench`
- Chapter 4: `https://github.com/efficient/HERD`
- Chapter 5: `https://github.com/efficient/fasst`
- Chapter 6: `https://github.com/erpc-io/eRPC`

"There are many highly respected motives which may lead people to prosecute research, but three which are much more important than the rest. The first (without which the rest must come to nothing) is intellectual curiosity, desire to know the truth. "

G. H. Hardy

Chapter 3

Guidelines for use of modern high-speed NICs

Designing fast end-to-end communication software for datacenter networks requires using NICs efficiently. Modern NICs are complex devices, with sophisticated in-NIC computation and memory subsystems, including packet processing engines, DMA engines, and memories for various data structures and metadata. Combined, these subsystems provide a wide array of knobs that govern performance of even basic Ethernet packet I/O. In addition, most current NICs go beyond packet I/O to provide a hardware transport layer and Remote Direct Memory Access.

This chapter covers two contributions. First, a central question for our work is choosing the right NIC primitives and knob configurations for RPCs. This chapter describes the experiments and evaluation results that back up our choice of involving CPUs in distributed systems, and the design of our RPC subsystems. For example, we illustrate why it is beneficial to use plain packet I/O instead of RDMA writes for implementing RPCs.

The second contribution is a set of high-level guidelines aimed at providing researchers and developers with a roadmap through the large design space of NIC primitives and knobs, without necessarily becoming NIC gurus. A lesson from our work is that low-level hardware factors, such as individual PCIe messages and NIC hardware architecture, are surprisingly important for fast communication. Reasoning about these factors is challenging, in part because doing so has traditionally required expensive or confidential resources, such as PCIe analyzers and proprietary NIC manuals. As a result, these factors have received little attention from the research community. This chapter aims to fill this gap.

These guidelines are backed by an open-source set of measurement tools (https://github.com/efficient/rdma_bench) for evaluating and optimizing the most important system factors that affect performance when using NICs. This chapter evaluates the effectiveness of our guidelines and optimizations using simple microbenchmarks. In later chapters, we show how the RPC subsystems in HERD, FaSST, and eRPC benefit from these guidelines. In those chapters, we refer back to our guidelines using the Guideline label.

We divide our guidelines into two broad classes.

1. We show that reducing traffic over the PCIe link mitigates PCIe bandwidth and latency bottlenecks that arise frequently in high-speed communication. During packet I/O, in addition to packet payloads, metadata such as PCIe bus headers, buffer descriptors and completions are also transferred over the PCIe bus. These metadata are not strictly necessary for application functionality. We present and evaluate several optimizations aimed at reducing PCIe traffic.
2. Second, we show that modern NICs' parallel architecture offers both opportunities and pitfalls. Today's datacenter NICs are composed of multiple processing units (PUs), such as packet processing engines and DMA engines. We develop techniques for exploiting intra-NIC parallelism, and show how and when doing so can be beneficial. We also show how NIC primitives that require synchronization among PUs, such as atomic operations, can reduce performance.

We begin our journey into modern NICs with an overview of the PCIe interconnect and NIC operation.

3.1 A review of PCI Express

PCI Express is a point-to-point interconnect used to connect CPUs to peripheral devices, such as NICs, GPUs, storage drives, and FPGAs. PCIe endpoints typically communicate by writing to each other's memory over the PCIe link, using PCIe protocol packets [128]. As shown in Figure 2.1, the NIC at each host in a datacenter is attached to the PCIe controller of the host's CPU, which is located on the same die as the CPU cores. The PCIe controller reads and writes the CPU's L3 cache to service the NIC's memory access requests.

The current widely-deployed PCIe generation is PCIe 3.0, which provides approximately 1 GB/s per lane, unidirectionally. Datacenter NICs typically use 8 or 16 PCIe 3.0 lanes (i.e., PCIe 3.0 x8 or PCIe 3.0 x16 links, respectively), for a total of 8 GB/s or 16 GB/s, respectively. We have evaluated the effectiveness of our PCIe-related guidelines on both the previous PCIe 2.0 generation, and PCIe 3.0. We expect our guidelines to also apply to hosts with the upcoming PCIe 4.0 generation, which doubles the per-lane bandwidth from PCIe 3.0 to 2 GB/s. This is because upcoming 200 GbE NICs stress the bandwidth of even PCIe 4.0 x16, so reducing PCIe use remains important. In addition, several of our optimizations target other characteristics of PCIe, such as latency, PCIe header overhead, and CPU cycle use. These characteristics remain largely unchanged with PCIe generation updates.

3.1.1 PCIe headers

Reasoning about PCIe performance in communication-intensive datacenter applications requires understanding the overhead of PCIe packet headers. This is because such applications generate primarily *small* PCIe packets, in which the size of PCIe headers is comparable to the

	PCIe 2.0	PCIe 3.0
Per-lane bandwidth	500 MB/s	985 MB/s
PCIe transaction request header size	24 B	26 B
PCIe transaction completion header size	20 B	22 B

Table 3.1: Lane bandwidth and header sizes for PCIe 2.0 and PCIe 3.0.

size of useful application payload. The PCIe packets are small for two reasons. First, as discussed in Section 2.2, the requests and responses in our target applications are primarily small, ranging from tens to a few hundred bytes. Second, CPUs write to NICs using memory mapped I/O (discussed in the next section), which generates PCIe packets with cacheline-sized, 64 B payloads.

PCIe is a layered protocol, consisting of physical, link, and transaction layers. The total header size is 20–26 B, depending on the PCIe generation and the transaction type. Table 3.1 lists the bandwidth and header overhead for the PCIe generations in our clusters.

CPU-NIC communication generates three types of PCIe transaction layer packets (TLPs): read requests, read completions, and write requests (there is no transaction-layer completion response for a write). For example, the NIC may write data to the CPU's memory by sending a write request transaction packet to the CPU.

3.1.2 Memory-mapped I/O and Direct Memory Access

Communication between CPUs and NICs uses one of two methods, with different latency, CPU cycle use, and bandwidth tradeoffs. CPUs access NIC memory using memory-mapped I/O (MMIO). NICs access CPU memory using Direct Memory Access (DMA). A NIC typically issues DMAs while servicing a request received from the network, or in response to an MMIO write received from its host CPU. For example, CPUs typically initiate packet transmission by writing a packet's address to the NIC; the NIC then DMA-reads the packet, and sends it on the network. Understanding MMIO and DMA along with their performance tradeoffs is important for designing fast communication software.

Memory-mapped I/O. To use MMIO, the application first uses the NIC device driver to map a portion of the NIC's memory into the application's address space. Then, the application can write to the NIC's memory with store instructions. To avoid generating a PCIe write for each store instruction, CPUs use an optimization called "write combining," which combines stores to generate cache line–sized PCIe transactions.

An MMIO write is the lowest-latency method for a CPU to transfer data to the NIC. However, it consumes CPU cycles, and incurs high PCIe header overhead because all PCIe packets have only cacheline-sized payloads. In several situations, the DMA approach reduces CPU cycles and bandwidth overhead.

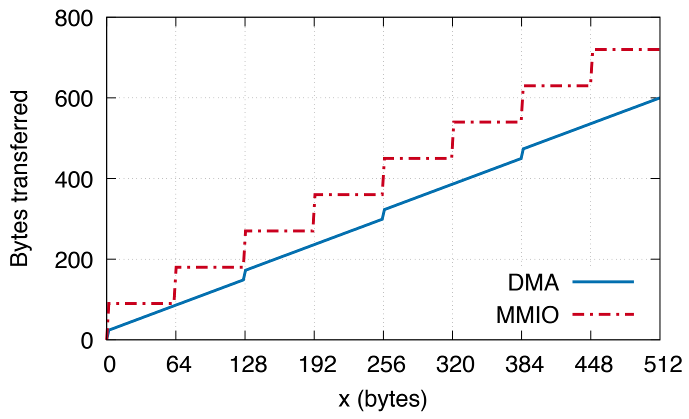


Figure 3.1: CPU-to-NIC PCIe traffic for an x-byte transfer with DMA and MMIO, assuming PCIe 3.0 and C_{rc} = 128 bytes.

Direct Memory Access. NICs have DMA engines that can access the CPU's memory subsystem without involving CPU cores. To transfer data to the NIC, the CPU writes just the buffer's descriptor (i.e., its address and length) using to the NIC using MMIO. The NIC then DMA-reads the buffer. This results in a CPU-efficient transfer, because CPU cycles are consumed only for the small descriptor MMIO, not for the entire buffer.

DMA is often more bandwidth-efficient than MMIO because DMA transfers are not restricted to cache line units. For example, a NIC can write up to 4 kB to the CPU's memory in one DMA write packet. Large DMA reads, however, are split into smaller packets. To read from the CPU's memory, a NIC sends a PCIe read request TLP to the CPU, whose PCIe controller replies with one or more read completion TLPs. A read completion TLP with a payload larger than the CPU's read completion combining size (C_{rc}) is split into multiple completions. We use the CX3, CIB, and CX clusters (Table 2.2) for measurements in this chapter. C_{rc} is 128 B for CX3 and CIB [70, 71]; we assume 128 B for the CX cluster because we did not find a publicly-available value for its CPU.

A DMA read always uses less CPU-to-NIC PCIe bandwidth than an equal-sized MMIO write; Figure 3.1 shows an analytical comparison. With MMIO, the buffer transfer uses TLPs with exactly 64 B payloads, which is inefficient compared to read completion TLPs with variable-sized payloads up to C_{rc} bytes in size. This factor is important, and we show how it affects performance of higher-layer protocols in the subsequent sections.

3.2 How modern NICs work

The performance of communication software depends on how it uses the NIC. This section provides an overview of modern NICs' operation that is common to both the traditional Ethernet packet interface and the RDMA interface. The next section reviews RDMA.

NICs provide a simple abstraction of multiple transmit and receive (TX and RX) queues.

Each queue consists of multiple slots that contain *descriptors*. To initiate packet transmission or reception, the NIC driver creates descriptors in the TX or RX queue resident in CPU memory, and transfers them to the NIC using either MMIO writes or DMA reads. We discuss the tradeoffs of these two methods in Section 3.6.2. On completing the work for a TX or RX descriptor, the NIC signals completion by DMA-writing a completion entry (CQE) to a completion queue (CQ) associated with the TX or RX queue.

Descriptor format and size is specific to the NIC vendor and model, and depends on several factors, including the operation type, transport, optimization flags, and operation initiation method. A minimal TX queue descriptor contains the address and length of a packet in host memory; the NIC fetches the packet using a DMA read, and places it on the wire. A minimal RX queue descriptor contains the address and length of a buffer in host memory where the next packet received by the NIC for that queue is DMA-written to. More sophisticated descriptors may be larger. For example, the TX descriptor may include the payload for small packets, which reduces latency by avoiding the DMA read used to fetch the payload.

Note that, in RDMA parlance, queue descriptors are also called Work Queue Elements (WQEs). We use the more general term descriptors to refer to both WQEs and Ethernet RX/TX queue descriptors.

3.3 RDMA terminology

Support for RDMA in datacenters is on the rise. At the time of writing, most major NIC vendors, including Mellanox, Intel, and Broadcom provide RDMA support in their state-of-the-art models. Understanding RDMA performance is therefore important for designing high-speed communication for distributed systems. While we chose to not rely on RDMA support in our final RPC design, we include guidelines and measurements for RDMA NICs in this chapter. Section 2.1.3.1 provides high-level background on RDMA. We provide more relevant details below.

3.3.1 RDMA verbs

Userspace programs access RDMA-capable NICs directly using functions called RDMA verbs. While there are several types of verbs, the ones most relevant to this work are RDMA reads, RDMA writes, RDMA atomics, sends, and receives. We abbreviate these verbs as READ, WRITE, ATOMIC, SEND, and RECV, respectively.

READs, WRITEs, and ATOMICs operate directly on remote memory, bypassing the remote CPU. Because these verbs involve only the initiator's CPU, they are termed *one-sided*. In contrast, SENDs and RECVs are called *two-sided* verbs, because both the initiator's CPU and the remote CPU is involved. A SEND's payload is written to a remote memory address that is specified by the remote CPU in a pre-posted RECV. This is similar to packet I/O semantics offered

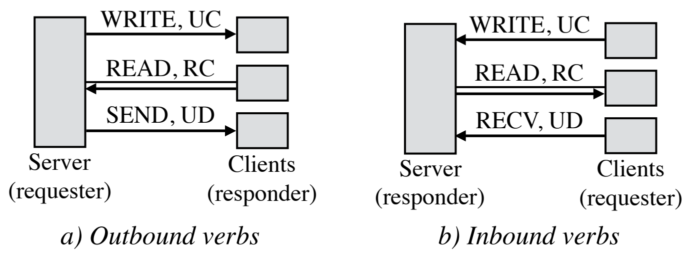


Figure 3.2: Inbound and outbound verbs at the server.

by non-RDMA NICs, except that the data transferred by an RDMA SEND or RECV may be multiple packets in size. SEND/RECV verbs are also called *messaging* verbs.

3.3.2 RDMA queue pairs

Verbs are posted by applications to RDMA queues. Queues always exist in pairs: a send queue and a receive queue form a queue pair (QP). Each queue pair has an associated completion queue (CQ), which the NIC fills in upon completion of verb execution. RDMA-enabled hosts post operations to QPs using a userspace verbs provider library, such as *libibverbs* in Linux. READs, WRITEs, ATOMICs, and SENDs are posted to the send queue, and RECVs are posted to the receive queue.

We call the host initiating a verb the *requester* and the destination host the *responder*. For some verbs, the responder does not actually send a response. We distinguish between *inbound* and *outbound* verbs because their performance differs significantly: READs, WRITEs, ATOMICs, and SENDs are outbound at the requester and inbound at the responder; RECVs are always inbound. Figure 3.2 depicts inbound and outbound verbs pictorially.

3.3.3 RDMA transport types

RDMA transports can be connected or unconnected. A connected transport requires a hardware connection between two queue pairs that communicate exclusively with each other. Current RDMA implementations support two main types of connected transports: Reliable Connected (RC) and Unreliable Connected (UC). There is no acknowledgment of packet reception in UC; packets can be lost and the affected message can be dropped. As UC does not generate ACK/NAK packets, it causes less network traffic than RC.

In an unconnected transport, one queue pair can communicate with any number of other queue pairs. Current implementations provide only one unconnected transport: Unreliable Datagram (UD), which is similar to plain packet I/O in classical Ethernet NICs. Because RDMA NICs maintain state for each active queue in their internal SRAM, datagram transports can scale better for applications with a one-to-many communication pattern.

Some transport types support only a subset of the available verbs. Table 3.2 lists the verbs

	Reliable Connected	Unreliable Connected	Unreliable Datagram
Two-sided SEND/RECV	✓	✓	✓
RDMA writes	✓	✓	✗
RDMA reads	✓	✗	✗

Table 3.2: Operations supported by each RDMA transport type

supported by each transport type. UC does not support READs because READs are implicitly acknowledged by the RDMA read response. UD does not support one-sided RDMA verbs.

RDMA networks, including InfiniBand and RoCE, typically employ lossless link-level flow control (Section 2.1.3.2). *Even with unreliable transports (UC/UD), packets are never lost due to switch buffer overflows.* RDMA networks lose packets due to bit errors on the wire and hardware failures, which are extremely rare.

3.4 Preface to the guidelines

We believe that these guidelines apply broadly to the kinds of NIC hardware currently deployed in datacenters. Although the measurements reported in this chapter are for InfiniBand NICs (running both datagram packet I/O and RDMA verbs), the guidelines are applicable to any PCIe-based NIC with an internally-parallel architecture—two requirements that are generally satisfied by high-speed NICs at the time of writing. NICs in today's datacenters are PCIe cards; some vendors are beginning to integrate NICs on-die or on-package [66, 68, 69], but these NICs still communicate with the CPU's PCIe controller using the PCIe protocol, and are less powerful than discrete NICs. The performance requirements of high-speed NICs generally necessitates internal parallelism with multiple processing units [58].

Importantly, we have found that several of our guidelines that benefit InfiniBand's UD transport also apply to packet I/O with classical Ethernet NICs, which eRPC targets. This is because packet I/O on Ethernet NICs works similarly at the PCIe and NIC level to InfiniBand's UD transport. Therefore, the two communication modes have similar performance characteristics, and benefit similarly from our optimizations.

We now present our guidelines. For each guideline (e.g., reduce CPU-initiated MMIOs), we provide insight on both how to determine whether this guideline is relevant (e.g., if an application can issue multiple concurrent RPCs, it can transmit them with one MMIO). We provide specific optimizations to realize each guideline, and discuss the guideline's effectiveness and limitations.

3.5 Guidelines for NICs with transport-layer offload

Some NICs, such as RDMA NICs and TCP offload engines, implement a reliable hardware-based transport layer that guarantees in-order message delivery with zero CPU use. Using such a transport is a seemingly attractive choice for implementing RPCs. For example, RPC designs for RDMA-capable NICs typically use RDMA's Reliable Connected transport [38, 113, 154]. However, our research suggests that such transports are not the best match for RPCs, and that using unreliable datagrams with the transport layer implemented in software provides higher performance, scalability, and flexibility. These benefits arise from two fundamental advantages of the latter approach: the ability to piggyback transport layer acknowledgments (ACKs) on RPC responses, and higher scalability from no per-connection on-NIC state.

3.5.1 Prefer application-level ACKs over transport-level ACKs

Reliable hardware transports use explicit, transport-level acknowledgments. However, for RPCs, it is possible to use fewer network messages by using a request's response as an implicit acknowledgment for the request. This implicit-ACK optimization is not new, originating with Birrell and Nelson's seminal paper on RPCs [18]. Our contribution here is in showing the value of foregoing CPU cycle savings from offloading reliability to NIC hardware in favor of reducing network messages.

With implicit ACKs, the client sends the RPC request over an unreliable transport, such as UDP on Ethernet networks, or RDMA's Unreliable Datagram or Unreliable Connected transport; the server replies using a second message over the unreliable transport. Doing so saves NIC and network resources consumed by explicit acknowledgments, improving performance. Although ACK packets are small, their size and in-NIC processing requirement is comparable to typical RPC packet size in several datacenter applications (Section 2.2). For short, single-packet RPCs, implicit-ACKs reduce the number of packets per RPC from four to two.

Effectiveness. In Section 4.3.2.2, we show that using implicit application-level ACKs instead of RDMA's transport-level ACKs improves the performance of HERD's RPC design by 13%. All three RPC designs presented in this thesis (i.e., HERD RPCs, FaSST RPCs, and eRPC) use implicit ACKs.

Using an unreliable transport does not mean high packet loss: First, on lossless networks, link-level flow control prevents congestion-based packet drops regardless of the transport layer (Section 2.1.3.2). Therefore, unreliable transports get near-reliable packet delivery without paying the cost at the transport level. In addition to link-level flow control, modern high-speed networks often employ forward error correction and link-layer retransmission for bit-error recovery, which benefits unreliable transports, too. Combined, these two techniques make packet loss extremely rare. In our experiments on a 70-node cluster with a lossless InfiniBand network, we observed no packet loss during a 50 petabyte data transfer with Unreliable Datagrams (Chapter 5).

Second, on lossy networks, we show in Chapter 6 that simple end-host software techniques are sufficient to prevent most packet drops. The key insight is that the bandwidth-delay product (BDP) in modern datacenter networks is much smaller than the buffering capacity of datacenter switches. Restricting each flow to one BDP of outstanding data effectively prevents packet loss during even heavy congestion, while allowing flows to achieve peak network bandwidth.

Limitations. An RPC server can send a response with the implicit ACK only after completing request processing, which involves running a user-provided request handler function. As a result, the ACK may get delayed arbitrarily. Congestion control protocols typically extract information about network congestion from ACKs, such as such as Explicit Congestion Notification marks [49] or RTT [21, 88, 115]. Delayed feedback degrades the quality of congestion control. In our target applications (Section 2.2), request handlers are typically short-running, allowing prompt congestion feedback in most cases. Congestion control protocols can handle a modest amount of noise and delay [165], which we believe is sufficient to handle delayed ACKs from a small fraction of long-running RPC handlers in our target workloads. However, implicit ACKs may not work well for other applications in which most RPC handlers are long-running (e.g., requiring tens of microseconds per request).

While the implicit-ACK optimization works well for RPCs, it does not work well for other common communication patterns that lack a response on which ACKs may be piggybacked, such as message passing, or RPC chains [141]. (In an RPC chain, a client sends a request message to a server, which performs some computation, and forwards the message along a chain of servers. The final server in the chain replies to the client.) Note that our other guidelines are not specific to the request-reply communication pattern, and they work well for other patterns.

3.5.2 Avoid storing connection state on NICs

NICs that provide a reliable transport layer implement hardware-based one-to-one connections between communicating entities. A small, on-NIC SRAM (Section 1.4) caches connection state, backed by the CPU's memory subsystem. At large scale, when the number of connections at the host exceeds the NIC's cache capacity, the NIC serves cache misses by fetching connection state from the CPU's memory subsystem over the slow PCIe bus, reducing performance.

Transport-layer offload NICs, including most RDMA NICs and TCP offload engines, use a connection-oriented design because connections are the natural implementation vehicle for transport-layer features such as end-to-end reliability and congestion control. RDMA's connection-oriented design goes back to the Virtual Interface Architecture (VIA), which is a popular model for user-level, zero-copy networking [41], and forms the basis of current commodity RDMA implementations such as InfiniBand and RoCE. VIA architects made the design decision to use connections to simplify NIC implementation. In Mellanox's RDMA implementation, each connection requires around 375 B of state. TCP offload engines typically use 128–256 B of state per connection [31, 46]. NICs have only a ∼2 MB of SRAM, preventing scaling to over a few thousand connections.

A software-based transport layer approaches can scale better by (1) using stateless data-

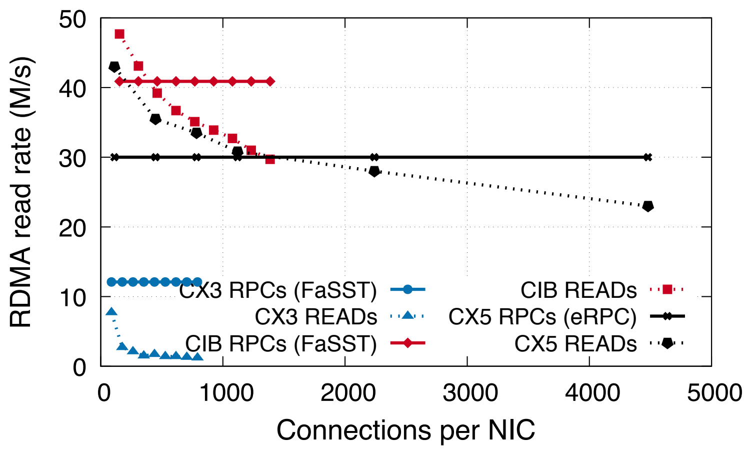


Figure 3.3: Connection scalability of three generations of RDMA NICs. For each NIC, we show how the throughput of 32 B RDMA reads as well as datagram-based RPCs (FaSST RPCs or eRPC), changes when the number of connections is increased.

grams to minimize on-NIC state, and (2) keeping connection state in the CPU's large memory subsystem. While NICs and CPUs will both cache recently-used connection state, CPU cache misses are served from low-latency DRAM, whereas NIC cache misses are served over the slow PCIe bus. The CPU's miss penalty is therefore much lower. Second, CPUs have substantially larger caches (~30 MB) than modern NICs, so the cache miss *frequency* is also lower.

Effectiveness. We measure the scalability of connection-based hardware transports on three generations of RDMA NICs using the following experiment. We use a cluster of hosts. Each host in the cluster creates a tunable number of connections to other hosts in the cluster, and issues either RDMA reads, or RPCs using either FaSST RPCs or eRPC that use stateless datagrams. We describe the experiment in more detail in Section 5.2.3.2.

Figure 3.3 shows the results from this experiment for 32 B RDMA and RPC messages. RDMA's performance drops by at least 40% with fewer than 4500 connections on each cluster, whereas RPC performance remains constant. The performance loss is dramatic on old NICs, such as CX3's ConnectX-3 NICs, which were released in 2011 (Table 2.2). The newer Connect-IB and ConnectX-5 NICs on CIB and CX5, released in 2012 and 2016, respectively, fare better.

Dragojevic et al. [40] and Zamanian et al. [159] hypothesize that improvements in NIC hardware will allow using connected transports at large scale [40, 159]. We believe that such an improvement is unlikely, for fundamental reasons that we described in Section 1.4. As a concrete example, ConnectX-5's connection scalability is 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 [2]. 5000 connections require 1.8 MB, so cache misses are unavoidable. The scalability issue of RDMA is exacerbated by the popularity of *multihost* NICs, which allow sharing a powerful NIC among 2–4 CPUs [3, 120].

NIC vendors have been trying to improve RDMA's scalability for a decade [30, 84], with techniques that do not involve putting more SRAM in NICs. We describe later in Section 5.6.1 why these techniques do not map well to our target workloads.

Among our three RPC designs presented in this thesis, only HERD RPCs require in-NIC connection state, and they do so in a fashion that allows a HERD server to handle a large number of clients: the server uses NIC-managed connections only for receiving requests over RDMA writes. The key insight is that these connections have a small NIC footprint because they carry only inbound traffic. In our later RPC designs (i.e., FaSST RPCs and eRPC), we created mechanisms to preserve the performance of HERD's RDMA write–based requests while using only scalable datagrams.

Limitations. Although all commodity implementations of RDMA use a connection-oriented design, there are niche specifications and implementations for connectionless RDMA. Portals is one such specification [14]; the Bull eXascale Interconnect [35] is a recent hardware implementation of Portals, currently available only to HPC customers. The availability of scalable one-sided RDMA may reduce the at-scale performance advantage of our RPC-based designs compared to RDMA-based designs. However, RPC-based designs will still retain their advantage of fewer round trips, and higher flexibility and simplicity. Further, it is likely that RPC implementations that use scalable one-sided RDMA writes will provide even better performance than our current datagram-based designs.

3.6 Reduce PCIe traffic

3.6.1 Measurement method: PCIe counters on commodity CPUs

Reasoning about and optimizing PCIe traffic requires understanding and modeling the PCIe interaction between NICs and CPUs. Doing so is challenging because a precise PCIe analysis requires difficult-to-obtain resources, including expensive PCIe analyzers, and proprietary/confidential NICs manuals. We contribute a method for PCIe analysis that does not require these resources: we use PCIe counters available on modern CPUs to create a nearly complete model of NIC PCIe behavior. Our analysis primarily uses counters for DMA reads and DMA writes, named `PCIeRdCur` `PCIeItoM`, respectively, on Intel CPUs. For each counter, the number of captured events per second is its counter rate.

CPU PCIe counters are imprecise: The CPU counts PCIe events by intercepting cache line–level activity between the PCIe controller and the L3 cache, so the counters can miss some critical information. For example, the counters indicate two PCIe DMA reads when the NIC reads a four-byte chunk straddling two cache lines.

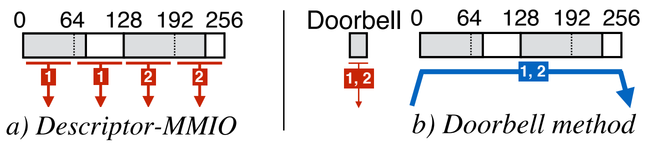


Figure 3.4: The Descriptor-MMIO and Doorbell methods for transferring two WRs (shaded) spanning two cache lines. Arrows represent PCIe transactions. Red (thin) arrows are MMIO writes; the blue (thick) arrow are DMA reads. Arrows are marked with descriptor numbers; arrow width represents transaction size.

3.6.2 Reduce CPU-initiated MMIOs

MMIO writes are relatively expensive for the CPU because they require flushing the write buffers, and using memory barriers for ordering. Reducing MMIOs, or replacing them with more CPU- and bandwidth-efficient DMAs can improve both CPU efficiency and NIC throughput. CPUs initiate network operations by sending a message to the NIC via MMIO. The message can either contain the new work requests, or it can refer to the new descriptors by using information such as the address of the last descriptors. This works because queue descriptors are laid out contiguously in host memory.

In the first case, the CPU uses 64 B write-combined MMIOs to transfer the descriptors to the NIC. In the second case, the NIC reads the descriptors using one or more DMAs. We make the simplifying assumption that the NIC reads *all* new descriptors in one DMA, which we have found to work well in practice. In reality, NICs may read one or more descriptors per DMA, depending on the NIC's proprietary prefetching logic. We refer to these methods as *Descriptor-MMIO* and *Doorbell*, respectively. Different NIC technologies have different terms for these two methods. For example, Mellanox uses “BlueFlame” and “Doorbell,” and Intel Omni-Path Architecture uses “Programmed I/O send” and “SDMA,” respectively. Figure 3.4 summarizes these two methods. The Descriptor-MMIO method optimizes for low latency and is typically the default method in userspace NIC drivers.

The size of TX descriptors is vendor-specific, and it depends on several factors. For example, with RDMA NICs, the TX descriptor size depends on the transport. In Mellanox's implementation, as of 2019, descriptors for RC and UD transports' TX queues have 36 B and 68 B headers, respectively. UD transport's descriptor is larger because it includes network routing information. For RC transport, routing information for the connection is cached inside the NIC.

Terminology and default assumptions. To discuss the impact on CPU and PCIe use of the optimizations in this chapter, we consider transferring N descriptors of size D bytes from the CPU to the NIC. We denote the per-lane bandwidth, request header size, and completion header size of PCIe 3.0 by P_{bw}, P_r, and P_c, respectively. As our study focuses on small messages, descriptors include inlined payloads by default (Section 3.6.3.1).

Figure 3.5 summarizes two optimizations for reducing MMIOs—Doorbell batching and descriptor shrinking—introduced below.

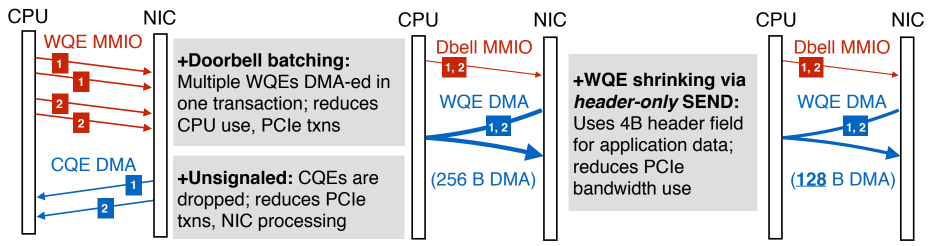

CPU
NIC
WQE MMIO
1
1
2
2
CQE DMA
1
2
+Doorbell batching: Multiple WQEs DMA-ed in one transaction; reduces CPU use, PCIe txns
+Unsignaled: CQEs are dropped; reduces PCIe txns, NIC processing
CPU
NIC
Dbell MMIO
1, 2
WQE DMA
1, 2
(256 B DMA)
+WQE shrinking via *header-only* SEND: Uses 4B header field for application data; reduces PCIe bandwidth use
CPU
NIC
Dbell MMIO
1, 2
WQE DMA
1, 2
(128 B DMA)

Figure 3.5: Optimizations for issuing two 4 B UD SENDs. A UD SEND descriptor spans two cache lines on Mellanox NICs because of the 68 B header; we shrink it to one cache line by using a 4 B header field for payload. Arrow notation follows Figure 3.4.

3.6.2.1 Doorbell batching

If an application can issue multiple descriptors to a TX queue, it can use one Doorbell MMIO for the batch of descriptors. Doorbell batching reduces CPU-generated MMIOs from $N * \lceil D/64 \rceil$ with per-descriptor MMIO to one. Because DMA reads are more bandwidth-efficient than MMIOs, doorbell batching also reduces PCIe bandwidth use. For example, with $N = 10$ and $D = 128$ and PCIe 3.0, descriptor transfer generates 1534 B of PCIe traffic, whereas Descriptor-MMIO generates 1800 B.[1]

Batching is relevant only to the Doorbell mode. In the Descriptor-MMIO mode, a batched descriptor transfer is identical to a sequence of individual descriptor transfers. Therefore, we use the more general but shorter term batching to refer to Doorbell batching. In this chapter, we use Descriptor-MMIO for non-batched operations, and Doorbell for batched operations. However, when batching is enabled but the available batch size is one, we use Descriptor-MMIO. This is because Doorbell provides little CPU savings for transferring a single small descriptor, and uses an extra PCIe transaction.

Effectiveness. Batching is highly effective for sending datagrams. Figure 3.6 shows the throughput and PCIe bandwidth limit of batched and non-batched UD SENDs on CIB. We use one server to issue SENDs to multiple client machines. When batching is enabled, we use batches of size 16 (i.e., the NIC DMA-reads 16 descriptors per Doorbell). Otherwise, the CPU writes the descriptors with the Descriptor-MMIO method. We use as many cores as required to maximize throughput. Batching improves peak SEND throughput by 27% from 80 million operations/s (Mops) to 101.6 Mops.

With batching, throughput is limited by DMA bandwidth: Every DMA completion of size C_{rc} bytes has a PCIe completion header of size P_c bytes (22 B, Table 3.1), leading to 13443 MB/s of useful DMA read bandwidth on CIB. Because UD SEND descriptors with non-empty payloads

[1]Let us denote the doorbell size (8 bytes) by d. The total data transmitted from CPU to NIC with the Descriptor-MMIO method is $T_{bf} = 10 * (\lceil 128/64 \rceil * (64 + P_r))$ bytes. Assuming $C_{rc} = 128$, the Doorbell method transmits $T_{db} = (d+P_r)+(10*(128+P_c))$ bytes. We ignore the PCIe link layer traffic since it is small compared to transaction-layer traffic: it is common to assume two link-layer packets (one flow control update and one acknowledgment, both 8 B) per 4-5 PCIe TLPs [150], making the link-layer overhead less than 5%. Substituting $d = 8$, $P_r = 26$, and $P_c = 22$ (Table 3.1) gives $T_{bf} = 1800$, and $T_{db} = 1534$.

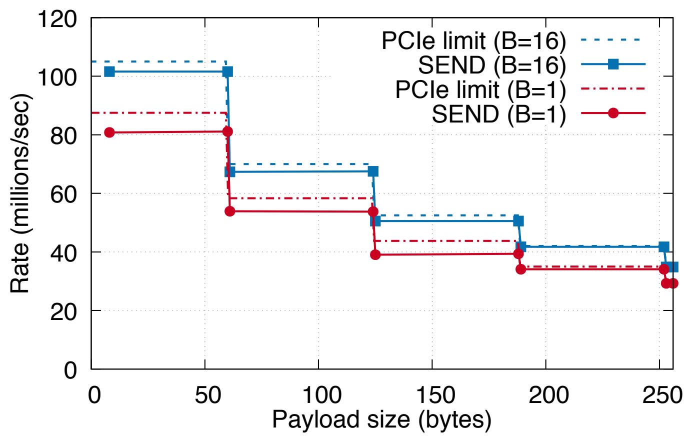


Figure 3.6: Peak batched and non-batched UD SEND throughput on CIB, with batch size *B*. Dotted lines show corresponding PCIe limits.

span at least two cache lines on CIB NICs, the maximum descriptor transfer rate is 13443/128 = 105 million/s, which is within 5% of our achieved throughput. We attribute the difference to link- and physical-layer PCIe overheads.

Non-batched throughput is limited by MMIO bandwidth. The write-combining MMIO rate on CIB is $(16 * P_{bw})/(64 + P_r) = 175$ million cache lines/s. Because our descriptors span at least 2 cache lines, the upper limit for throughput is 87.5 Mops. This is within 10% of our achieved 80 Mops.

Limitations. Batching is primarily useful for datagram transports only: all descriptors in a batch must use the same queue because Doorbells are per queue. This limitation seems fundamental to the parallel architecture of NICs: In a hypothetical NIC design where Doorbells contained information relevant for multiple queues (e.g., a compact encoding of "5 and 2 new descriptors for queue 1 and queue 2, respectively"), sending the Doorbell to the NIC processing units handling these queues would require an expensive selective broadcast inside the NIC. These PUs would then issue separate DMAs for descriptors, losing the coalescing advantage of batching. This limitation makes batching less useful for RDMA's connected QPs, which provide only one-to-one communication between two machines: the chances that a process has multiple messages for the same remote machine are low in large deployments. We therefore discuss batching for UD transport only.

3.6.2.2 Descriptor shrinking

Reducing the number of cache lines used by a TX queue descriptor can improve throughput drastically. For example, consider reducing TX descriptor size by only one byte from 129 B to 128 B. This can be done, for example, by shaving off one byte from the application payload. With the Descriptor-MMIO method, the number of CPU-generated MMIOs decreases from three to two. Descriptor shrinking mechanisms include compacting the inlined application payload, or overloading unused descriptor header fields with application data.

	RECV 0	RECV ≥ 1	SEND 0	SEND ≥ 1
CIB	122.0	82.0	157.0	101.6
CX3	34.0	21.8	32.1	26.0
CX	15.3	9.6	11.9	11.9

Table 3.3: Per-NIC rate (millions/s) for header-only (0) and regular (≥ 1) SENDs and RECVs

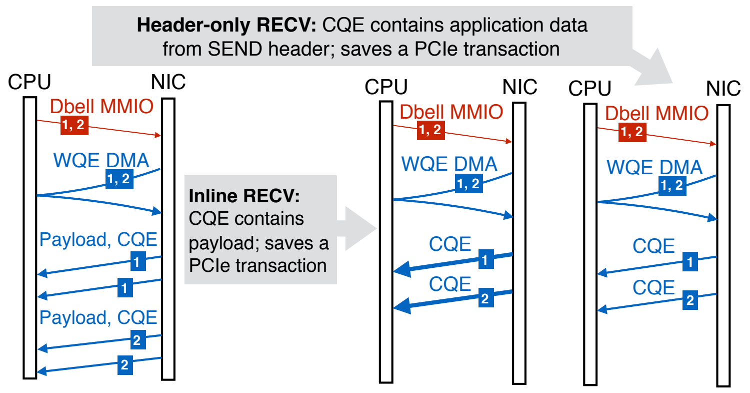

Header-only RECV: CQE contains application data from SEND header; saves a PCIe transaction
CPU
NIC
Dbell MMIO
1, 2
WQE DMA
1, 2
Payload, CQE
1
1
Payload, CQE
2
2
Inline RECV: CQE contains payload; saves a PCIe transaction
CQE
1
CQE
2

Figure 3.7: Optimizations for RECVs with small SENDs

Effectiveness. We use TX descriptor shrinking to implement an optimization called *header-only* SENDs. On Mellanox NICs, a UD SEND descriptor with a non-empty payload requires at least two cache lines. The descriptor's header has four bytes of space that can hold application data. Table 3.3 shows that header-only SENDs achieve up to 54% higher throughput than payload-carrying SENDs.

Applications that require messages larger than four bytes may use header-only SENDs in combination with *speculation*. In general, speculation for RPCs works as follows: clients transmit their expected response along with requests, and get a small confirmation response in the common case, using a header-only SEND. We demonstrate such a design for an eight-byte sequencer in Section 4.7.

Limitations. Shrinking application payloads is typically not straightforward or always possible. In cases when it is possible, doing so can increase application complexity and make the codebase more challenging to maintain. Header-only SENDs are useful only for simple applications (e.g., a sequencer) that can encode their application payload in a few bytes.

3.6.3 Reduce NIC-initiated DMAs

Reducing DMAs saves NIC processing power and PCIe bandwidth, improving throughput. The two optimizations in Section 3.6.2 affect DMAs, too: transferring a fixed number of descriptors with large batches requires fewer DMA reads than with smaller batches; descriptor shrinking further makes these DMAs smaller. Note that the Doorbell batching optimization above *adds* a DMA read, but it avoids *multiple* MMIOs, which is usually a good tradeoff.

3.6.3.1 Payload inlining

Inlining reduces latency, NIC processing, and PCIe bandwidth use by eliminating the DMA read for the payload. By default, TX queue descriptors (i.e., descriptors for packet transmission in non-RDMA NICs, as well as RDMA writes and SENDs) contain a pointer to the payload; the NIC fetches it via a DMA read. Modern NICs typically support encapsulating small payloads up to a few hundred bytes inside the descriptor. The CPU can then write the inlined descriptor to the NIC using Descriptor-MMIO, or the NIC can fetch it using DMA.

Effectiveness. We demonstrate the performance benefits of inlining in the context of RDMA in detail in Chapter 4. For example, the round-trip latency of an inlined WRITE is up to 400 ns lower than a non-inlined WRITE (Figure 4.1). For HERD's choice of RDMA primitives to implement RPCs, inlining improves RPC throughput by 35% for 32 B requests and responses (Figure 4.4).

Limitations. Inlining increases descriptor size, increasing the size (in bytes) of TX queues. This results in a higher NIC memory footprint per queue, reducing the number of TX queues the NIC can hold before suffering cache misses. On modern multi-CPU platforms, it is possible for over one hundred cores to share a NIC, requiring at least one hundred datagram queues for efficient network I/O without expensive sharing of queues among cores.

We have found that using smaller, non-inlined descriptors increases the number of cores an RPC library can support before suffering from NIC cache misses. eRPC supports changing the size of the inlined payload to allow users to make a tradeoff between low latency from enabling inlining, and higher number-of-cores scalability from disabling inlining.

3.6.3.2 Unsignaled transmission

A feature in modern NICs allows omitting TX completion queue entries, improving performance by reducing PCIe bandwidth use, and NIC and CPU processing. Host software may mark TX queue descriptors as *unsignaled* by setting a flag in the descriptor. The host's NIC does not DMA-write a CQE for these operations. An application that uses unsignaled transmission may detect TX completion using other, application-specific methods. For example, an RPC library at the client side may use the server's response as an implicit signal for the completion of request transmission, instead of relying on an explicit CQE from the client's NIC. In Chapter 6, we show how an RPC library can use unsignaled request transmission even in a lossy network that might drop the response.

Communication patterns that lack implicit ACKs, such as messaging and RPC chains (Section 3.5.1), can use a variant of unsignaled transmission, called selective signaling. With one TX queue, selective signaling works as follows: we post N unsignaled descriptors, followed by one signaled descriptor. The NIC processes descriptors in a TX queue in order, so the CQE for the signaled descriptor indicates transmission completion for the previous N descriptors. This technique cuts down CQE overhead by a factor of N. We typically use values of N between 32 and 128, making CQE overhead negligible.

Effectiveness. In Section 4.3.2.2, we show that using unsignaled completions improves RPC performance by 10–43%, measured with different combinations of RDMA primitives used to implement RPCs. All three RPC designs presented in this thesis use unsignaled completions.

Limitations. One potential drawback of unsignaled transmission is reduced packet pacing precision, leading to bursty traffic that might cause queue buildup in network switches. With signaled transmission, the client's RPC library has more control over the rate at which its NIC places packets on the wire: CQEs generated by the client's local NIC are a fine-grained and predictable mechanism for clocking packet transmission. Implicit completions generated by remote software are relatively coarse-grained, and less predictable than CQEs because an application-level request handler at the server governs response turnaround time. We do not explore this drawback of unsignaled completions in this thesis.

3.6.3.3 Header-only RECVs

Unlike TX queue operations, NICs must DMA a completion queue entry for completed RX queue operations; this provides an additional optimization opportunity, as discussed below. RX CQEs contain important metadata such as the size of received data, whereas TX CQEs only signal completion, and are therefore dispensible, NICs typically generate two separate DMA writes for payload and completion, writing them to application- and driver-owned memory respectively.

Assume that the corresponding SEND for a RECV carries an X-byte payload. If $X = 0$ (i.e., the SEND is header-only), the payload DMA is not generated at the receiver. Some information from the packet's header is included in the DMA-ed CQE, which can be used to implement application protocols.

Effectiveness. Header-only RECVs are useful in the applications of header-only SENDs, discussed earlier in Section 3.6.3.1. Table 3.3 shows that header-only RECVs achieve up to 62% higher throughput than payload-carrying RECVs. In all three clusters, header-only RECVs achieve similar throughput as inbound RDMA writes. This observation explains the rule of thumb that two-sided operations are slower than RDMA writes: RDMA writes generate only one DMA write at the responder, whereas RECVs normally generate two.

Limitations. The limitations of header-only RECVs are similar to header-only SENDs (Section 3.6.2.2).

3.7 Guidelines based on NIC architecture

We next present three guidelines based on three different aspects of the hardware architecture of modern high-speed NICs: parallelism, concurrency control, and caching. Although it is well-known that these aspects affect performance of computing devices, their effect on NIC performance has received little attention in the past.

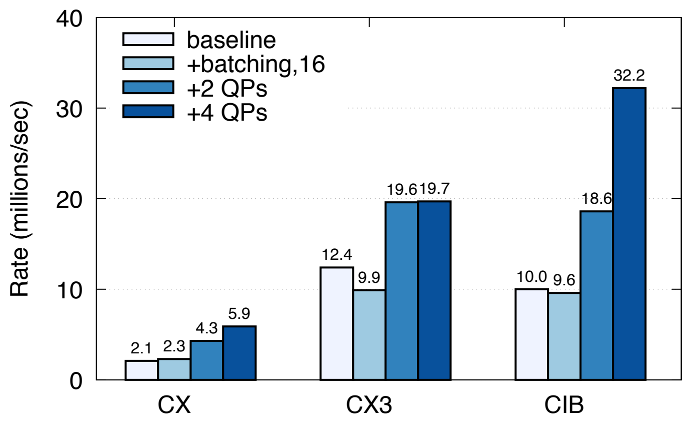


Figure 3.8: Effect of optimizations on single-core UD SEND throughput with a 60 B payload. Each descriptor spans two cache lines, so the NIC reads 128 B per descriptor.

3.7.1 Engage multiple NIC processing units

Exploiting NIC parallelism is necessary for high performance, but requires explicit attention. A common design decision in networking software is to use one NIC TX queue per CPU core, but doing so limits NIC parallelism to the number of queues. This is because operations on the same TX queue have ordering dependencies and are ideally handled by the same NIC processing unit to avoid synchronization across PUs.

For example, in datagram-based communication, one TX queue per CPU core is sufficient for communication with all remote hosts. Using one TX queue consumes the least NIC SRAM to hold queue state, while avoiding queue sharing among CPU cores. However, it “binds” a CPU core to a PU and may limit core throughput to PU throughput. This is likely to happen when per-message application processing is small, and a high-speed CPU core overwhelms a less powerful PU. In such cases, using multiple TX queues per core increases CPU efficiency; we call this the *multi-queue* optimization.

Effectiveness. Figure 3.8 shows single-core throughput for batched and non-batched 60 B UD SENDs—the largest payload size for which the TX descriptors fit in two cache lines. Interestingly, batching *decreases* core throughput if only one TX queue is used: with one queue, a core is coupled to a NIC processing unit, so throughput depends on how the PU handles batched and non-batched operations. Batching has the expected effect when we break this coupling by using multiple queues. Batched throughput increases by $\sim$ 2x on all clusters with two queues, and between 2–3.2x with four queues. Non-batched (i.e., Descriptor-MMIO) throughput does not increase with multiple queues (not shown in graph), showing that it is CPU-limited.

Designing distributed systems often requires choosing between CPU-bypassing and CPU-involving designs. Our results show that achieving peak message rate on even the most powerful NICs *does not* require a prohibitive amount of CPU power: just four cores saturate even PCIe 3.0 x16. Therefore, CPU-involving designs will not be limited by CPU processing power, provided that their application-level processing permits so.

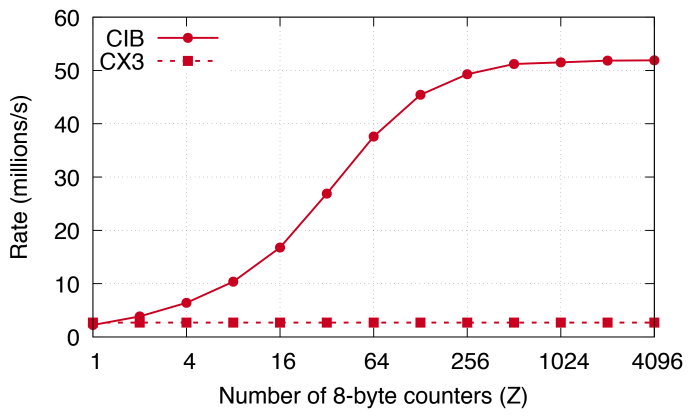


Figure 3.9: Atomics throughput with increasing concurrency

Limitations. The multi-queue optimization reduces number-of-cores scalability by increasing the NIC memory footprint per-core, similar to inlining (Section 3.6.3.1). Developers must therefore make a tradeoff between higher per-core efficiency from using multiple queues, or higher scalability from one queue per core.

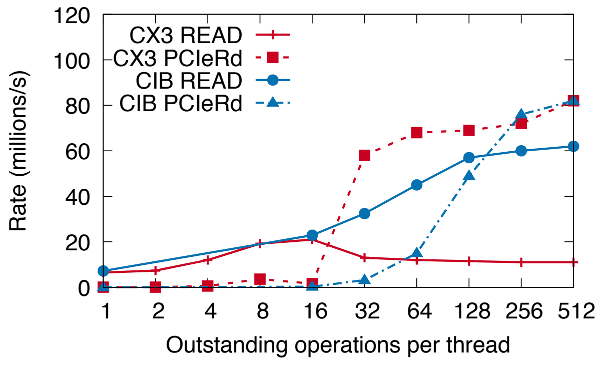
3.7.2 Avoid contention among NIC processing units

RDMA NICs provide atomic compare-and-swap and fetch-and-add operations on remote memory. These operations require cross-queue synchronization that introduces contention among PUs. Under some workload patterns, these operations can perform over an order of magnitude worse than uncontended operations.

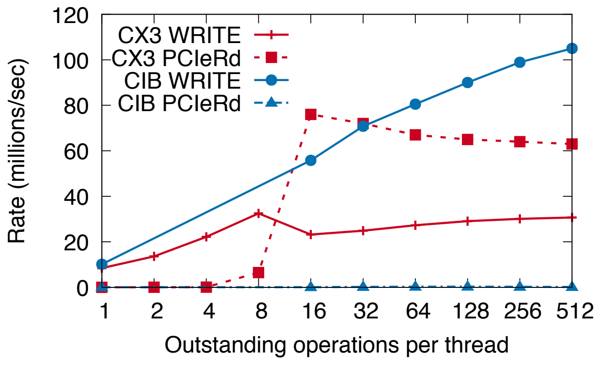
To our knowledge, all RDMA NICs available at the time of writing use internal concurrency control for atomics: PUs acquire a lock inside the NIC for the target address, and issue read-modify-write over PCIe. Note that atomic operations contend with non-atomic verbs too. In the future, NICs may use PCIe's atomic transactions for higher performing, cache coherence-based concurrency control. Therefore, the NIC's internal locking *mechanism*, such as the number of locks and the mapping of atomic addresses to these locks, is important. Due to the limited SRAM in NICs, the number of available locks is small, which amplifies contention in the workload.

Measurements. The performance of atomics depends on the amount of parallelism in the workload with respect to the NIC's internal locking scheme. To vary the amount of parallelism, we create an array of Z 8 B counters in a server's memory, and multiple remote client processes issue atomic operations on counters chosen randomly at each iteration. Figure 3.9 shows the total client throughput in this experiment. For CX3, it remains 2.7 Mops irrespective of Z; for CIB, it rises to 52 Mops.

Simple experiments allow reverse-engineering the NIC's locking mechanism. The flatness of CX3's throughput graph indicates that it serializes all atomic operations. Throughput on CX3 is limited by PCIe latency because of serialization. For CIB, we measured performance with randomly chosen pairs of addresses and observed lower performance for pairs where both



(a) Descriptor cache misses for READs



(b) Descriptor cache misses for WRITEs

Figure 3.10: RDMA throughput and corresponding descriptor cache misses for READs and WRITEs over RC transport, measured for different window sizes.

addresses have the same 12 least significant bits. This suggests that CIB uses 4096 buckets to slot atomic operations by address—a new operation waits until its slot is empty. For CIB, buffering and computation needed for PCIe read-modify-write makes NIC processing power the bottleneck.

The abysmal throughput for $Z = 1$ on both NICs shows that atomics are a poor choice for a sequencer; our optimized sequencer in Section 4.7 provides 12.2x higher performance with a *single* server CPU core. A lock service for data stores, however, might use a larger Z. Atomics could perform well if such an application used CIB, but they are very slow with CX3, which is the NIC used in recent atomics-based designs [143, 154]. With CIB, careful lock placement is still necessary. For example, if page-aligned data records have their lock variables at the same offset in the record, all lock requests will have the same 12 LSBs and will get serialized. A deterministic scheme that places the lock at different offsets in different records, or a scheme that keeps locks separate from the data will perform better.

3.7.3 Avoid NIC cache misses

NICs cache several types of information; it is critical to maintain a high cache hit rate because a miss translates to a read over PCIe. Cached information includes (1) virtual to physical address translations for RDMA-registered memory, (2) QP state, and (3) a descriptor cache. While the first two are known [38], the third is undocumented and was discovered in our experiments. Address translation cache misses can be reduced by using large (e.g., 2 MB) pages, and QP state cache misses by using fewer QPs [38].

Measurements. All types of NIC cache misses are transparent to the application and can be difficult to detect. We show that PCIe counters can be used to detect and measure NIC cache misses. In general, subtracting the application's expected PCIe reads from the actual reads reported by PCIe counters gives an estimate of cache misses. Estimating expected PCIe reads in turn requires PCIe models of RDMA operations.

Achieving high outbound throughput requires maintaining multiple outstanding transmissions via pipelining. On receiving a new descriptor from the CPU, the NIC saves the descriptor in its descriptor cache. However, if the CPU injects new descriptors faster than the NIC's processing speed, this descriptor can be evicted by newer descriptors. When the NIC eventually processes the evicted descriptor, a cache miss occurs.

Figure 3.10 quantifies this phenomenon on CIB for RDMA reads and writes over RC transport. For these operations, the requester's NIC can suffer a cache miss when generating the RDMA request packet, or while servicing the RDMA response. We conduct the following experiment: 14 requester threads on a server issue windows of N 8 B READs or WRITEs over RC transport to 14 remote processes. In Figures 3.10a and 3.10b, we show the cumulative RDMA request rate, and the extent of descriptor cache misses using the `PCIeRdCur` counter rate. Each thread waits for the N requests to complete before issuing the next window. We use all 14 cores on the server to generate the maximum possible request rate, and RC transport to include cache misses generated while processing ACKs for WRITEs. We make the following observations, showing the importance of the descriptor cache in improving and understanding communication throughput:

- The optimal window size for maximum throughput is not obvious: throughput does not always increase with increasing window size, and is dependent on the NIC. For example, $N = 16$ and $N = 512$ maximize READ throughput on CX3 and CIB respectively.
- Higher throughput may be obtained at the cost of PCIe reads. For example, on CIB, both READ throughput and PCIe read rate increases as N increases. Although the largest N is optimal for a machine that only issues outbound READs, it may be suboptimal if it also serves other operations.
- CIB's NIC can handle the CPU's peak descriptor injection rate for WRITEs and never suffers cache misses. This is not true for READs, suggesting that READs require more NIC processing than WRITEs.

3.8 Related work

Although there is a large body of work that measures the performance of network communication at a high level [50, 56, 137], there is little prior literature on understanding the low-level behavior of network cards. Flajslik and Rosenblum [48] present a high-level picture of basic PCIe interactions between Ethernet NICs and CPUs, but their work does not cover more sophisticated interactions, such as those that arise during batched transfers. Larsen and Lee [87] study the PCIe behavior of Intel 82599 Ethernet cards using a PCIe protocol analyzer, and divide the PCIe traffic into Doorbells, packet descriptors, and actual payloads. We contribute a method of reasoning about PCIe traffic that does not depend on an expensive PCIe protocol analyser, and works generally for PCIe-based NICs. In addition, while Larsen and Lee [87] propose to eliminate PCIe with a new integrated DMA controller architecture, we present optimizations and guidelines of improving PCIe use in the confines of existing platform architectures.

Several of our guidelines apply to RDMA-based distributed systems that use advanced NIC primitives. A key design decision in such systems is the choice of NIC primitives, made using a microbenchmark-based performance comparison [38, 113, 154, 155, 159]. Our work shows that there are more dimensions to such comparisons than these projects explore. The new dimensions include user-controlled aspects such as batching, inlining, and selective signaling, as well as in-NIC aspects that are transparent to the user, such as caching, parallelism, and concurrency control. A common theme in later chapters of this dissertation is that comparing two choices of NIC primitives without exhaustively exploring the space of low-level factors and optimizations often leads to a sub-optimal choice. We show that our choice of NIC primitives, made based on the guidelines in this chapter, outperforms prior state-of-the-art choices made using microbenchmarks that missed some of the low-level factors.

3.9 Conclusion

Designing high-performance distributed systems requires a deep understanding of low-level hardware details such as PCIe behavior and NIC architecture. We believe that by presenting clear guidelines, significant optimizations based on these guidelines, and tools and experiments for low-level measurements on their NIC hardware, our work will help researchers and developers to better use NICs in their high-performance systems.

Over the next three chapters, we will discuss how major design decisions in HERD, FaSST, and eRPC tie back to these guidelines. At these decision points, we will use the Guideline label to refer back to the insights presented in this chapter.

Chapter 4

Case study 1: HERD – An RPC-based key-value store

This chapter is the first of our two case studies demonstrating the benefits of RPC-based designs over those that use in-network capabilities. We present an RPC-based key-value store called HERD, which is designed to make the best use of an RDMA-capable network. The key insight in HERD is that high-performance RPCs—implemented using RDMA's low-level primitives—outperform RDMA's core CPU-bypassing features for key-value access.

Unlike prior RDMA-based key-value systems, we optimize HERD's design to minimize network round trips instead of using one-sided RDMA in an attempt to save CPU cycles at the key-value server. The result is substantially lower latency, and throughput that saturates modern, commodity RDMA hardware. HERD has two unconventional decisions: First, it does not use RDMA reads, despite the allure of operations that bypass the remote CPU entirely. Second, it uses a mix of RDMA's one-sided and two-sided verbs, despite the conventional wisdom that the two-sided primitives are slow. A HERD client writes its request into the server's memory; the server's CPU computes and sends the reply. This design uses a single round trip for all requests. Notably, for small key-value items, our full system throughput is similar to native RDMA read throughput and is over 2x higher than recent RDMA-based key-value systems. HERD further serves as an effective template for the construction of other communication-intensive distributed systems.

4.1 Introduction

This chapter explores a question that has important implications for the design of modern communication-intensive distributed systems: What is the best method for using RDMA features to support remote key-value store access? To answer this question, we first evaluate the performance that, with sufficient attention to engineering, can be achieved by each of the RDMA communication primitives. Using this understanding, we show how to use an unexpected combination of methods and system architectures to achieve the best throughput and latency possible on a high-performance RDMA network.

Our work is motivated by the seeming contrast between the fundamental time requirements for cross-node traffic vs. CPU-to-memory lookups, and the designs that have recently emerged that use multiple RDMA reads. As discussed previously in Section 1.2, on the one hand, going between nodes takes roughly 2–10 µs, compared to 60–120 ns for a memory lookup, suggesting that a multiple-RTT design as found in the recent Pilaf [113] and FaRM [38] systems should be fundamentally slower than a single-RTT design. On the other hand, an RDMA read bypasses many potential sources of overhead, such as servicing interrupts and initiating control transfers, which involve the host CPU. In this chapter, we show that there is a better path to taking advantage of RDMA to achieve high-throughput, low-latency key-value storage.

A challenge for both our and prior work lies in the lack of richness of RDMA operations. An RDMA operation can only read or write a remote memory location. It is not possible to do more sophisticated operations such as dereferencing and following a pointer in remote memory. Recent work in building key-value stores [38, 113] has focused exclusively on using RDMA reads to traverse remote data structures, similar to what would have been done had the structure been in local memory. This approach invariably requires multiple round trips across the network.

Consider an ideal RDMA read–based key-value store (or cache) where each GET request requires only one small RDMA read. Designing such a store is as hard as designing a hashtable in which each GET request requires only one random memory lookup. We instead provide a solution to a simpler problem: we design a key-value cache that provides performance similar to that of the ideal cache. However, our design does not use RDMA *reads* at all.

HERD is a key-value cache that leverages RDMA features to deliver low latency and high throughput. As we demonstrate later, RDMA reads cannot harness the full performance of the RDMA hardware. In HERD, clients transmit their request to the server's memory using RDMA writes. The server's CPU polls its memory for incoming requests. On receiving a new request, it executes the GET or PUT operation in its local data structures and sends the response back to the client. As RDMA write performance does not scale with the number of outbound connections, a HERD server sends its response with a SEND message over a datagram transport.

We make three main contributions in this chapter:

- We present a thorough analysis of the performance of RDMA verbs, and expose the various design options for key-value systems.
- We provide evidence that RPCs are better than one-sided RDMA reads for key-value systems, refuting the previously held assumption [38, 113].
- We describe the design and implementation of HERD, a key-value cache that achieves the best possible throughput and latency on RDMA hardware.

The following section provides additional background on key-value stores, and describes recent efforts in building key-value stores using one-sided RDMA. Section 4.3 discusses the rationale behind our design decisions and demonstrates that messaging verbs are a better choice than RDMA reads for key-value systems. Section 4.4 discusses the design and implementation of the HERD key-value cache. In Section 4.5, we evaluate our system on a cluster of 187 nodes and compare it against FaRM [38] and Pilaf [113].

4.2 Background

4.2.1 Recent research on key-value stores

Section 2.2.1 discussed the high-level architecture and widespread use of key-value stores and caches in Internet services. This chapter focuses on the communication architecture to support both primary store and cache applications; we use a cache implementation for end-to-end validation of our resulting design.

Although recent in-memory object stores have used both tree and hash table-based designs, we focus on hash tables as the basic indexing data structure. Hash table design has a long and rich history, and the particular flavor one chooses depends largely on the desired optimization goals. In recent years, several systems have used advanced hash table designs such as cuckoo hashing [44, 96, 126] and hopscotch hashing [60]. Cuckoo hash tables are an attractive choice for building fast key-value systems [44, 96, 163] because, with K hash functions (usually, K is 2 or 3), they require only K memory lookups for `GET` operations, plus an additional pointer dereference if the values are not stored in the table itself. In many workloads, `GET`s constitute over 95% of the operations [9, 119]. This property makes cuckoo hashing an attractive backend for high-performance networked key-value stores [113, 163]. Cuckoo and hopscotch-based designs often emphasize workloads that are read-intensive: `PUT` operations require moving values within the tables. We evaluate both write-intensive (95% `PUT`) and read-intensive (95% `GET`) workloads in this chapter.

To support both types of workloads without being limited by the performance of currently available data structure options, HERD internally uses a *cache* data structure that can evict items when it is full. Our focus, however, is on the network communication architecture—our results generalize across both caches and stores, so long as the data structure implementation is fast enough that a high-performance communication architecture is needed. HERD's cache design is based on Lim et al. [97]'s system that provides both cache and store semantics. MICA's cache mode uses a lossy associative index to map keys to pointers, and stores the values in a circular log that is memory efficient, avoids fragmentation, and does not require expensive garbage collection. This design requires only two random memory accesses for both `GET` and `PUT` operations.

4.2.2 One-sided RDMA–based key-value stores

Pilaf [113] is a key-value store that aims for high performance and low CPU use. For `GET`s, clients access a cuckoo hash table at the server using READs, which requires 2.6 round trips on average for one GET request. For PUTs, clients send their requests to the server using a SEND message over RDMA's Reliable Connected (RC) transport. To ensure consistent GETs in the presence of concurrent PUTs, Pilaf's data structures are *self-verifying*: each hash table entry is augmented with two 64-bit checksums.

The second key-value store we compare against is based upon the distributed hash table designed in FaRM [38]. It is important to note that FaRM is a more general-purpose distributed transactional computing platform that exposes memory of a cluster of machines as a shared address space; we compare *only* against a key-value store implemented on top of FaRM that we call FaRM-KV. Unlike the client-server design in Pilaf and HERD, FaRM is symmetric, befitting its design as a cluster architecture: each machine acts as both a server and client.

FaRM's design provides two components for comparison. First is its key-value store design, which uses a variant of hopscotch hashing [60] to create a locality-aware hash table. For GETs, clients READ several consecutive hopscotch slots, one of which contains the key with high probability. Clients use another READ to fetch the value if it is not stored inside the hash table's buckets. For small, fixed-size key-value pairs, FaRM can "inline" the value with the key. For PUTs, clients WRITE their request to a circular buffer in the server's memory. The server polls this buffer to detect new requests. This design is not specific to FaRM—we use it merely as an extant alternative to Pilaf's cuckoo-based design to provide a more in-depth comparison for HERD.

The second important aspect of FaRM is its symmetry; here it differs from HERD, which is designed for asymmetric settings with many clients and few servers. Interestingly, although FaRM-KV uses fewer CPU cycle at the server than HERD, it uses more CPU cycles at clients. Therefore, FaRM-KV offers an attractive design point in use cases where server CPU cycles are more valuable than client cycles. In the next chapter, we show that in symmetric settings, the overall CPU consumption—summed over both clients and servers—of our RPC-based designs is lower than designs that use one-sided RDMA (Section 5.3.5).

4.3 Design decisions

Towards our goal of supporting key-value servers that achieve the highest possible throughput on RDMA-capable networks, we explain in this section the reasons we choose to use—and not use—particular RDMA features and other design options. HERD uses a hybrid of RDMA NIC primitives to implement RPCs, using both one-sided and two-sided verbs, as well as both connection-oriented and datagram transports, to best effect. Clients write their requests to the server using RDMA writes over an Unreliable Connection (UC). This write places the PUT or GET request into a per-client memory region in the server. The server polls these regions for new requests. Upon receiving one, the server process executes in conventional fashion using its local data structures. It then sends a reply to the client using messaging verbs: a SEND over an Unreliable Datagram (UD).

To explain why we use this hybrid of one-sided RDMA and messaging, we describe the performance experiments and analysis that support it. Particularly, we describe our construction of an RPC primitive that performs as well as single RDMA reads, allowing HERD to outperform prior key-value stores that use multiple RDMA reads for GETs by around 2x. We also describe how HERD's RPC primitive uses two-sided verbs despite conventional wisdom that they are slower than one-sided RDMA.

4.3.1 Notation and experimental setup

In this section, we present microbenchmarks from the CX3 cluster (Table 2.2), a large testbed equipped with 56 Gbps InfiniBand. These experiments use one server machine and several client machines. We denote the server machine by M_S and its NIC by NIC_S. We denote client machine i by C_i. The server and client machines may run multiple server and client processes, respectively. For throughput experiments, clients maintain a window of several outstanding verbs in their send queues. Using windows allows us to saturate our NICs with fewer processes. In all of our throughput experiments, we manually tune the window size for maximum aggregate throughput.

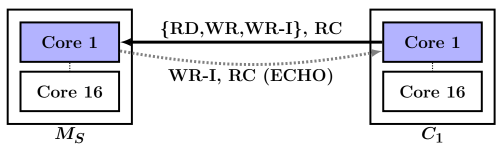
4.3.2 Constructing a fast RPC primitive

We begin by constructing an RPC primitive using two RDMA writes—one for the request and one for the response—that performs similarly to one RDMA read. In the next section, we describe how we improve the scalability of this basic primitive by using UD SENDs for responses, while preserving performance.

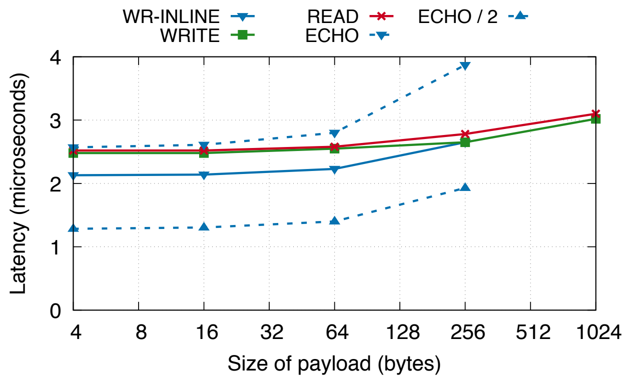
The RPC primitive in this section uses WRITEs over the UC transport, which makes it possible to replace one READ by two WRITEs without degrading latency or throughput. This is because UC WRITEs have lower latency and higher throughput than READs. First, as one might expect, the latency of an unsignaled WRITE is about half that ($\frac{1}{2}$ RTT) of a READ. Second, because the UC WRITE's responder does not need to send ACKs back, its NIC performs less processing, and thus can support higher throughput than with READs. The reduced network bandwidth use similarly benefits both the server and client throughput. We discuss these advantages in detail next.

4.3.2.1 WRITEs have lower latency than READs

Measuring the latency of an unsignaled WRITE is not straightforward because the requester gets no indication of completion. Therefore, we measure it indirectly by measuring the latency of an "ECHO" RPC. In an ECHO, a client sends a request to a server and the server relays the same message back to the client. The latency of an unsignaled WRITE is at most half of the latency of an ECHO implemented with unsignaled WRITEs.



(a) Setup for measuring verbs and ECHO latency. We use one client process to issue operations to one server process



(b) The one-way latency of WRITE is half that of ECHO. ECHO operations used unsignaled verbs.

Figure 4.1: Latency of verbs, and ECHO RPCs implemented using UC WRITEs

We also measure the latency of signaled READ and WRITE operations. As these operations are signaled, we use the completion entry to measure latency. For WRITE, we also report the latency with payload inlining (Guideline 3.6.3.1). Figure 4.1 shows the average latency from these measurements. We use inlined and unsignaled WRITEs for ECHOs. On our ConnectX-3 NICs, the maximum size of the inlined payload is 256 B. Therefore, we show graphs for WR-INLINE and ECHO only up to 256 B.

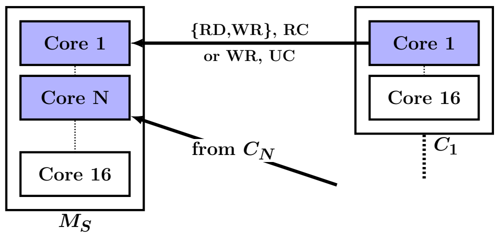
Unsignaled verbs. For payloads up to 64 B, the latency of ECHOs is close to READ latency, which confirms that the one-way WRITE latency is about half of the READ latency. For larger ECHOs, the latency increases because of the time spent in writing WRITE descriptors to the NIC via MMIO.

Signaled verbs. The solid lines in Figure 4.1 show the latencies for three signaled verbs—WRITEs, READs, and WRITEs with inlining (WR-INLINE). READ and WRITE latencies are similar because they travel identical network/PCIe paths. By avoiding one DMA operation, inlining reduces the latency of small WRITEs by around 400 ns.

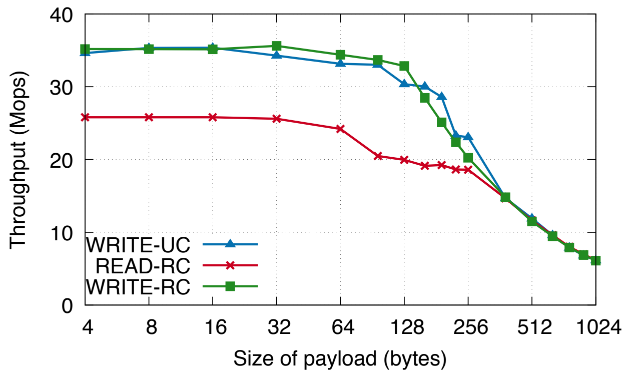
4.3.2.2 WRITEs have higher throughput than READs

To evaluate throughput, it is first necessary to observe that with many client machines communicating with one server, different verbs perform very differently when used at the clients (talking to one server) and at the server (talking to many clients).

Inbound throughput. We first measure the throughput of *inbound* verbs, i.e., the rate of verbs that multiple remote machines (the clients) can issue to one machine (the server). Using the notation introduced above, $C_1, \ldots, C_N$ issue operations to M_S, as shown in Figure 4.2a. Figure 4.2b shows the cumulative throughput observed across the active machines. For up to 128 B payloads, WRITEs achieve 35 Mops, which is 34% higher than the maximum READ throughput

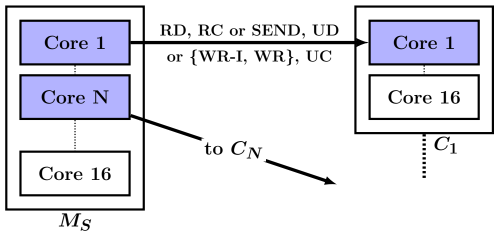


(a) Setup for measuring inbound throughput. Each client process communicates with only one server process

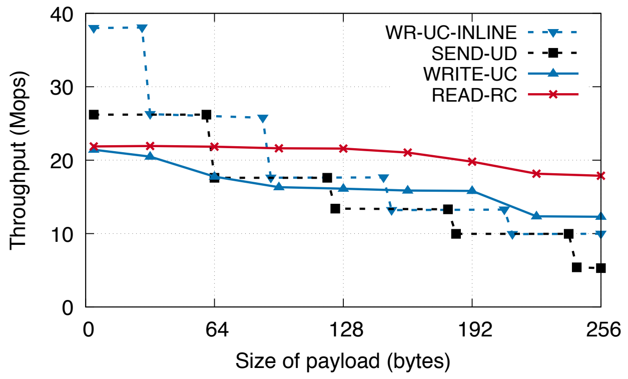


(b) For moderately-sized payloads, WRITE has much higher inbound throughput than READs.

Figure 4.2: Comparison of inbound verbs throughput



(a) Setup for measuring outbound throughput. Each server process communicates with only one client process.



(b) For small payloads, WRITE with inlining has higher outbound throughput than READ.

Figure 4.3: Comparison of outbound verbs throughput

(26 Mops). Interestingly, reliable WRITEs deliver significantly higher throughput than READs despite their identical InfiniBand path. This is explained as follows: writes require less state maintenance both at the RDMA and the PCIe level because the requester does not need to wait for a response. For reads, however, the requester must hold on to the request until a response arrives. For example, at the RDMA level, each queue pair can only service a few outstanding READ requests (16 in our RDMA NICs). Similarly, at the PCIe level, reads are performed using non-posted transactions, whereas writes use cheaper, posted transactions.

Although the inbound throughput of WRITEs over UC and RC is nearly identical, using UC is still beneficial: It requires less processing at NIC_S, and HERD uses this saved capacity for responses.

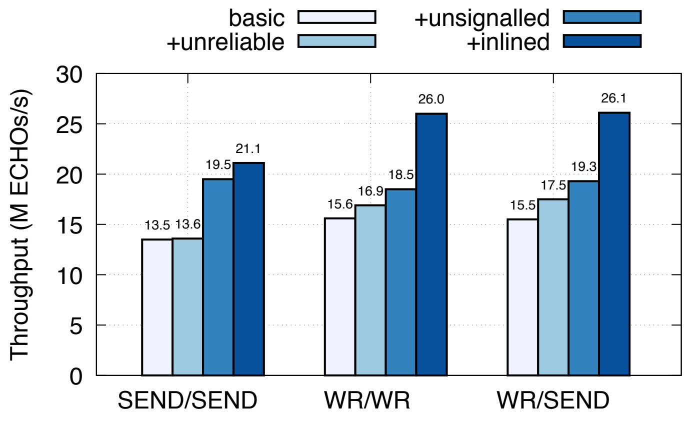


Figure 4.4: Throughput of ECHOs with 32 B messages. WR-SEND uses UD transport for responses.

Outbound throughput. We next measure the throughput for *outbound* verbs. Here, M_S issues operations to $C_1, ..., C_N$. As shown in Figure 4.3a, there are N processes on M_S; the i^{th} process communicates with C_i only (we explain the scalability problems associated with all-to-all communication later in Section 4.3.3). Apart from READs, WRITEs, and inlined WRITEs over UC, we also measure the throughput for inlined SENDs over UD, for reasons described in Section 4.3.3. Figure 4.3b plots the throughput achieved by M_S for different payload sizes. For small sizes, inlined WRITEs and SENDs have significantly higher outbound throughput than READs. For large sizes, the throughput of all WRITE and SEND variants is less than READs, but it is never less than 50% of the READ throughput. Thus, even for these larger items, using a single WRITE (or SEND) for responses remains a better choice than using multiple READs for key-value items.

ECHO throughput is interesting for two reasons. First, it provides an upper bound on the throughput of a key-value cache based on one round trip of communication. Second, ECHOs help characterize the processing power of the RDMA NIC: although the advertised message rate of the ConnectX-3 cards used in our evaluation of HERD is 35 Mops, bidirectionally, they can process many more messages.

Varying the verbs and transport types used for the request and the response message yields several different implementations of RPCs. Figure 4.4 shows the throughput for some of the possible combinations and for ECHO RPCs with 32 B payloads. The figure also shows that using unreliable transports, payload inlining, and unsignaled transmission increases ECHO throughput significantly.

ECHOs achieve maximum throughput (26 Mops) when both the request and the response are sent using RDMA writes. However, as we show in Section 4.3.3, this approach does not scale with the number of connections. HERD uses RDMA writes over UC for requests and SENDs (over UD) for responses. *An ECHO server using this hybrid also achieves 26 Mops—it gives the performance of WRITE-based ECHOs, but with much better scalability.*

By avoiding the overhead of posting RECVs at the server, our method of WRITE-based requests and SEND-based responses provides better throughput than purely SEND-based ECHO

RPCs. Interestingly, however, after enabling all optimizations, the throughput of purely SEND-based ECHOs (with no RDMA operations) is 21 Mops, which is more than three-fourths of the peak inbound READ throughput (26 Mops). Both Pilaf and FaRM have noted that RDMA reads vastly outperform SEND-based ECHOs, which our results agree with *if our optimizations are removed*. With these optimizations, however, SENDs significantly outperform READs in cases where a single SEND-based RPC can be used in place of multiple READs per request.

Our experiments show that several ECHO designs, with varying degrees of scalability, can perform better than multiple-READ designs. From a network-centric perspective, this is fortunate: it also means that designs that use only one cross-datacenter RTT can potentially outperform multiple-RTT designs both in throughput and in latency.

Discussion of verbs throughput. Mellanox advertises that the ConnectX-3 NICs in our evaluation support 35 million messages per second. Our experiments show that the card can achieve this rate for inbound WRITEs (Figure 4.2b) and slightly exceed it for very small outbound WRITEs (Figure 4.3b). All other verbs are slower than 30 Mops regardless of operation size. While the manufacturer does not specify bidirectional message throughput, we know empirically that NIC_S can service 30 million ECHOs per second (WRITE-based ECHOs achieve 30 Mops with 16 B payloads; Figure 4.4 uses 32 B payloads), or at least 60 total Mops of inbound WRITEs and outbound SENDs.

We attribute the reduced throughputs to several factors:

- For outbound WRITEs larger than 28 B, the NIC's message rate is limited by the PCIe MMIO throughput. The sharp decreases in the WR-UC-INLINE and SEND-UD graphs in Figure 4.3b at 64 B intervals are explained by the use of write-combining buffers for MMIO. Due to the larger UD transport descriptors, the throughput for SEND-UD drops for smaller payload sizes than for WRITEs

- The maximum throughput for inbound and outbound READs is 26 Mops and 22 Mops respectively, which is considerably smaller than the advertised 35 Mops message rate. Unlike WRITEs, READs are bottlenecked by the NIC's processing power. This is as expected. Outbound READs involve a MMIO operation, a packet transmission, a packet reception, and a DMA write, whereas outbound WRITEs (inlined and over UC) avoid the last two steps. Inbound READs require a DMA read by the NIC followed by the response packet's transmission, whereas inbound WRITEs require only a DMA write.

4.3.3 Using datagram transport for responses

Our previous experiments did not show that as the number of connections increases, connected transports begin to slow down due to NIC cache misses (Guideline 3.5.2). This is a potentially important effect to avoid both for cluster scaling, but also because it interacts with the cache or store architectural decisions. For example, the cache design we build on in HERD partitions the key space between several server processes in order to achieve efficient CPU and memory utilization. Such partitioning further increases the fan-in and fan out of connections to a single machine.

To evaluate this effect, we modified our throughput experiments to enable *all-to-all* communication. We use N client processes (one process each at $C_1, ..., C_N$) and N server processes at M_S. For measuring inbound throughput, client processes select a server process at random and issue a WRITE to it. For outbound throughput, a server process selects a client at random and issues a WRITE to it. Figure 4.5 shows the results of these experiments for 32 B messages. Several results stand out:

Outbound WRITEs scale poorly. for $N = 16$, there are 256 active queue pairs at NIC_S and the server-to-clients throughput degrades to 21% of the maximum outbound WRITE throughput (Figure 4.3b). With many active queue pairs, each posted verb can cause a cache miss, severely degrading performance.

Inbound WRITEs scale well. Clients-to-server throughput is high even for $N = 16$. This is because RDMA NICs maintain more connections state at requesters than at responders. Therefore, the server's NIC can support a much larger number of active queue pairs (QPs) without incurring cache misses. The large number of clients amortizes the requester overhead in this experiment.

In a different experiment, we used 1600 client processes spread over 16 machines to issue WRITEs over UC to one server process. HERD uses this many-to-one configuration to reduce the number of active connections at the server (Section 4.4.3). This configuration also achieves 30 Mops.

Outbound WRITEs scale poorly only because NIC_S must manage many connected queue pairs. This problem cannot be solved if we use connected transports because they require at least as many queue pairs at M_S as the number of client machines. Scaling outbound communication therefore mandates using datagrams. UD transport supports one-to-many communication, i.e., a single UD queue can be used to issue operations to multiple remote UD queues. The main challenge with using UD in a high performance application is that it only supports two-sided verbs and not one-sided verbs.

Fortunately, the main overhead of two-sided verbs—posting RECVs and handling RECV completions—exists at only the receiver. Senders can directly transmit their requests, similar to issuing RDMA writes. Figure 4.5 shows that, when performed over UD, SEND-side throughput is high and scales well with the number of connected clients.

The slight degradation of SEND throughput beyond 10 connected clients happens because the SENDs are unsignaled, i.e., server processes get no indication of verb completion. This leads to the server processes overwhelming NIC_S with too many outstanding operations, causing cache misses inside the NIC. As HERD uses SENDs for responding to requests, it can use new requests as an indication of the completion of old SENDs, thereby avoiding this problem.

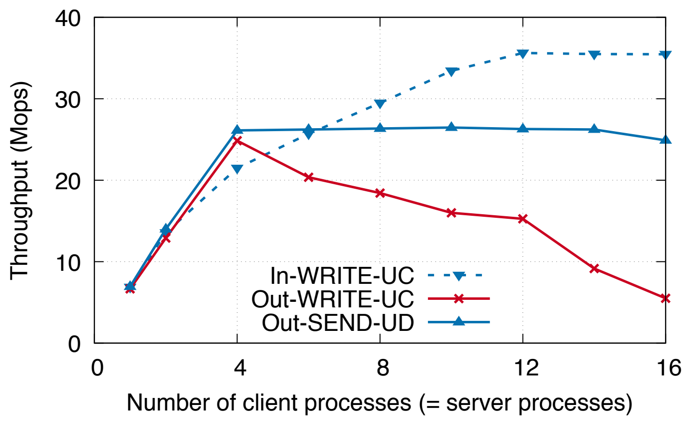


Figure 4.5: Comparison of UD and UC for all-to-all communication with 32 B payloads. Inbound WRITEs over UC and outbound SENDs over UD scale well up to 256 queue pairs. Outbound WRITEs over UC scale poorly. All operations are inlined and unsignaled.

4.4 Design of HERD

Our HERD setup consists of one server machine and several client machines. The server machine runs N_S server processes. We run N_C client processes, uniformly spread across the client machines.

4.4.1 HERD's key-value data structure

HERD borrows its key-value caching data structures from MICA [97], a high-speed key-value store and cache designed for classical Ethernet. We restrict our discussion of MICA to its cache mode. MICA uses a lossy index to map keys to pointers, and stores the actual values in a circular log. Insertion can cause item eviction from the index (thereby making the index lossy), or from the log in a FIFO order. We use MICA's algorithm for both GETs and PUTs: each GET requires up to two random memory lookups, and each PUT requires one.

MICA shards the key space into several partitions based on the key's hash. In its Exclusive Read Exclusive Write (EREW) mode, which we use, each server core has exclusive read and write access to one partition. MICA uses the Flow Director [65] feature of modern classical Ethernet NICs to direct request packets to the core responsible for the given key. HERD achieves the same effect by allocating per-core request memory at the server, and allowing clients to WRITE their requests directly to the appropriate core.

4.4.2 Masking DRAM latency with prefetching

To service a GET, a HERD server must perform two random memory lookups, prepare the SEND response (with the key's value inlined in the SEND descriptor), and then post the SEND verb

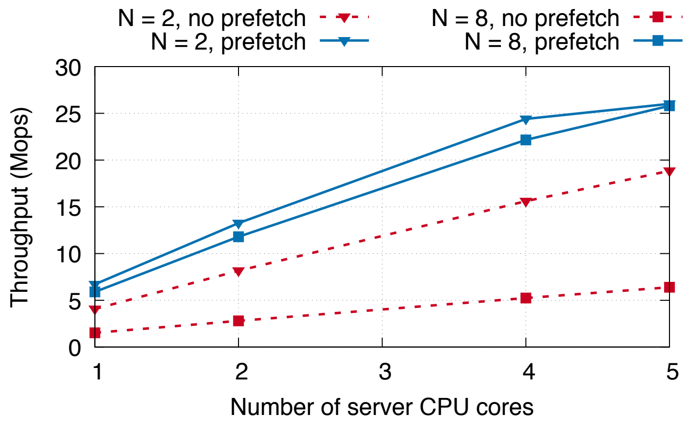


Figure 4.6: Effect of memory prefetching on throughput

using the `post_send()` verbs function. The memory lookups and the `post_send()` function are the two main sources of latency at the server. Each random memory access takes 60–120 ns, and the `post_send()` function takes about 150 *ns*. While the latter is unavoidable, we can mask the memory access latency by overlapping memory accesses of one request with computation of another request.

MICA and CuckooSwitch [163] mask latency by overlapping memory fetches and prefetches, or request decoding and prefetches. HERD takes a different approach: we overlap prefetches with the `post_send()` function used to transmit replies. We use the following approach: In HERD, the maximum number of memory lookups for each request is two. We create a request pipeline in software with two stages. When a request is in stage i of the pipeline, it performs the i-th memory access for the request and issues a prefetch for the next memory address. In this way, requests only access memory for which a prefetch has already been issued. On detecting a new request, the server issues a prefetch for the request's index bucket, advances the old requests in the pipeline, pushes in the new request, and finally calls `post_send()` to SEND a reply for the pipeline's completed request. The server process expects the issued prefetches to finish by the time `post_send()` returns.

Figure 4.6 shows the effectiveness of prefetching. We use a WRITE/SEND-based ECHO server, but this time the server performs N random memory accesses before sending the response. Prefetching allows fewer cores to deliver higher throughput: five cores are sufficient to achieve the peak throughput with even $N = 8$. We conclude that there is significant headroom to implement more complex key-value applications, for instance, key-value stores, on top of HERD's request-reply communication mechanism.

With a large number of server processes, this pipelining scheme can lead to a deadlock. A server does not advance its pipeline until it receives a new request, and a client does not advance its request window until it gets a response. A HERD server process avoids this deadlock by pushing a no-op into the pipeline if it fails to get a new requests for 100 consecutive iterations.

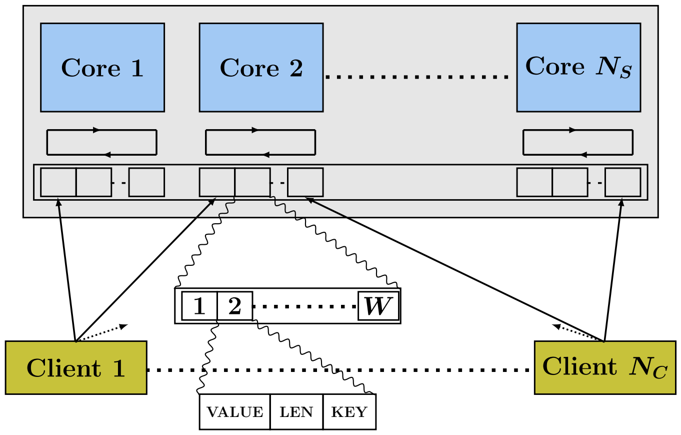


Figure 4.7: Layout of the request region at the server

4.4.3 Request format and handling

Clients WRITE their `GET` and `PUT` requests to a contiguous memory region on the server machine, which is allocated during initialization. This memory region, called the *request region*, is shared among all the server processes by mapping it using `shmget()`. The request region logically consists of 1 kB slots, which is the maximum size of a key-value item in HERD.

HERD's request format is as follows. A `GET` request consists of only a 16 B keyhash. A `PUT` request contains a 16 B keyhash, a 2 B length field specifying the value's length, and up to 1000 B for the value. To poll for incoming requests, we use the left-to-right ordering of PCIe DMA writes [38, 100]. We use the keyhash field to poll for new requests; therefore, the keyhash is written to the rightmost bytes of the slot. A non-zero keyhash indicates a new request, and we do not allow clients to use a zero hash. The server zeroes out the keyhash field of the slot after sending a response, freeing it up for a new request.

Figure 4.7 shows the layout of the request region at the server machine. It consists of separate chunks for each server process, which are further sub-divided into per-client chunks. Each per-client chunk consists of W slots, i.e., each client can have up to W pending requests to each server process. The size of the request region is $N_S \cdot N_C \cdot W$ KB. With $N_C = 200$, $N_S = 16$ and $W = 2$, this is approximately 6 MB and fits inside the server's L3 cache. Each server process polls the per-client chunks for new requests in a round robin fashion. If server process s has seen r requests from client number c, it polls the request region at the request slot number $s \cdot (W \cdot N_c) + (c \cdot W) + r \ mod\ W$.

A network configuration using bidirectional, all-to-all, communication with connected transports would require $N_C \cdot N_S$ queue pairs at the server. HERD, however, uses connected transports for only the request side of communication, and thus requires only N_C connected queue pairs. The initial setup works as follows. An initializer process creates the request region, registers it with NIC_S, establishes a UC connection with each client, and goes to sleep. The N_S server processes then map the request region into their address space via `shmget()` and do not create additional connections for receiving requests.

4.4.4 Response format and handling

In HERD, responses are sent as SENDs over UD. Each client creates N_S UD QPs, whereas each server process uses only one UD QP. Before writing a new request to server process s, a client posts a RECV to its s-th UD QP. This RECV specifies the memory area on the client where the server's response will be written. Each client allocates a response region containing $W \cdot N_S$ response slots: this region is used for the target addresses in the RECVs. After writing out W requests, the client starts checking for responses by polling for RECV completions. On each successful completion, it posts another request.

The design outlined thus far deliberately shifts work from the server's NIC to the client's, with the assumption that client machines often perform enough other work that saturating their network bandwidth is not their primary concern. The servers, however, in an application such as Memcached, are often dedicated machines, and achieving high bandwidth is important.

4.5 Evaluation

We evaluate HERD primarily on the InfiniBand-based CX3 cluster. To evaluate HERD on RoCE, we use NSF PRObE's [51] Susitna cluster. Susitna has 36 hosts, equipped with AMD's Opteron 6272 CPUs, and Mellanox ConnectX-3 NICs connected via PCIe 2.0x8. Although Susitna uses a similar NIC model as CX3, the slower PCIe 2.0 bus reduces the throughput of all compared systems. Our evaluation shows that:

- HERD uses the full processing power of the NIC. A single HERD server can process up to 26 million requests per second on ConnectX-3 NICs. For value size up to 60 B, HERD's request throughput is higher than even native READ throughput, and is over 2x higher than that of FaRM-KV and Pilaf.
- HERD delivers its 26 Mops on CX3 with approximately 5 µs average latency. Its latency is over 2x lower than FaRM-KV and Pilaf at their peak throughput, respectively.
- HERD scales to the moderately-sized CX3 cluster, sustaining peak throughput with over 250 connected client processes.

4.5.1 Experimental setup

Unless stated otherwise, we run our throughput and latency experiments on 18 machines in CX3. The 17 client machines run up to three client processes each. With at most four outstanding requests per client, our implementation requires at least 36 client processes to saturate the server's throughput. We over-provision slightly by using 51 client processes. The server machine runs six server processes, each pinned to a distinct physical core.

Comparison against stripped-down alternatives. In keeping with our focus on understanding the effects of network-related decisions, we compare our (full) HERD implementation against simplified implementations of Pilaf and FaRM-KV. These simplified implementations use the same communication methods as the originals, but *omit* the actual key-value storage, instead returning a result instantly. We made this decision for two reasons. First, while working with Pilaf's code, we observed several optimization opportunities; we did not want our evaluation to depend on the relative performance tuning of the systems. Second, we did not have access to FaRM's source code. Instead, we created and evaluated emulated versions of the two systems, which do not include their backing data structures. This approach gives these systems the maximum performance advantage possible, so the throughput we report for both Pilaf and FaRM-KV may be higher than is actually achievable by those systems.

Pilaf is based on two-level lookups: a hash-table maps keys to pointers. The pointer refers to the key's value, stored in flat memory regions called *extents*. FaRM-KV, in its default operating mode, uses single-level lookups. It achieves this by inlining values hash-table buckets. It also has a two-level mode, where the value is stored "out-of-table." Because the out-of-table mode is necessary for memory efficiency with variable length keys, we compare HERD against both modes. In the following two subsections, we denote the size of a key, value, and pointer by S_K, S_V, and S_P respectively.

4.5.1.1 Emulating Pilaf

In K-B cuckoo hashing, K candidate buckets can store a key, determined by K orthogonal hash functions. For associativity, each bucket contains B slots. Pilaf uses 3-1 cuckoo hashing with 75% memory efficiency and 1.6 average probes per `GET` (higher memory efficiency with fewer, but slightly larger, average probes is possible with 2-4 cuckoo hashing [44]). When reading the hash index with an RDMA read, the smallest unit that must be read is a bucket. A bucket in Pilaf has only one slot that contains a 4 B pointer, two 8 B checksums, and a few other fields. We assume the bucket size in Pilaf to be 32 B for alignment.

`GET`: A `GET` in Pilaf consists of 1.6 bucket READs (on average) to find the value pointer, followed by a S_V-byte READ to fetch the value. It is possible to reduce Pilaf's latency by issuing concurrent READs for both cuckoo buckets. As this comes at the cost of decreased throughput, we wait for the first READ to complete and issue the second READ only if it is required.

`PUT`: For a `PUT`, a client SENDs a $S_K + S_V$ byte message containing the new key-value item to the server. This request may require relocating entries in the cuckoo hash-table, but we ignore that as our evaluation focuses on the network communication only.

In emulating Pilaf, we enable all of our RDMA optimizations for both request types; we call the resulting system Pilaf-em-OPT.

4.5.1.2 Emulating FaRM-KV

FaRM-KV uses a variant of hopscotch hashing to locate a key in approximately one READ. Its algorithm guarantees that a key-value pair is stored in a small neighborhood of the bucket that the key hashes to. The size of the neighborhood is tunable, but its authors set it to six to balance good space utilization and performance for items smaller than 128 B. FaRM-KV can inline the values in the buckets, or it can store them separately and only store pointers in the buckets. We call our version of FaRM-KV with inlined values FaRM-em and without inlining FaRM-em-VAR (for variable length values).

GET: A GET in FaRM-em requires a 6 * ($S_K + S_V$) byte READ. In FaRM-em-VAR, a GET requires a 6 * ($S_K + S_P$) byte READ followed by a S_V byte READ.

PUT: FaRM-KV handles PUTs by sending messages to the server via WRITEs, similar to HERD. The server notifies the client of PUT completion using another WRITE. Therefore, a PUT in FaRM-em (and FaRM-em-VAR) consists of one $S_K + S_V$ byte WRITE from a client to the server, and one WRITE from the server to the client. For higher throughput, we perform these WRITEs over UC unlike the original FaRM paper, which used RC (Figure 4.4).

4.5.2 Workloads

Three main workload parameters affect the throughput and latency of a key-value system: relative frequency of PUTs and GETs, item size, and skew.

We use two types of workloads: read-intensive (95% GET, 5% PUT) and write-intensive (50% GET, 50% PUT). We use both uniform and skewed workloads. Under a uniform workload, the keys are chosen uniformly at random from the 16 B keyhash space. The skewed workload draws keys from a Zipf distribution with parameter .99. We generate the workload offline using YCSB [29]. We generated 480 million keys once and assigned 8 million keys to each of the 51 client processes.

4.5.3 Throughput comparison with one-sided RDMA approaches

We now compare the end-to-end throughput of HERD against the emulated versions of Pilaf and FaRM.

Figure 4.8 plots the throughput of these system for read-intensive and write-intensive workloads for 48 B items (S_K = 16, S_V = 32). We chose this item size because it is representative of real-life workloads: an analysis of Facebook's general-purpose key-value store [9] showed that the 50-th percentile of key sizes is approximately 30 B, and that of value sizes is 20 B. To compare the READ-based GETs of Pilaf and FaRM with Pilaf's SEND/RECV-based PUTs, we also plot the throughput when the workload consists of 100% PUTs.

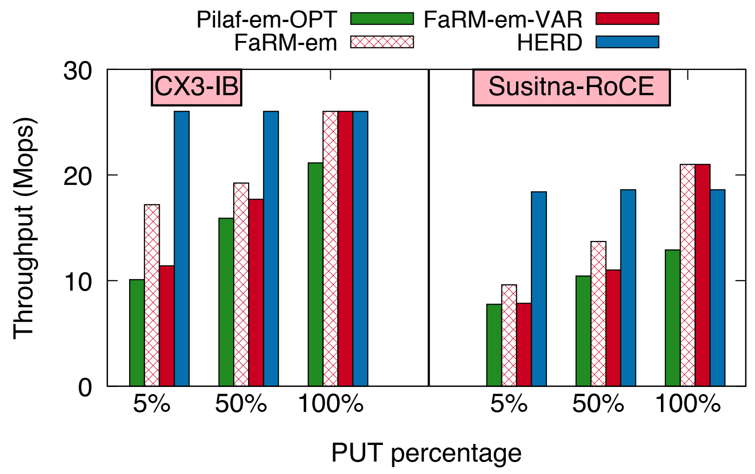


Figure 4.8: End-to-end throughput comparison for 48 B key-value items

In HERD, both read-intensive and write-intensive workloads achieve 26 Mops, which is slightly larger than the throughput of native RDMA reads of a similar size (Figure 4.2b). For small key-value items, there is little difference between PUT and GET requests at the RDMA layer because both types of requests fit inside one cache line. Therefore, the throughput does not depend on the workload composition.

The throughput of READs directly determines the GET throughput of Pilaf-em-OPT and FaRM-em(-VAR). A GET in Pilaf-em-OPT involves 2.6 READs on average. Its GET throughput is 9.9 Mops, which is about 2.6x lower than the maximum READ throughput. For GETs, FaRM-em requires a single 288 B READ, delivering 17.2 Mops. FaRM-em-VAR requires a second READ and has throughput of 11.4 Mops for GETs.

Surprisingly, the PUT throughput in our emulated systems is much larger than their GET throughput. This is explained as follows. In FaRM-em(-VAR), PUTs use small WRITEs over UC that outperform the large READs required for GETs. Pilaf-em-OPT uses SEND/RECV-based requests and replies for PUT. Both Pilaf and FaRM assume that messaging-based ECHOs are much more expensive than READs. (Pilaf reports that for 17 B messages, the throughput of RDMA reads is 2.449 Mops whereas the throughput of SEND/RECV-based ECHOs is only 0.668 Mops.) If SEND/RECV can provide only one fourth the throughput of READ, it makes sense to use multiple READs for GET.

However, we believe that these systems do not achieve the full capacity of SEND/RECV. After optimizing SENDs by using unreliable transport, payload inlining, and unsignaled transmission, SEND/RECV based ECHOs, as shown in Figure 4.4, achieve 21 Mops, which is considerably more than half of our READ throughput (26 Mops). Therefore, we conclude that SEND/RECV-based communication, when used effectively, is more efficient than using multiple READs per request.

Figure 4.9 shows the throughput of the three systems with 16 B keys and different value sizes for a read-intensive workload. For up to 60 Mops items, HERD delivers over 26 Mops, which is slightly greater than the peak READ throughput. Up to 32 B values, FaRM-em also delivers high

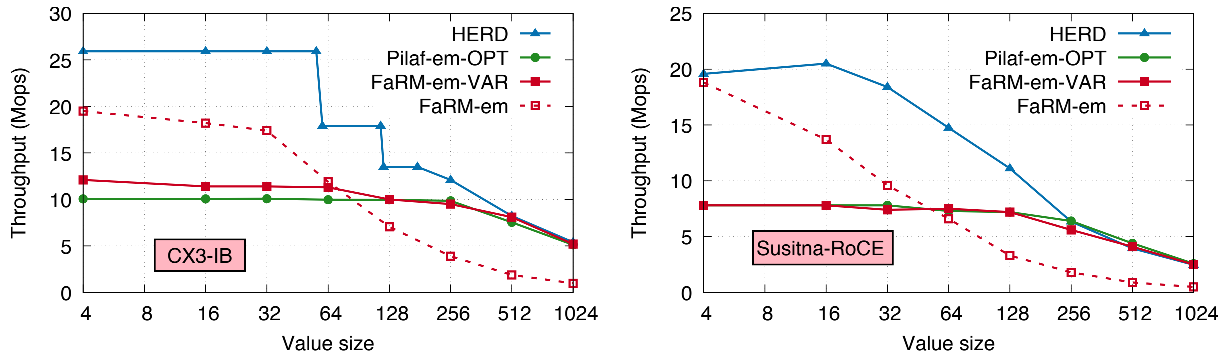


Figure 4.9: End-to-end throughput comparison with different value sizes

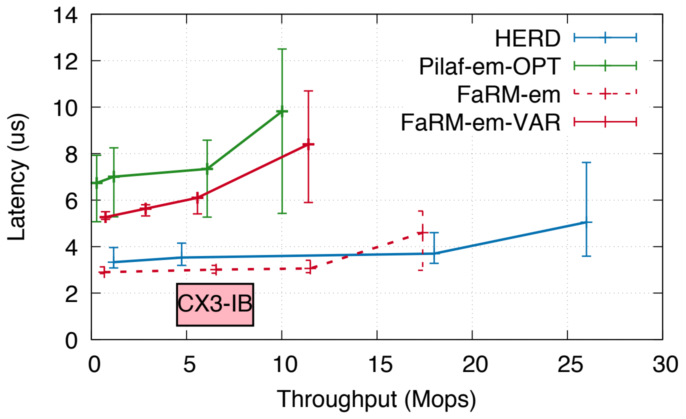


Figure 4.10: End-to-end latency with 48 B items and read-intensive workload

throughput. However, its throughput declines quickly with increasing value size because the size of FaRM-em's READs grow rapidly (as 6 * $(S_V + 16)$). This problem is fundamental to the hopscotch-based data structure that amplifies the READ size to reduce round trips. FaRM-KV quickly saturates link bandwidths (PCIe or InfiniBand/RoCE) with smaller items than HERD, which conserves network bandwidth by transmitting only necessary data. Figure 4.9 illustrates this effect. FaRM-em saturates the PCIe 2.0 bandwidth on Susitna with 4 B values, and the 56 Gbps InfiniBand bandwidth on CX3 with 32 B values.

HERD hits the PCIe MMIO bottleneck for up to 32 B values on Susitna, and 60 B values on CX3. With large values (144 B on CX3, 192 B on Susitna), HERD switches to using non-inlined SENDs for responses. The outbound throughput of large inlined messages is less than non-inlined messages because DMA outperforms MMIO for large payloads. For large values, the performance of HERD, FaRM-em, and Pilaf-em-OPT are within 10% of each other.

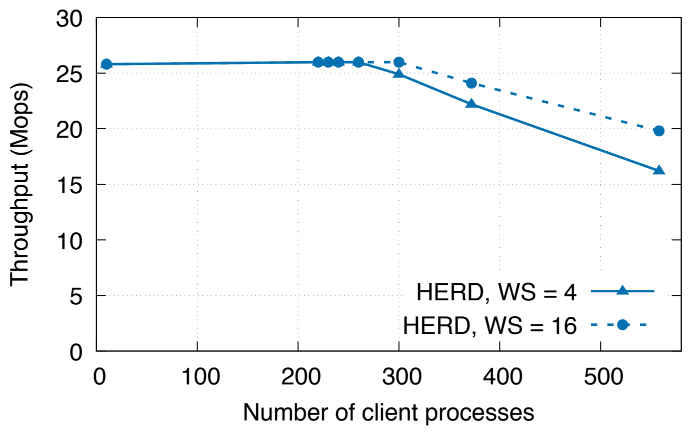


Figure 4.11: Throughput with variable number of client processes and different window sizes

4.5.4 Latency comparison with one-sided RDMA approaches

Unlike FaRM-KV and Pilaf, HERD uses only one network round trip for any request. FaRM-KV and Pilaf use one round trip for PUT requests but require multiple round trips for GETs, except when FaRM-KV inlines values in the hash-table. This causes their GET latency to be higher than the latency of a single RDMA READ.

Figure 4.10 compares the average latencies of the three systems for a read-intensive workload; the error bars indicate the 5th and 95th percentile latency. To understand the dependency of latency on throughput, we increase the load on the server by adding more clients until the server is saturated. When using six CPU cores at the server, HERD is able to deliver 26 million requests per second with approximately 5 µs average latency. For fixed-length key-value items, FaRM-em provides the lowest latency among the three systems because it requires only one network round trip (unlike Pilaf-em-OPT) and no computation at the server (unlike HERD). For variable length values, however, FaRM's variable length mode requires two RTTs, yielding worse latency than HERD.

The PUT latency for all three systems (not shown) is similar because the requests traverse the same network path. HERD's latency is slightly higher than that of the emulated systems because it performs actual hash table and memory manipulation for inserts, but this is an artifact of the performance advantage we give Pilaf-em and FaRM-em.

4.5.5 Cluster scalability

We conducted a larger experiment to understand HERD's performance as we increase the number of clients. We used one machine to run 6 server processes and the remaining 186 machines for client processes. The experiment uses 16 B keys and 32 B values.

Figure 4.11 shows the results from this experiment. HERD delivers its maximum throughput for up to 260 client processes. With even more clients, HERD's throughput starts decreasing

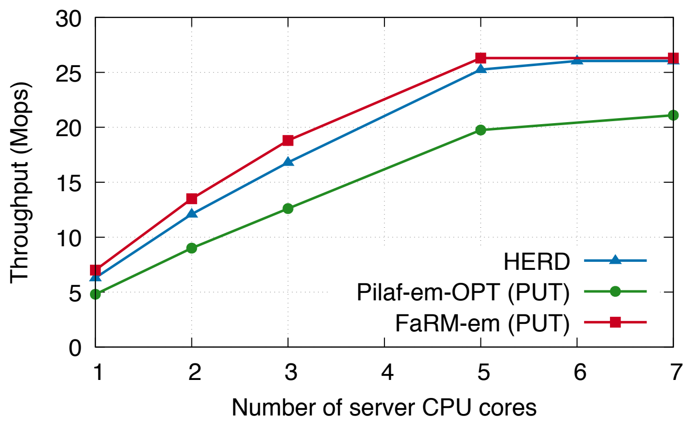


Figure 4.12: Throughput as a function of the number of server CPU cores

almost linearly. Increasing the number of outstanding requests per client reduces the rate of performance degradation, at the cost of higher request latency. Figure 4.11 shows the results for two window sizes: 4 (HERD's default) and 16. This observation suggests that the decline is due to cache misses in NIC_S, as more outstanding verbs in a queue can reduce cache pressure.

Another likely scalability limit of our current HERD design is the server's round-robin polling for requests. With thousands of clients, using WRITEs for inbound requests may incur too much CPU overhead; mitigating this effect may necessitate switching to a SEND/SEND architecture over Unreliable Datagram transport. Figure 4.4 shows there is a 4–5 Mops decrease to this change, but once made, the system should scale up to many thousands of clients, while still outperforming an RDMA read–based architecture.[1]

4.5.6 CPU use

A seeming drawback of the HERD's RPC-based design relative to CPU-bypassing designs is its higher server CPU use. Below, we put this in context with the *total* (client + server) CPU use in all systems.

The primary drawback of not using READs in HERD is that `GET` operations require the server CPU to execute requests, in exchange for saving one cross-datacenter RTT. While at first glance, it might seem that HERD's CPU usage should be higher than Pilaf and FaRM-KV, we show that in practice these two systems also have significant sources of CPU usage that reduce the extent of the difference.

First, issuing extra READs adds CPU overhead at the Pilaf and FaRM-KV clients. To issue the second READ, the clients must poll for the first READ to complete. HERD shifts this overhead to the server's CPU, making more room for application processing at the clients.

[1]Figure 4.4 uses SENDs over UC, but we have verified that similar throughput is possible using SENDs over UD.

Second, handling PUT requests requires CPU involvement at the server. Achieving low-latency PUTs requires dedicating server CPU cores that poll for incoming requests. Therefore, the exact CPU use depends on the fraction of PUT throughput that server is *provisioned* for, because this determines the CPU resources that must be allocated to it, not the dynamic amount actually used. For example, our experiments show that, even ignoring the cost of updating data structures, provisioning for 100% PUT throughput in Pilaf and FaRM-KV requires over five CPU cores. Figure 4.12 shows FaRM-em and Pilaf-em-OPT's PUT throughput for 48 B key-value items and different numbers of CPU cores at the server. Pilaf-em-OPT's CPU usage is higher because it must post RECVs for new PUT requests, which is more expensive than FaRM-em's request-region polling.

In Figure 4.12, we also plot HERD's throughput for the same workload by varying the number of server CPU cores. HERD is able to deliver over 95% of its maximum throughput with 5 CPU cores. The modest gap to FaRM-em arises because the HERD server in this experiment is handling hash table lookups and updates, whereas the emulated FaRM-KV is handling only the network traffic.

We believe, therefore, that HERD's higher throughput and lower latency, along with the significant CPU utilization in Pilaf and FaRM-KV, justifies the architectural decision to have the CPU involved on the GET path for small key-value items. For a 50% PUT workload, for example, the moderate extra cost of adding a few more cores—or using the already-idle cycles on the cores—is likely worthwhile for many applications.

4.5.7 Resistance to workload skew

To understand how HERD's behavior is impacted by skew, we tested it with a workload where the keys are drawn from a Zipf distribution. HERD adapts well to skew, delivering its maximum performance even when the Zipf parameter is 0.99. HERD's resistance to skew comes from two factors. First, the back-end MICA architecture [97] that we use in HERD performs well under skew; a skewed workload spread across several partitions produces little variation in the partitions' load compared to the skew in the workload's distribution. Under our Zipf-distributed workload, with six partitions, the most loaded CPU core is only 50% more so than the least loaded core, even though the most popular key is over 10^5 times more popular than the average.

Second, because the CPU cores share the NIC, the highly loaded cores are able to benefit from the idle time provided by the less-used cores. Figure 4.12 demonstrates this effect: with a *uniform* workload and using only a single core, HERD can deliver 6.3 Mops. When the system is configured to use six cores—the minimum required by HERD to deliver its peak throughput—the system delivers 4.32 Mops *per core*. The per-core performance reduction is not because of a CPU bottleneck, but because the server processes saturate the PCIe MMIO throughput. Therefore, even if the workload is skewed, there is ample CPU headroom on a given core to handle the extra requests.

Figure 4.13 shows the per-core throughput of HERD for a skewed workload. The experimental configuration is: 48 B items, read-intensive, skewed workload, with six server CPU

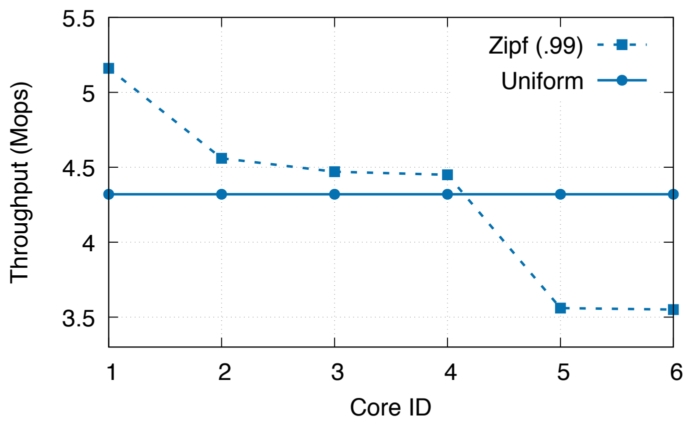


Figure 4.13: Per-core throughput under skewed and uniform workloads. Note that the y-axis does not begin at zero.

cores. We include the per-core throughput for a uniform workload for comparison.

4.6 Revisiting HERD's design for faster NICs

Until now, we have presented the design and evaluation of our original HERD system published in SIGCOMM 2014 [79], which uses Mellanox ConnectX-3 NICs. We later received access to the CIB cluster (Table 2.2) that has newer Mellanox Connect-IB NICs, which can handle up to four times more messages per second than ConnectX-3 NICs. This provided us an opportunity to revisit HERD's design in the context of both faster NICs and newer one-sided RDMA–based key-value stores.

4.6.1 Applying Doorbell batching to HERD

When we evaluated HERD's original design on CIB, we found that the server's CPU could not keep up with the NIC's higher message rate: the bottleneck had shifted from the NIC's processing power in our original evaluation (that used slower ConnectX-3 NICs) to the CPU's processing power. We reduced HERD's server CPU use by applying Doorbell batching (Guideline 3.6.2.1), as follows. Instead of handling requests one-by-one, the server processes all available requests in a batch. After processing the requests, instead of sending responses one-by-one using Descriptor-MMIO, the server sends the entire batch of responses using one Doorbell. Note that server-side batching does not significantly increase request latency because we do it opportunistically [80, 97], i.e., our server never waits for a number of requests to accumulate.

We evaluate the effect of Doorbell batching on HERD's performance on CIB using the following setup. We run a HERD server with a variable number of threads. We use 128 client threads running on eight client machines to issue requests. We pre-populate the key space partition owned by each server thread with 8 million key-value pairs, which map 16 B keys to 32 B

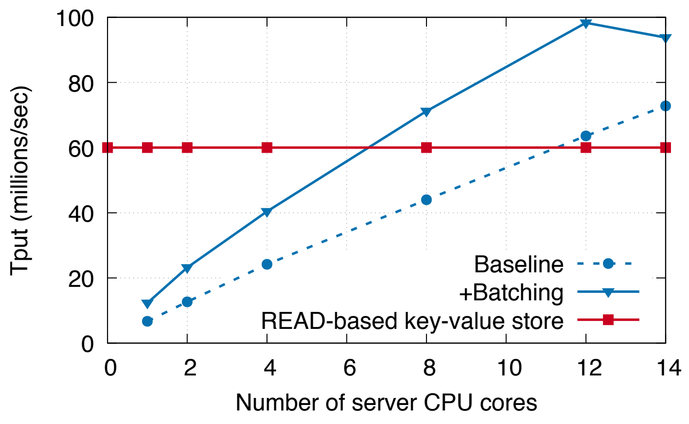


Figure 4.14: Effect of Doorbell batching on HERD's throughput on CIB with 5% PUTs

values. The workload consists of 95% GET and 5% PUT operations, with keys chosen uniformly at random from the inserted keys.

Figure 4.14 shows HERD's throughput in the above experiment. We also include the maximum throughput achievable by a READ-based key-value store such as Pilaf or FaRM-KV that uses two or more RDMA reads per GET, computed analytically by halving CIB's peak inbound RDMA read throughput of 120 Mops. We make three observations. First, batching improves HERD's single-core throughput by 83% from 6.7 Mops to 12.3 Mops. Second, it improves the server's peak throughput by 35% from 72.8 Mops to 98.3 Mops. Batched throughput is bottlenecked by PCIe DMA bandwidth: The DMA bandwidth limit for outbound, batched UD SENDs is 101.6 million operations/s (Section 3.6.2.1). At 98.2 Mops, HERD is within 5% of this limit; we attribute the gap to PCIe link- and physical-layer overheads for the DMA-ed requests, which are absent in our SEND-only benchmark. Third, with batching, HERD's throughput is up to 63% higher than a READ-based key-value store. While HERD's original non-batched design requires 12 cores to outperform a READ-based design, the improved design with batching needs only seven CPU cores.

4.6.2 Comparison with key-value stores that use RDMA atomics

While FaRM-KV and Pilaf use RPCs to handle PUTs, subsequent RDMA-based key-value stores published after our HERD paper in 2014, such as DrTM-KV [154] and Nessie [143, 144], use RDMA atomics in combination with RDMA reads and writes to bypass the remote CPU for PUTs. However, these projects do not consider the impact of the NIC's concurrency control on performance (Guideline 3.7.2), and present performance for either GET-only (DrTM-KV) or GET-mostly (Nessie) workloads. We show that locking overhead inside the NIC results in low PUT throughput in these systems, and degrades throughput even when only a small fraction of key-value operations are PUTs.

We discuss DrTM-KV here because of its simplicity, but similar observations apply to Nessie. DrTM-KV caches some fields of its key-value index at all clients; GETs for cached keys use one

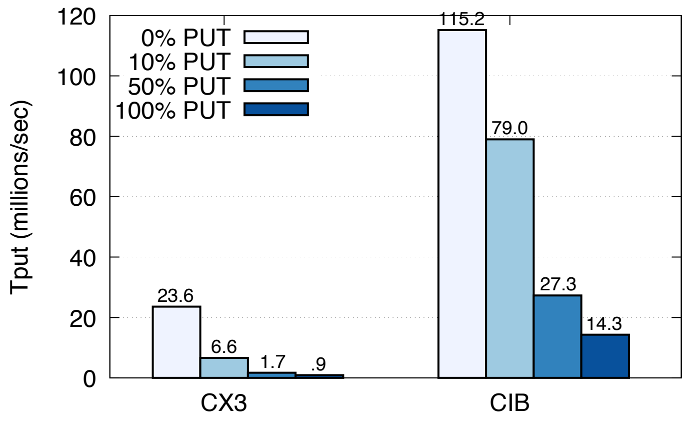


Figure 4.15: Throughput of emulated DrTM-KV for increasing fractions of PUTs

READ. (We discuss later in Section 5.2.1 that such caching is rarely effective at reducing the number of READs from two or more to one, because the cluster-level index size far exceeds the caching capacity at each host.) PUT operations lock, update, and unlock key-value items; locking and unlocking uses RDMA atomics. We give our emulated version of DrTM-KV a performance advantage by assuming a 100% cache hit rate, emulating its GETs with one READ, and its PUTs with two atomics.

Figure 4.15 shows the throughput of our emulated DrTM-KV server on CIB with different fractions of PUT operations in the workload. The server hosts 16 million items with 16 B keys and 32 B values. Clients choose keys uniformly at random and we use as many clients as required to maximize throughput. Although throughput for a 100% GET workload is high, adding only 10% PUTs degrades it by 72% on CX3 and 31% on CIB. Throughput with 100% PUTs is a tiny fraction of GET-only throughput: 4% on CX3 and 12% on CIB. Throughput on CIB degrades more gradually than CX3 because of the better locking mechanism in Connect-IB NICs (Section 3.7.2).

4.7 A networked sequencer with HERD RPCs

Centralized sequencers are useful building blocks for a variety of network applications, such as ordering operations in distributed systems via logical or real timestamps [16], or providing increasing offsets into a linearly growing memory region [12]. A centralized sequencer can be the bottleneck in high-performance distributed systems, so building a fast sequencer is an important step to improving whole-system performance.

We design a sequence server that runs on a single machine and provides an increasing 8 B integer to client processes running on remote machines. Our baseline design uses HERD RPCs. The server threads share an 8 B counter; each client can send a sequencer request to any thread. The server-side RPC handling consists of atomically incrementing the shared counter by one. When Doorbell batching is enabled, we use an additional application-level optimization to re-

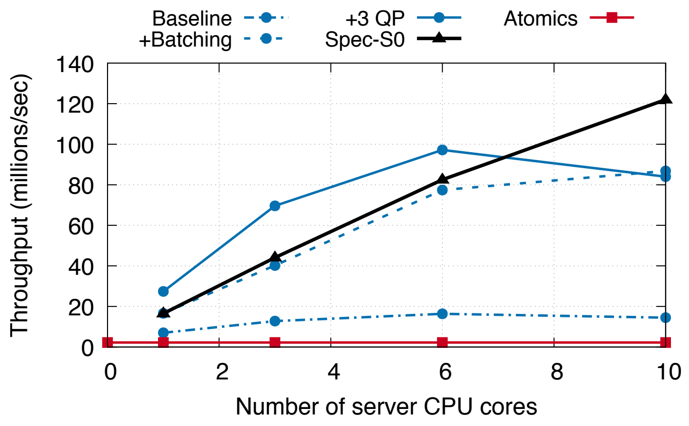


Figure 4.16: Impact of optimizations on HERD RPC–based sequencer (blue lines with circular dots), and throughput of `Spec-S0` and the atomics-based sequencer

duce contention for the shared counter: After collecting B requests, the server thread atomically increments the shared counter by B, thereby claiming ownership of a sequence of B consecutive integers. It then sends these B integers to the clients (one integer per client) using a batched Doorbell.

An important difference between our sequencer and the HERD key-value store is the much smaller per-request server-side processing in the sequencer. As a result, while per-thread server throughput is limited by CPU speed in the key-value store application, it is limited by the NIC processing unit handling the thread's UD queue pairs in the sequencer application. We relieve this bottleneck by applying the multi-queue optimization (Guideline 3.7.1), as follows: Each server thread alternates among a tunable number of UD QPs across its batched SENDs for responses.

Figure 4.16 shows the effect of batching and multi-queue on the HERD RPC–based sequencer's throughput for an increasing number of server CPU cores. We run one server thread per core and use 70 client processes spread across five client machines. Batching increases single-core throughput from 7.0 Mops to 16.6 Mops. In this mode, each core still uses two response UD QPs—one for each NIC port on CIB—and is bottlenecked by the two NIC processing units handling the QPs; engaging more PUs with multi-queue (three per-port QPs per core) increases core throughput to 27.4 Mops. With six cores, and with both Doorbell batching and multi-queue optimizations, the server's throughput increases to 97.2 Mops and is bottlenecked by PCIe DMA bandwidth. With more than six cores, throughput drops because the response batch size is smaller: With six cores (97.2 Mops), each batch contains 15.9 responses on average, which drops to 4.4 responses per batch with 10 cores (84 Mops).

4.7.1 Specializing HERD RPCs for the sequencer's workload

The design presented above uses a straightforward adoption of HERD's RPCs for a sequencer, and inherits the limitations of HERD's RPC protocol. First, the connected QPs used for RDMA-

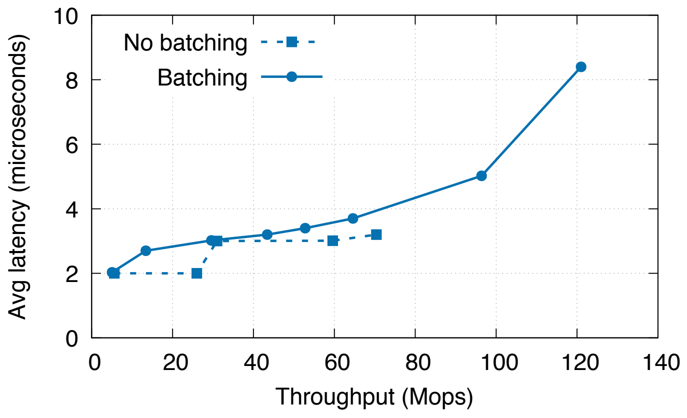


Figure 4.17: Impact of response batching on `Spec-S0` latency

writing requests require state in the server's NIC and limit scalability. Higher scalability necessitates exclusive use of datagram transports. The challenge then is to use datagrams instead of WRITEs for sequencer *requests*, without sacrificing server performance. Second, the PCIe use of HERD RPCs is inefficient for the sequencer: UD SEND queue descriptors on Mellanox's NICs span two or more cache lines because of their 68 B header; sending 8 B of useful sequencer data requires the NIC to DMA-read 128 B (two cache lines).

We exploit the specific requirements of the sequencer to overcome these limitations. We use header-only SENDs and RECVs (Guideline 3.6.2.2 and 3.6.3.3) for both requests and responses to solve both problems:

1. The client's header-only request SENDs generate header-only, single-DMA RECVs at the server, which are as fast as the RDMA writes in HERD's original design.
2. The server's header-only response SEND descriptors use a header field for application payload and fit in one cache line, reducing the data DMA-ed per response by 50%, to 64 B.

Using header-only SENDs requires encoding application information in the SEND packet header; we use the four-byte *immediate integer* field of RDMA's Unreliable Datagram packets [64]. Our 8 B sequencer works around the 4 B limit as follows: Clients speculate the four highest bytes of the counter and send it in a header-only SEND. If the client's guess is correct, the server sends the four least significant bytes in a header-only SEND, else it sends the entire eight-byte counter value in a regular, non header-only SEND, which later triggers an update of the client's guess. Only a tiny fraction ($\leq C/2^{32}$ with C clients) of SENDs require sending the full 8 B counter.

We call this datagram-only sequencer `Spec-S0` (speculation with header-only SENDs). Figure 4.16 shows its throughput with increasing server CPU cores. `Spec-S0`'s DMA bandwidth limit is higher than HERD RPCs because of smaller response descriptors; it achieves 122 Mops and is limited by the NIC's processing power instead of PCIe bandwidth. `Spec-S0` has lower single-core throughput than the HERD RPC–based sequencer because of the additional CPU overhead of posting RECVs.

	Baseline HERD RPCs	**+Batching, multi-queue**	**`Spec-S0`**	**Atomics**
Throughput	26 Mops	97.2 Mops	122 Mops	2.24 Mops
Bottleneck	CPU lock contention	PCIe DMA bandwidth	NIC processing	PCIe RTT

Table 4.1: Summary of sequencer throughput and bottlenecks on CIB

Figure 4.17 shows the average end-to-end latency of `Spec-S0` with and without response batching. Both modes receive a batch of requests from the NIC; the two modes differ only in the method used to send responses. The non-batched mode sends responses one-by-one using Descriptor-MMIO, whereas the batched mode uses Doorbell when multiple responses are available to send. We batch atomic increments to the shared counter in both modes. We use ten server CPU cores, which is the minimum required to achieve peak throughput. We measure throughput with increasing client load by adding more clients, and by increasing the number of outstanding requests per client. Batching adds up to 1 µs of latency because of the additional DMA read in the Doorbell method. We believe that the small additional latency is acceptable because of the large throughput and CPU-efficiency gains from batching.

4.7.2 Comparison with atomics-based sequencers

Atomic fetch-and-add over RDMA is an appealing method to implement a sequencer: Binnig et al. [16] use this design for the timestamp server in their distributed transaction protocol. However, lock contention for the counter among the NIC's PUs results in poor performance. The *duration* for which locks are held—several hundred nanoseconds for PCIe round trips—exacerbates the effect of contention. Our RPC-based sequencers have lower contention and shorter lock duration: the programmability of general-purpose CPUs allows us to batch updates to the counter which reduces cache line contention, and proximity to the counter's storage (i.e., core caches) makes these updates fast. Figure 4.16 shows the throughput of our atomics-based sequencer on CIB: it achieves only 2.24 Mops, which is 50x worse than our optimized design, and 12.2x worse than our single-core throughput.

Table 4.1 summarizes the performance of our sequencers. Interestingly, each design hits a different hardware bottleneck, highlighting the importance of low-level hardware factors in reasoning about the performance of high-speed networked systems.

4.8 Related work

RDMA in the HPC research community. Jose et al. [77] describe a memcached implementation using a hybrid of UD and RC transports. Their design uses SEND/RECV messages for all requests and, for skewed workloads, avoids the overhead of UD transport by actively switching connections between RC and UD. Although their cluster (ConnectX NICs, 32 Gbps) is comparable to Susitna (ConnectX-3 NICs, 40 Gbps), their server's peak request rate is around 1.5 Mops, which is over 10x lower than HERD.

The Direct Access File System [33] 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 RPCs in the Network File System (NFS) [24] and some MPI implementations [101]. The MPI implementation for InfiniBand by Liu et al. [101] uses RDMA writes for messaging: the server polls the head of a circular buffer that is written to by a client. HERD extends this messaging in a scalable fashion for many-to-one request-reply communication.

Several other projects use RDMA verbs to improve the performance of systems such as distributed databases (HBase), Hadoop RPCs, and distributed file systems (PVFS) [61, 103, 157]. Most of these projects use only SEND/RECV verbs as a fast alternative to sockets-based communication. The distributed file system implementation over InfiniBand by Wu et al. [157] favors RDMA writes over reads for similar reasons as our work, supporting our observation that the performance gap between RDMA writes and reads has existed over several generations of RDMA hardware. While several researchers have benchmarked the performance of RDMA verbs before us [61, 101, 103, 157], they have focused on large messages in the context of applications like Network File System and MPI. Our work exploits the performance differences that appear only for small messages and are relevant for message rate–bound applications like key-value stores.

General key-value stores. MICA [97] is a recent key-value system for classical Ethernet. It assigns exclusive partitions to server cores to minimize lock contention, and exploits the NIC's capability to steer requests to the responsible core. A MICA server with 16 CPU cores and four dual-port, 10 GbE PCIe 2.0 x8 NICs delivers 77 Mops with ≈50 µs average latency (19.25 Mops with one NIC). HERD delivers comparable throughput as MICA, but an order of magnitude lower latency. One of HERD's contributions is showing that MICA's request-reply–based approach, which was designed for classical Ethernet, works better than CPU-bypassing approaches on RDMA-capable interconnects. RAMCloud [122] is a DRAM-based, persistent key-value store that uses RDMA's messaging verbs for low latency communication. RAMCloud primarily targets low latency, whereas HERD targets both low latency and high throughput.

4.9 Conclusion

This chapter explored the options for implementing fast, low-latency key-value systems atop RDMA, arriving at an unexpected and novel combination that outperforms prior designs and uses fewer network round-trips. Our work shows that, contrary to widely held beliefs about engineering for RDMA, single-RTT designs with server CPU involvement can outperform the "optimization" of CPU-bypassing remote memory access when the RDMA approaches require multiple RTTs. Taken together, one lesson from the union of HERD, Pilaf, FaRM, and MICA is that the biggest boost to throughput comes from bypassing the network stack and avoiding CPU interrupts, *not* necessarily from bypassing the CPU entirely. These results contribute not just a practical artifact—the HERD low-latency, high-performance key-value cache—but an improved understanding of how to use datacenter networks to construct high-speed distributed systems.

HERD's use of NIC-managed connections to send requests works well in its asymmetric setting with many clients and a few servers. However, in symmetric settings, where every host acts as both client and server, the large number of active outbound connections at each host prevents scaling to large clusters. In the next chapter, we show how switching over entirely to datagram transports allows a more scalable design, as well as CPU savings from Doorbell batching.

"Once you free your mind about a concept of Harmony and of music being correct, you can do whatever you want."

Giovanni Giorgio Moroder

Chapter 5

Case study 2: FaSST – Fast, Scalable, and Simple Distributed Transactions

This chapter presents the second of our two case studies: The FaSST distributed transaction processing system. In FaSST, we build and improve on the lessons from HERD to create a more flexible and scalable RPC subsystem. Unlike HERD, which uses one-sided RDMA, we choose to use exclusively datagram transports in FaSST RPCs. This decision brings two benefits. First, it makes RPC design more flexible since it does not depend on NIC support for one-sided RDMA. Second, the stateless nature of datagram transports improves scalability, allowing FaSST to handle the large number of communicating pairs in distributed transaction workloads. We show that, counter to prior assumptions, with our optimizations, datagram-based RPCs perform similarly to single one-sided RDMA operations. The insights in this chapter lay down the groundwork for even higher flexibility and generality in eRPC, presented in the next chapter.

FaSST provides distributed in-memory transactions (Section 2.2.2) with serializability and durability. Although earlier work in this space sacrificed strong transactional semantics for performance [34], recent research shows that the availability of fast networks and non-volatile memory in datacenters now allows high-performance distributed transactions with strong consistency [17, 26, 39, 154].[1] A common thread in these recent systems is that they rely on one-sided RDMA for performance. The intent behind this decision is to harness one-sided RDMA's ability to save remote CPU cycles.

We show that RPCs implemented over scalable unreliable datagrams are a better primitive than one-sided RDMA for transactions. Compared to prior systems that use one-sided RDMA, FaSST has all four advantages listed in Chapter 1. First, FaSST is faster because it uses fewer round trips during the data access phase of transactions. Second, FaSST is more flexible because it does not depend on network support for one-sided RDMA. Third, FaSST is more scalable because it uses only connectionless datagram transports. Fourth, FaSST is simpler because of the higher generality of RPCs compared to RDMA.

[1]We discuss only distributed transactions in this work, so we use the more general but shorter term transactions.

Previous high-speed RPC implementations do not perform well in the cluster setting required for transactions. The prior highest-performing RPC design—HERD RPCs—deliver high performance in an *asymmetric* setting, with many-to-one communication where one server handles RPCs from many clients, but their design does not scale well with all-to-all communication. However, modern high-speed transaction processing systems typically operate in the *symmetric* setting, where every host acts as both transaction client and server [39]. This allows co-locating data with computation when possible, improving performance. HERD's RPCs do not perform well in the symmetric setting because of NIC cache misses caused by the large number of active connected QPs at each transaction processing host.

The key contribution described in this chapter is FaSST RPCs: an all-to-all RPC system that is fast, scalable, and CPU-efficient. This is made possible by using only datagrams, which we show can achieve high performance with our optimizations, such as CPU savings from Doorbell batching (Guideline 3.6.2). We show that FaSST RPCs provide (1) up to 8x higher throughput, and 13.9x higher CPU efficiency than FaRM's RPCs (Section 5.3.5), and (2) 1.7–2.15x higher CPU efficiency, or higher throughput, than one-sided READs, depending on whether or not the READs scale to clusters with more than a few tens of nodes (Section 5.2.3).

A novel aspect of FaSST is its handling of lost datagrams. FaSST targets lossless networks with link-layer flow control, such as InfiniBand, RoCE and OmniPath. In these networks, packet loss is extremely rare because the underlying link layer provides reliability. We did not observe any lost packets in our experiments that transmitted over 50 PB of network data on a real-world InfiniBand cluster with up to 69 nodes. Nevertheless, packet loss can occur during hardware failures, and corner cases of the link-layer's reliability protocol. We detect these losses with low CPU overhead using coarse-grained timeouts triggered at the RPC requester, and describe how they can be handled similarly to conventional machine failures.

FaSST uses optimistic concurrency control, two-phase commit, and primary-backup replication for transactions. Our current implementation supports transactions on an unordered key-value store based on MICA [97], and maps 8 B keys to opaque objects. We evaluate FaSST using three workloads: a transactional object store, a read-mostly OLTP benchmark called TATP, and a write-intensive OLTP benchmark called SmallBank. FaSST compares favorably against published per-machine throughput numbers. On TATP, FaSST outperforms FaRM by 1.87x when using close to half the hardware (NIC and CPU) resources. On SmallBank, FaSST outperforms DrTM+R [26] by 1.68x with similar hardware without making data locality assumptions.

5.1 Distributed transactions background

This section outlines the environment that we target with FaSST. FaSST aims to provide distributed transactions inside a single datacenter where an instance of the system can scale to a few hundred nodes. Each node in the system is responsible for a partition of the data based on a primary key, and nodes operate in the *symmetric model*, whereby each node acts both as a client and a server. For workloads with good data locality (e.g., transactions that only access data in one partition), the symmetric model can achieve higher performance by co-locating transactions with the data they access [38, 39].

FaSST targets high-speed, low-latency key-value transaction processing with throughputs of several million transactions/sec and average latencies around one hundred microseconds on common OLTP benchmarks with short transactions with up to a few tens of keys. Achieving this performance requires in-memory transaction processing, and fast userspace network I/O with polling (i.e., the overhead of a kernel network stack or interrupts is unacceptable). We assume commercially available network equipment: 10-100 Gbps of per-port bandwidth and $\approx 2\,\mu s$ end-to-end latency.

Making data durable across machine failures requires logging transactions to persistent storage, and quick recovery requires maintaining multiple replicas of the data store. Keeping persistent storage such as disk or SSDs on the critical path of transactions limits performance. Similar to recent work, FaSST assumes that the transaction processing nodes are equipped with battery-backed DRAM [39], though current NVM technologies, such as Intel's Optane DC Persistent Memory, would also work.

Finally, FaSST uses primary-backup replication to achieve fault tolerance. We assume that failures will be handled using a separate fault-tolerant configuration manager that is off of the critical path (the Vertical Paxos model [86]), similar to recent work on RDMA-based distributed transactions [26, 39]. We do not currently implement such a configuration manager.

5.2 Choosing networking primitives

We now describe the rationale behind our decision to build an RPC layer using two-sided datagram verbs. We show that FaSST's RPCs are:

1. **Fast.** Although READs can outperform similarly-sized FaSST RPCs on small clusters, FaSST RPCs perform better when accounting for the amplification in size or number of READs required to access real data stores.
2. **Scalable.** FaSST RPCs' throughput and CPU use remains stable as the cluster size increases, whereas READ performance degrades because READs must use connected transport with today's NICs.
3. **Simple.** RPCs reduce the software complexity required to design distributed data stores and transactions compared to one-sided RDMA-based systems.

5.2.1 Advantages of RPCs for transactions

Recent work on designing distributed OLTP systems for modern datacenter networks has largely focused on how to use one-sided RDMA primitives. Similar to key-value stores designed for one-sided RDMA, in these designs, clients access remote data structures in servers' memory using one or more READs, similar to how one would access data in local memory. Various optimizations help reduce the number of READs needed; we discuss two such optimizations and their limitations below.

Value-in-index. As discussed in Section 4.5.1.2, FaRM supports hash table access in $\approx$ 1 READ on average, but at the cost of READ-ing 6–8x more data than strictly necessary. This amplification highlights the importance of comparing the application-level capabilities of networking primitives: although micro-benchmarks suggest that READs can outperform similar-sized RPCs, READs require extra network traffic and/or round-trips due to their CPU-bypassing nature, tipping the scales in the other direction.

Caching the index. DrTM [26, 154] caches the index of its hash table at all servers in the cluster, allowing single-READ GETs; FaRM uses a similar approach for its B-Tree. Although this approach works well when the workload has high locality or skew, it does not work in general because indexes can be large: Zhang et al. [160] report that indexes occupy over 35% of memory for popular OLTP benchmarks in a single-node transaction processing system; the percentage is similar in our implementation of distributed transaction processing benchmarks. In this case, caching even 10% of the index requires *each* machine to dedicate 3.5% of the *total cluster memory capacity* for the index, which is impossible if the cluster contains more than $100/3.5 \approx 29$ nodes. The Cell B-Tree [114] caches B-Tree nodes 4 levels above the leaf nodes to save memory and reduce churn, but requires multiple round trips (~ 4) when clients access the B-Tree using READs.

RPCs allow access to partitioned data stores in one round trip. They do not require message size amplification, multiple round trips, or caching. The simplicity of RPC-based programming reduces the software complexity required to take advantage of modern fast networks in transaction processing: to implement a partitioned, distributed data store, the user writes only short RPC handlers for a single-node data store. This approach eliminates the software complexity required for one-sided RDMA-based approaches. For example, FaSST uses MICA's hash table design [97] for unordered key-value storage. We made only minor modifications to the MICA codebase to support distributed transactions.

5.2.2 Advantages of datagram transport

Datagram transport allows each CPU core to create one datagram queue that can communicate with all remote cores. Since the number of queues is relatively small (as many as the number of cores), providing each core exclusive access to a queue is possible without overflowing the NIC's cache. Providing exclusive access to RDMA's RC connections, however, is not scalable: In a cluster with N machines and T threads per machine, doing so requires $N * T$ connections at every machine, which may not fit in the NIC's cache. Threads can share connections to reduce their NIC memory footprint [38]. However, sharing connections reduces CPU efficiency because threads contend for locks, and the cache lines for queue descriptors bounce between CPU cores. The effect can be dramatic: in our experiments, connection sharing reduces the per-core throughput of READs by up to 5.4x (Section 5.2.3.2). Similarly, FaRM's RPCs that use one-sided WRITEs and connection sharing become CPU-bottlenecked at 5 million requests/sec (Mrps) per machine [39]. Our datagram-based RPCs, however, do not require QP sharing and achieve up to 40.9 Mrps per machine, and even then they are bottlenecked by the NIC, not CPU (Section 5.2.3.1).

In comparison with connected transports, datagram transport confers a second important advantage in addition to scalability: Doorbell batching, which reduces CPU use (Guideline 3.6.2). In transactional systems, applications can amortize the cost of posting descriptors to the datagram queue by ringing the doorbell once for multiple descriptors. Examples include reading or validating multiple keys for multi-key transactions, or sending update messages to the replicas of a key. With a datagram queue, the process only needs to ring the Doorbell once per batch, regardless of the individual message destinations within the batch. With connected QPs, however, the process must ring multiple Doorbells—as many as the number of destinations appearing in the batch. Note that Doorbell batching does not put multiple application-level messages in a single network packet. Doorbell batching also does not add latency because we do it opportunistically, i.e., we do not wait for a batch of messages to accumulate.

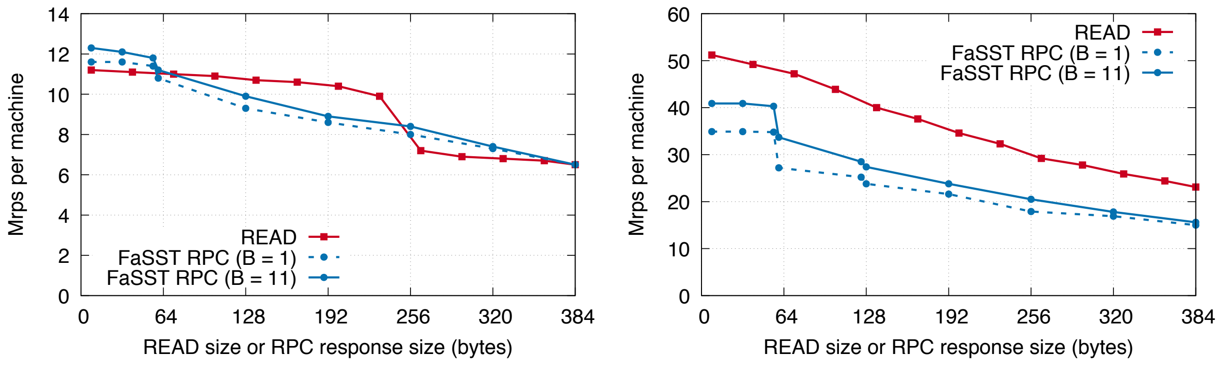
5.2.3 Performance considerations

HERD's RPC performs similarly to READs, but only in the asymmetric setting where multiple clients send requests to one server. In this setting, HERD's approach scales well with the number of clients because the number of active queues at the server is small. The server's connected QPs are passive because the server's CPU does not access them; these passive QPs consume little memory in the NIC. The active datagram queues are few in number.

In FaRM, Dragojević et al. [39] note that HERD's RPC design does not scale well in the symmetric setting required for distributed transactions, where every machine issues requests and responses. This scenario requires many active QPs on each node for sending requests. In FaRM's experiments [39] in the symmetric setting, READs outperform *their* RPCs by 4x.

We now present experimental results showing that FaSST's datagram-based RPCs are a better choice than one-sided RDMA for distributed transactions. We discuss the design and implementation of FaSST's RPC subsystem in detail in Section 5.3; here, we use it to implement basic RPCs where both the request and reply are fixed-size buffers. We first compare the raw throughput of RPCs and one-sided READs by using a small cluster where READs do not require connection sharing. Next, we compare their performance on more realistic, medium-sized clusters.

Experiment setup. We conduct our experiments on the CX3 and CIB clusters (Table 2.2). We use machines in a symmetric setting, i.e., every machine issues requests (RPC requests or READs) to every other machine. For READs without connection sharing, each thread creates as many RC QPs as the number of machines, and issues READs to randomly chosen machines. We evaluate RPC performance for two request batch sizes (1 and 11) to show the effect of Doorbell batching for requests. We prevent RPC-level request coalescing (Section 5.3) by sending each request in a batch to a different machine; this restricts our maximum batch size on CIB to 11.



(a) CX3 cluster (ConnectX-3 NICs) **(b)** CIB cluster (Connect-IB NICs)

Figure 5.1: Small clusters: Throughput comparison of FaSST RPCs (11 nodes) and READs (6 nodes). Note that the two graphs use a different Y scale.

We compare RPC and READ performance for different response sizes; for RPCs, we fix the request size at 32 B, which is sufficient to read from FaSST's data stores. We report millions of requests per second per machine (Mrps/machine). Note that for RPCs, each machine's CPU also serves responses to requests from other machines, so the number of messages sent by a machine is approximately twice the request rate that we report. Our results show that:

1. FaSST RPCs provide good raw throughput. For small messages up to 56 B, RPCs deliver a significant percentage of the maximum throughput of similar-sized READs on small clusters: 103–106% on CX3 and 68–80% on CIB, depending on the request batch size. When accounting for the amplification in READ size or number required to access data structures in real data stores, RPCs deliver higher raw throughput than READs.

2. On medium-sized clusters, if READs do not share connections, RPCs provide 1.38x and 10.1x higher throughput on CIB and CX3, respectively. If READs do share connections, their CPU efficiency drops by up to 5.4x, and RPCs provide 1.7–2.15x higher CPU efficiency.

These experiments highlight the sometimes dramatic difference in performance between micro-benchmarks and more realistic settings.

5.2.3.1 On small clusters

To measure the maximum raw throughput of READs, we use six nodes so that only a small number of queues are needed even for READs: each node on CX3 (8 cores) and CIB (14 cores) uses 48 and 84 QPs, respectively. We use 11 nodes for RPCs to measure performance with a large request batch size—using only 6 nodes for RPCs would restrict the maximum non-coalesced request batch size to 6. (As shown in Section 5.2.3.2, using 11 nodes for READs gives lower throughput due to cache misses in the NIC, so we use fewer nodes to measure their peak throughput.) Figure 5.1 shows Mrps/machine for READs and RPCs on the two clusters.

Raw throughput. Depending on the request batch size, FaSST RPCs deliver up to 11.6–12.3 Mrps on CX3, and 34.9–40.9 Mrps on CIB. READs deliver up to 11.2 Mrps on CX3, and 51.2 Mrps on CIB. The throughput of both RPCs and READs is bottlenecked by the NIC: although our experiment used all cores on both clusters, fewer cores can achieve similar throughput, indicating that the CPU is not the bottleneck.

Comparison with READs. Although RPCs usually deliver lower throughput than READs, the difference is small. For response sizes up to 56 B, which are common in OLTP, RPC throughput is within 103–106% of READ throughput on CX3, and 68–80% of READ throughput on CIB, depending on the request batch size. For larger responses, READs usually outperform our RPCs, but the difference is smaller than 4x, as is the case for FaRM's RPCs. This is because FaSST's RPCs are bottlenecked by the NIC on both clusters, whereas FaRM's RPCs become CPU-bottlenecked due to QP sharing (Section 5.2.3.2). As noted above, these "raw" results are only baseline micro-benchmarks; the following paragraphs consider the numbers in the context of real-world settings.

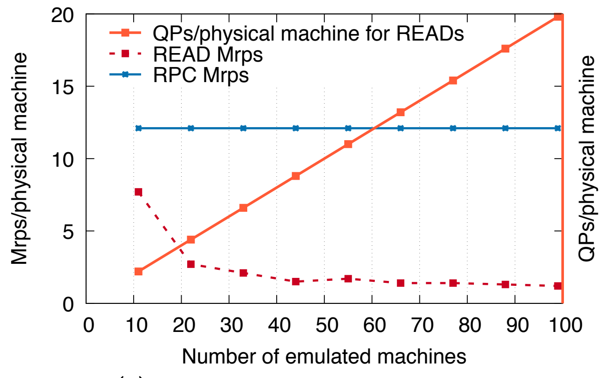
Effect of multiple READs. In all cases (i.e., regardless of cluster used, response size, and request batch size), RPCs provide higher throughput than using 2 READs. Thus, for any data store/data structure that requires two or more READs, RPCs provide strictly higher throughput.

Effect of larger READs. Consider, for example, a hash table that maps 8 B keys to 40 B values. This configuration is used in one of the database tables in the TATP benchmark in Section 5.5). For this hash table, FaRM's single-READ GETs require approximately 384 B READs (8x amplification) and can achieve up to 6.5 Mrps/machine on CX3. With FaSST RPCs, these key-value requests require one RPC with an 8 B request and a 40 B response (excluding header overheads), and can achieve 11.4–11.8 Mrps/machine (over 75% higher) before the ConnectX-3 NIC becomes the bottleneck. On CIB, 384 B READs achieve 23.1 Mrps, whereas FaSST RPCs achieve 34.9–40.9 Mrps (over 51% higher).

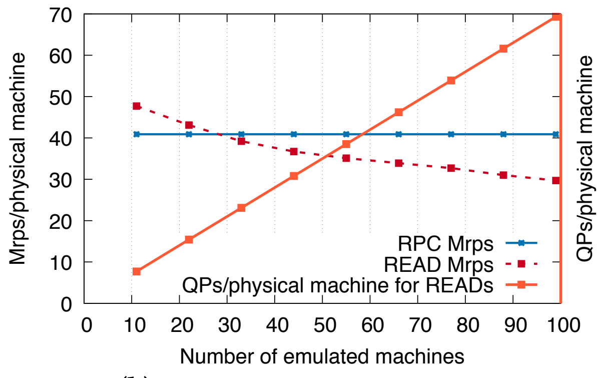
5.2.3.2 On medium-sized clusters

Measuring the impact of one-sided RDMA's poor scalability requires more nodes. As the CIB cluster has only 11 physical machines, we emulate the effect of a larger cluster by creating as many connections on each machine as would be used in the larger cluster. With N physical nodes, we emulate clusters of $N * M$ nodes for different values of M. Instead of creating N QPs, each worker thread creates $N * M$ QPs, and connects them to QPs on other nodes. Note that we only do so for READs because for FaSST's RPCs, the number of local QPs does not depend on the number of machines in the cluster.

Figure 5.2 compares READ and RPC throughput for increasing emulated cluster sizes. We use 32 B READs and RPC requests and responses. Note that the peak READ throughput in this graph is lower than Figure 5.1 that used 6 nodes. This is because NIC cache misses occur with as few as 11 nodes. On CX3, READ throughput drops to 24% of its peak with as few as 22 emulated nodes. On CIB, READs lose their throughput advantage over RPCs on clusters with 33 or more



(a) CX3 cluster (ConnectX-3 NICs)



(b) CIB cluster (Connect-IB NICs)

Figure 5.2: Comparison of FaSST RPC and READ throughput, and the number of QPs used for READs with increasing *emulated* cluster size.

nodes. The decline with Connect-IB NICs is more gradual than with ConnectX-3 NICs. This may be due to a larger cache or better cache miss pipelining [35] in the Connect-IB NIC.

We have repeated the scalability experiment above with the latest RDMA NICs (ConnectX-5) available at time of writing, and achieved similar results (Figure 3.3).

Sharing QPs among worker threads reduces the number of connections that the NIC must cache, but doing so drastically reduces the CPU efficiency of one-sided RDMA. We implement QP sharing similar to Dragojević et al. [38]: we create several sets of N QPs, where each set is connected to the N machines.

We measure the loss in CPU efficiency as follows. We use one server machine that creates a tuneable number of QPs and connects them to QPs spread across 5 client machines (this is large enough to prevent the clients from becoming a bottleneck). We run a tuneable number of worker threads on the server that share these QPs, issuing READs on QPs chosen uniformly at random.

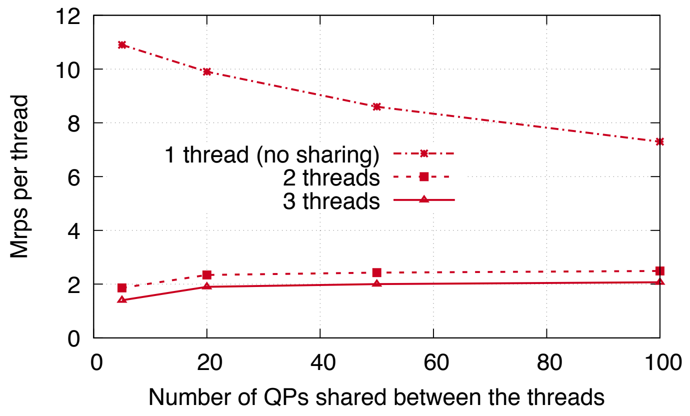


Figure 5.3: Per-thread READ throughput with QP sharing (CIB)

We choose the number of QPs and threads per set based on a large hypothetical cluster with 100 nodes and CIB's CPUs and NICs. A Connect-IB NIC supports ≈ 400 QPs before READ throughput drops below RPC throughput (Figure 5.2). In this 100-node cluster, the 400 QPs are used to create four sets of 100 connections (QPs) to remote machines. CIB's CPUs have 14 cores, so sets of 3–4 threads share a QP set.

Figure 5.3 shows per-thread throughput in this experiment. For brevity, we only show results on CIB; the loss in CPU efficiency is comparable on CX3. The hypothetical configuration above requires sharing 100 QPs among at least three threads; we also show other configurations that may be relevant for other NICs and cluster sizes. With one thread, there is no sharing of QPs and throughput is high—up to 10.9 Mrps. Throughput with QP sharing between three threads, however, is 5.4x lower (2 Mrps).

This observation leads to an important question: If the increase in CPU utilization at the local CPU due to QP sharing is accounted for, do one-sided READs use fewer *cluster-wide* CPU cycles than FaSST's RPCs that do not require QP sharing? We show in Section 5.3 that the answer is no. FaSST's RPCs provide 3.4–4.3 Mrps per core on CIB—1.7–2.15x higher than READs with QP sharing between three threads. Note that in our symmetric setting, each core runs both client and server code. Therefore, READs use cluster CPU cycles at only the client, whereas RPCs use them at both the client and the server. However, RPCs consume fewer *overall* CPU cycles.

5.2.4 Reliability considerations

Unreliable transports do not provide reliable packet delivery, which can introduce programming complexity and/or have performance implications (e.g., increased CPU use), since reliability mechanisms such as timeouts and retransmissions must be implemented in the software RPC layer or application. In the next chapter on eRPC, we show that these mechanisms are cheap to implement in software.

In FaSST, however, we took a different approach to packet loss, based on two observations. First, we note that transaction processing systems usually include a reconfiguration mechanism to handle node failures. Reconfiguration includes optionally pausing ongoing transactions, informing nodes of the new cluster membership, replaying transaction logs, and re-replicating lost data [39]. In FaSST, we assume a standard reconfiguration mechanism; we have not implemented such a mechanism because FaSST's contribution is not in that space. We expect that, similar to DrTM+R [26], FaRM's recovery protocol [39] can be adapted to FaSST.

The second observation is that in normal operation, packet loss in modern RDMA networks is extremely rare: in our experiments (discussed below), we observed zero packet loss in over 50 PB of data transferred. Packets can be lost during network hardware failures, and corner cases of the link/physical layer reliability protocols. FaSST's RPC layer detects these losses using coarse-grained timeouts maintained by the RPC requester (Section 5.3.3).

Based on these two observations, we believe that an acceptable first solution for handling packet loss in FaSST is to simply restart one of the two FaSST processes that is affected by the lost RPC packet, allowing the reconfiguration mechanism to make the commit decision for the affected transaction. We discuss this in more detail in Section 5.4.1.

5.2.4.1 Stress tests for packet loss

Restarting a process on packet loss requires packet losses to be extremely rare. To quantify packet loss on realistic RDMA networks, we set up an experiment on the CX3 cluster, which is similar to real-world clusters with multiple switches, oversubscription, and sharing. It is a shared cluster with 192 machines arranged in a tree topology with seven leaf and two spine switches, with an oversubscription ratio of 3.5. The network is shared by Emulab users. Our largest experiment used 69 machines connected to five leaf switches.

Threads on these machines use InfiniBand's UD transport to exchange 256 B RPCs. We used 256 B messages to achieve both high network utilization and message rate. Threads send 16 requests to remote threads chosen uniformly at random, and wait for all responses to arrive before starting the next batch. A thread stops making progress if a request or reply packet is lost. Threads routinely output their progress messages to a log file; we manually inspect these files to ensure that all threads are making progress.

We ran the experiment without a packet loss for approximately 46 hours. (We stopped the experiment when a log file exhausted a node's disk capacity.) The experiment generated around 100 trillion RPC packets and 33.2 PB of network data. Including other smaller-scale experiments with 20–22 nodes, our experiments transferred over 50 PB of network data without a lost packet.

While we observed zero packet loss, we detected several reordered packets. Using sequence numbers embedded in the RPC packets, we observed around 1500 reordered packets in 100 trillion packets transferred. Reordering happens due to multi-path in CX3: although there is usually a single deterministic path between each source and destination node, the InfiniBand subnet manager sometimes reconfigures the switch routing tables to use different paths.

5.3 FaSST RPCs

We designed FaSST's RPCs specifically for transaction workloads that use small ($\approx$ 1000 B or smaller) objects and a few tens of keys. Key features of FaSST's RPCs include integration with coroutines for efficient network latency hiding, and optimizations such as Doorbell batching and message coalescing.

5.3.1 Coroutines for network latency hiding

Network latency in modern datacenters is on the order of 10 µs under load, which is much higher than the time spent by our applications in computation and local data store accesses. It is critical to not block a thread while waiting for an RPC reply. Similar to Grappa [117], FaSST uses coroutines (cooperative multitasking) to hide network latency: a coroutine yields after initiating network I/O, allowing other coroutines to do useful work while a request is in flight. Our experiments showed that a small number (≈ 20) of coroutines per thread is sufficient for latency hiding, so FaSST uses standard coroutines from the Boost C++ library instead of Grappa's coroutines, which are optimized for use cases with thousands of coroutines. We measured the CPU overhead to switch between coroutines to be 13–20 ns.

In FaSST, each thread creates one RPC endpoint, which is shared by the coroutines spawned by the thread. One coroutine serves as the master; the remaining are workers. Worker coroutines only run application logic and issue RPC requests to remote machines, where they are processed by the master coroutine of the thread handling the request. The master coroutine polls the network to identify any newly-arrived request or response packets. The master computes and sends responses for request packets. It buffers response packets received for each worker until all needed responses are available, at which time it invokes the worker.

5.3.2 RPC interface and optimizations

A worker coroutine operates on batches of $b \geq 1$ requests, based on what the application logic allows. The worker begins by first creating new requests without performing network I/O. For each request, it specifies the request type (e.g., access a particular database table, transaction logging, etc.), and the ID of destination machine. After creating a batch of requests, the worker invokes an RPC function to send the request messages. Note that an RPC request specifies the destination machine, not the destination thread; FaSST chooses the destination thread as the local thread's ID–based peer on the destination machine. Restricting RPC communication to between thread peers improves FaSST's scalability by reducing the number of coroutines that can send requests to a thread (Section 5.3.4). In the next chapter, we show how eRPC leverages NIC hardware features to provide even higher scalability, thereby allowing scalable thread-level all-to-all communication (Section 6.3.1).

Request batching. Operating on batches of requests has several advantages. First, it allows Doorbell batching, reducing the number of Doorbells per batch from b to 1 (Guideline 3.6.2). Second, it allows the RPC layer to coalesce messages sent to the same destination machine. This is particularly useful for multi-key transactions that access multiple tables with same primary key, e.g., in the SmallBank benchmark (Section 5.5). Since our transaction layer partitions tables by a hash of the primary key, the table access requests are sent in the same packet. Third, batching reduces coroutine switching overhead: the master yields to a worker only after receiving responses for all b requests, reducing switching overhead by a factor of b.

Response batching. Similar to request batching, FaSST also uses batching for responses. When the master coroutine polls the NIC for new packets, it typically receives more than one packet. On receiving a batch of B request packets, it invokes the request handler for each request, and assembles a batch of B response packets. These responses are sent using one Doorbell. Note that the master does not wait for a batch of packets to accumulate before sending responses to avoid adding latency.

Cheap RECV descriptor posting. FaSST's RPCs use two-sided verbs, so we must post RECVs on RECV queues before an incoming SEND arrives. Posting RECVs requires creating descriptors in the host-memory RECV queue, and updating the queue's host-memory head pointer. The CPU need not initiate PCIe MMIOs because the NIC fetches the descriptors using DMA reads. In FaSST, we populate the RECV queue with descriptors once during initialization, after which the CPU does not modify the descriptors. New RECVs re-use descriptors in a circular fashion, and require a single write to the cached head pointer for posting. This optimization requires modifying the NIC's device driver, but it saves CPU cycles.

It is interesting to note that in FaSST, the NIC's RECV descriptor DMA reads are redundant, since the descriptors never change after initialization. In the next chapter, we show how eRPC leverages the multi-packet receive queue of modern NICs to eliminate these DMA reads (Section 6.3.1).

5.3.3 Detecting packet loss

The master coroutine at each thread detects packet loss for RPCs issued by its worker coroutines. The master tracks the progress of each worker by counting the number of responses received for the worker. A worker's progress counter stagnates if and only if one of the worker's RPC packets (either the request or the response) is lost: If a packet is lost, the master never receives all responses for the worker; it never invokes the worker again, preventing it from issuing new requests and receiving more responses. If no packet is lost, the master eventually receives all responses for the worker. The worker gets invoked and issues new requests—we do not allow workers to yield to the master without issuing RPC requests.

If the counter for a worker does not change for `timeout` seconds, the master assumes that the worker suffered a packet loss. On suspecting a loss, the master kills the FaSST process on its machine (Section 5.4.1). Note that, before it is detected, a packet loss affects only the progress of one worker, i.e., other workers can successfully commit transactions until the loss is detected. This allows us to use a large value for `timeout` without affecting FaSST's availability. We currently set `timeout` to one second. In our experiments with 50+ nodes, we did not observe a false positive with this timeout value. We observed false positives with significantly smaller timeout values such as 100 ms. This can happen, for instance, if the thread handling the RPC response gets preempted [39].

5.3.4 Limitations of FaSST RPCs

We designed FaSST's RPC subsystem as a proof-of-concept to show that RPCs can outperform one-sided RDMA for transactions. As a result, FaSST RPCs do not provide all features of a general-purpose RPC library. In eRPC, we show that all these missing features can be supported with little CPU overhead.

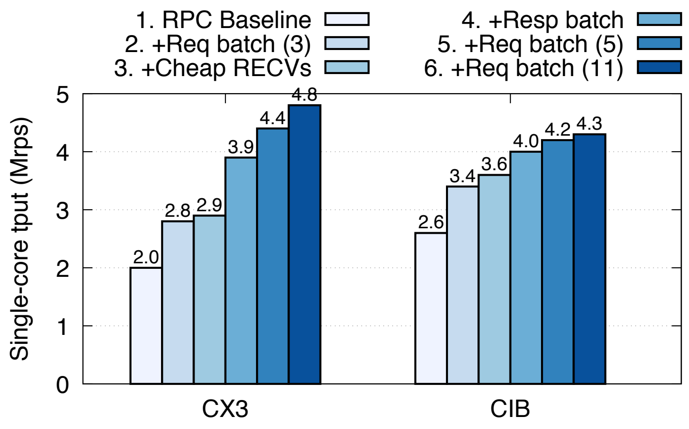


Figure 5.4: Single-core RPC throughput as optimizations 2–6 are added

FaSST RPCs do not support workloads that require messages larger than the network's MTU (4 kB on our InfiniBand network). These workloads are likely to be bottlenecked by network bandwidth with both RPC- and one-sided RDMA-based designs, achieving similar performance. In addition, FaSST RPCs do not provide packet retransmission or congestion control.

FaSST also restricts each coroutine to one message per destination machine per batch; the message, however, can contain multiple coalesced requests. This restriction is required to keep the RECV queues small so that they can be cached by the NIC. Consider a cluster with N nodes, T threads per node, and c coroutines per thread. For a given thread, there are N peer threads, and $N * c$ coroutines that can send requests to it. At any time, each thread must provision as many RECVs in its RECV queue as the number of requests that can be sent to it. Allowing each coroutine m messages per destination machine requires maintaining $(N * c * m)$ RECVs per RECV queue. A fairly large cluster with $N = 100$, $c = 20$, and $T = 14$ requires 14 RECV queues of size $2000 * m$ at each machine. $m = 1$ was sufficient for our workloads and worked well in our experiments, but significantly larger values of m reduce RPC performance by causing NIC cache thrashing.

One possible method to increase scalability by reducing RECV queue size can work as follows: we reduce the number of requests allowed from a local thread to a particular remote machine from c to some smaller number; a coroutine yields if its thread's budget for a remote machine is temporarily exhausted. This will work well with large clusters and workloads without high skew, where the probability of multiple coroutines sending requests to the same remote machine is small.

5.3.5 Single-core RPC performance

We showed in Section 5.2.3 that FaSST RPCs provide good per-NIC throughput. We now show that they also provide good single-core throughput. To measure per-core throughput, we run one thread per machine, 20 coroutines per thread, and use 32 B RPCs. We use all 11 available machines on CIB; we use 11 machines on CX3 for comparison. We evaluate RPC performance

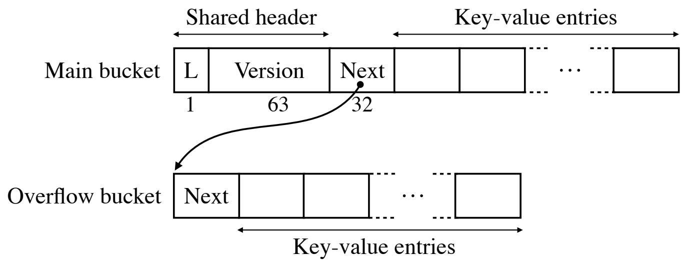


Figure 5.5: Layout of main and overflow buckets in our MICA-based hash table

with multiple request batch sizes. To prevent request coalescing by our RPC layer, we choose a different machine for each request in the batch.

For our RPC baseline, we use a request batch size of one, and disable response batching. We then enable the request batching, cheap RECV posting, and response batching optimizations in succession. Figure 5.4 shows the results from this experiment. Recall that the Doorbell batching–based optimizations do not apply to READs. For brevity, we discuss only CIB here.

Even without any optimizations, FaSST RPCs are more CPU-efficient than READs with connection sharing: our baseline achieves 2.6 Mrps, whereas READs achieve up to 2 Mrps with connection sharing between 3 or more threads (Figure 5.3). With a request batch size of three and all optimizations enabled, FaSST RPCs achieve 4 Mrps—2x higher than READs. Peak RPC throughput with one request per batch is 3.4 Mrps (not shown in the Figure).

With 11 requests per batch, FaSST RPCs achieve 4.3 Mrps. At this request rate, each CPU core issues 17.2 million network operations per second on average: 4.3 million SENDs each for requests and responses, and 8.6 million for their RECVs. This large advantage over one-sided READs (which achieve 2 million verbs per second) arises from FaSST's use of datagram transport, which allows exclusive access to QPs and Doorbell batching.

Comparison with FaRM RPCs. FaRM's RPCs achieve up to 5 Mrps with one ConnectX-3 NIC and 16 CPU cores [38]. Their throughput does not increase noticeably on adding another ConnectX-3 NIC [39], so we expect them to provide $\approx$ 5 Mrps with a Connect-IB NIC. FaSST RPCs can achieve 34.9–40.9 Mrps (Figure 5.1), i.e., up to 8x higher throughput per NIC. FaRM's RPCs achieve $5/16 = 0.31$ Mrps per core; FaSST can achieve 3.4–4.3 Mrps per core depending on the request batch size (up to 13.9x higher).

5.4 Transactions

FaSST provides transactions with serializability and durability on partitioned distributed data stores. FaSST's data stores map 8 B keys to opaque application-level objects. Each key is associated with an 8 B header, consisting of a lock bit, and a 63-bit version number. The header is used for concurrency control and for ordering commit log records during recovery. Several keys can map to the same header.

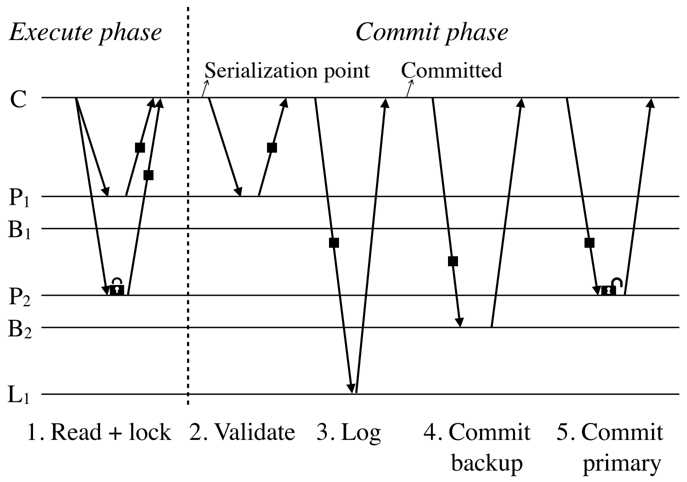


Figure 5.6: FaSST's transaction protocol with tolerance for one node failure. P_1 and P_2 are primaries and B_1 and B_2 are their backups. C is the transaction coordinator, whose log replica is L_1. The solid boxes denote messages containing application-level objects. The transaction reads one key from P_1 and P_2, and updates the key on P_2.

We have implemented transactions for an unordered key-value store based on MICA [97]. The key-value store uses a hash table composed of associative buckets (Figure 5.5) with multiple (7–15) slots to store key-value items. Each key maps to a *main* bucket. If the number of keys mapping to a main bucket exceeds the bucket capacity, we dynamically link the main bucket to a chain of *overflow* buckets. The main bucket maintains the header for all keys stored in a main bucket and its linked overflow buckets.

In FaSST, worker coroutines run the transaction logic and act as transaction coordinators. FaSST's transaction protocol is inspired by FaRM's, with some modifications for simplicity. FaSST uses optimistic concurrency control and two-phase commit for distributed atomic commit, and primary-backup replication to support high availability. We use the Coordinator Log [142] variant of two-phase commit for its simplicity. Figure 5.6 summarizes FaSST's transaction protocol. We discuss the protocol's phases in detail below. FaSST RPCs are used to send all messages. We denote the set of keys read and written by the transaction by R (read set) and W (write set) respectively. We assume that the transaction first reads the keys it writes, i.e., $W \subseteq R$.

1. Read and lock. The transaction coordinator begins execution by reading the header and value of keys from their primaries. For a key in W, the coordinator also requests the primary to lock the key's header. The flexibility of RPCs allows us to read and lock keys in a single round trip. Achieving this with one-sided RDMA requires two round trips: one to lock the key using an RDMA atomic operation, and one to read its value [26]. If any key in R or W is already locked, the coordinator aborts the transaction by sending unlock RPCs for successfully locked keys.

2. Validate. After locking the write set, the coordinator checks the versions of its read set by requesting the versions of R again. If any key is locked or its version has changed since the first phase, the coordinator aborts the transaction.

3. Log. If validation succeeds, the transaction can commit. To commit a transaction, the coordinator replicates its commit log record at $f+1$ log replicas so that the transaction's commit decision survives f failures. The coordinator's host machine is always a log replica, so we send f RPCs to remote log replicas. The commit log record contains W's key-value items and their fetched versions.

4. Commit backup. If logging succeeds, the coordinator sends update RPCs to backups of W. It waits for an ACK from each backup before sending updates to the primaries. This wait ensures that backups process updates for a bucket in the same order as the primary. This ordering is not required in FaRM, which can drop out-of-order bucket updates as each update contains the contents of the entire bucket. FaSST's updates contain only one key-value item and are therefore smaller, but cannot be dropped.

5. Commit primary. After receiving all backup ACKs, the coordinator sends update RPCs to the primaries of W. On receiving an update, a primary updates the key's value, increments its version, and unlocks the key.

Similar to existing systems [39, 154], FaSST omits validation and subsequent phases for single-key read-only transactions.

5.4.1 Handling failures and packet loss

Our FaSST implementation provides serializability and durability, but not high availability. Similar to prior single-node transaction systems [149], we have implemented the normal case datapath (logging and replication) to the extent that fast recovery is possible, but we have not implemented the actual logic to recover from a machine failure. We assume that FaRM's mechanisms to detect and recover from machine failures, such as leases, cluster membership reconfiguration, log replay, and re-replication of lost data can be adapted to FaSST; we discuss how packet losses can be handled below. Note that our implementation is insensitive to packet reordering since each RPC message is smaller than the network's MTU.

We convert a packet loss to a machine failure by killing the FaSST process on the machine that detects a lost RPC packet (Section 5.3.3). The transaction affected by the lost packet will not make progress until the killed FaSST process is detected (e.g., via leases); then the transaction's commit/abort decision will be handled by the recovery mechanism. This basic scheme can be improved (e.g., the victim node can be re-used to avoid data re-replication since it need not reboot), but that is not the focus of our work.

In Section 5.2.4, we measured the packet loss rate of our network at less than one in 50 PB of data. Since we did not actually lose a packet, the real loss rate may be much lower, but we use this upper-bound rate for a ballpark availability calculation. In a 100-node cluster where each node is equipped with 2x 56 Gbps InfiniBand and transfers data at full-duplex bandwidth, it will take approximately 5 hours to transfer 50 PB. Therefore, packet losses will translate to less than five machine failures per day. Assuming that each failure causes 50 ms of downtime as in FaRM [39], FaSST will achieve five-nines of availability.

5.4.2 Implementation

We now discuss details of FaSST's transaction implementation. FaSST provides transactions on 8 B keys and opaque objects up to 4060 B in size. The value size is limited by our network's MTU (4096 B) and the commit record header overhead (36 B). To extend a single-node data store for distributed transactions, a FaSST user writes RPC request handlers for pre-defined key-value requests (e.g., get, lock, put, and delete). This may require changes to the single-node data store, such as supporting version numbers. The user registers database tables and their respective handlers with the RPC layer by assigning each table a unique RPC request type; the RPC subsystem invokes a table's handler on receiving a request with its table type.

The data store must support concurrent local read and write access from all threads in a node. An alternate design is to create exclusive data store partitions per thread, instead of per-machine partitions as in FaSST. As shown in prior work [97], this alternate design is faster for local data store access since threads need not use local concurrency control (e.g., local locks) to access their exclusive partition. However, when used for distributed transactions, it requires the RPC subsystem to support all-to-all communication between threads, which reduces scalability by amplifying the required RECV queue size (Section 5.3.4). We chose to sacrifice higher CPU efficiency on small clusters for a more pressing need: cluster-level scalability.

5.4.2.1 Transaction API

The user writes application-level transaction logic in a worker coroutine using the following API.

`AddToReadSet(K, *V)` and **`AddToWriteSet(K, *V, mode)`** enqueue key `K` to be fetched for reading or writing, respectively. For write set keys, the write mode is either insert, update, or delete. After the coroutine returns from `Execute` (see below) the value for key `K` is available in the buffer `V`. At this point, the application's transaction logic can modify `V` for write set keys to the value it wishes to commit.

`Execute()` sends the execute phase RPCs of the transaction protocol. Calling `Execute` suspends the worker coroutine until all responses are available. Note that the `AddToReadSet` and `AddToWriteSet` functions above do not generate network messages immediately: requests are buffered until the worker coroutine calls `Execute`. This allows the RPC layer to send all requests in with one Doorbell, and coalesce requests sent to the same remote machine. Applications can call `Execute` multiple times in one transaction after adding more keys. This allows transactions to choose new keys based on previously fetched keys.

`Execute` fails if a read or write set key is locked. In this case, the transaction layer returns failure to the application, which then must call `Abort`.

	Nodes	NICs	CPUs (cores used, GHz)
FaSST (CX3)	50	1	1x E5-2450 (8, 2.1 GHz)
FaRM [39]	90	2	2x E5-2650 (16, 2.0 GHz)
DrTM+R [26]	6	1	1x E5-2450-v3 (8, 2.3 GHz)

Table 5.1: Comparison of clusters used to compare published numbers. The NIC count is the number of ConnectX-3 NICs. All CPUs are Intel Xeon CPUs. DrTM+R's CPU has 10 cores but their experiments use only 8 cores.

`Commit()` runs the commit protocol, including validation, logging and replication, and returns the commit status. **`Abort()`** sends unlock messages for write set keys.

5.5 Evaluation

We evaluate FaSST using three benchmarks: an object store, a read-mostly OLTP benchmark, and a write-intensive OLTP benchmark. We use the simple object store benchmark to measure the effect of two factors that affect FaSST's performance: multi-key transactions and the write-intensiveness of the workload. All benchmarks include three-way logging and replication, and use 14 threads per machine. We use at most 19 worker coroutines per thread to limit the RECV queue size required on a (hypothetical) 100-node cluster to 2048 RECVs (Section 5.3.4). Using the next available RECV queue size with 4096 RECVs can cause NIC cache misses for some workloads.

We use the other two benchmarks to compare against two recent RDMA-based transaction systems, FaRM [39] and DrTM+R [26]. Unfortunately, we are unable do a direct comparison by running these systems on our clusters. FaRM is not open-source, and DrTM+R depends on Intel's Restricted Transactional Memory (RTM), which our evaluation clusters (CX3 and CIB) do not support.[2]

For a comparison against published numbers, we use the CX3 cluster which has less powerful hardware (NIC and/or CPU) than used in FaRM and DrTM+R; Table 5.1 shows the differences in hardware. We believe that the large performance difference between FaSST and other systems (e.g., 1.87x higher than FaRM on TATP with half the hardware resources; 1.68x higher than DrTM+R on SmallBank without locality assumptions) offsets performance variations due to system and implementation details. We also use the CIB cluster in our evaluation to show that FaSST can scale up to more powerful hardware.

TATP is an OLTP benchmark that simulates a telecommunication provider's database. It consists of four database tables with small key-value pairs up to 48 B in size. TATP is read-intensive: 70% of TATP transactions read a single key, 10% of transactions read 1–4 keys, and

[2]Intel's RTM is disabled by default on CIB's Haswell processors due to a hardware bug that can cause unpredictable behavior. It can be re-enabled by setting model-specific registers [154], but we were not permitted to do so on the CIB cluster.

the remaining 20% of transactions modify keys. TATP's read-intensiveness and small key-value size makes it well-suited to FaRM's design goal of exploiting remote CPU bypass: 80% of TATP transactions are read-only and do not involve the remote CPU. Although TATP tables can be partitioned intelligently to improve locality, we do not do so (similar to FaRM).

SmallBank is a simple OLTP benchmark that simulates bank account transactions. SmallBank is write-intensive: 85% of transactions update a key. Our implementation of SmallBank does not assume data locality. In DrTM+R, however, single-account transactions (comprising 60% of the workload) are initiated on the server hosting the key. Similarly, only a small fraction (< 10%) of transactions that access two accounts access accounts on different machines. These assumptions make the workload well-suited to DrTM+R's design goal of optimizing local transactions by using hardware transactional memory, but they save messages during transaction execution and commit. We do not make either of these assumptions and use randomly chosen accounts in all transactions.

Although the TPC-C benchmark [148] is a popular choice for evaluating transaction systems, we chose not to include it in our benchmarks for two reasons. First, TPC-C has a high degree of locality: only around 10% of transactions (1% of keys) access remote partitions. The speed of local transactions and data access, which our work does not focus on, has a large impact on TPC-C performance. Second, comparing performance across TPC-C implementations is difficult. This is due to differences in data structures (e.g., using hash tables instead of B-Trees for some tables), interaction of the benchmark with system optimizations (e.g., FaRM and DrTM+R use caching to reduce READs, but do not specify cache hit rates), and contention level. For example, DrTM+R uses one TPC-C "warehouse" per thread whereas FaRM uses ≈ 7, which may reduce contention.

5.5.1 Performance for an object store

We create an object store with small objects with 8 B keys and 40 B values. We scale the database by using 1 million keys per thread in the cluster. We use workloads with different read and write set sizes to evaluate different aspects of FaSST. We denote an object store workload in which transactions read r keys, and update w of these keys (on average) by $O(r, w)$; we use $O(1, 0)$, $O(4, 0)$, and $O(4, 2)$ to evaluate single-key read-only transactions, multi-key read-only transactions, and multi-key read-write transactions. All workloads choose keys uniformly at random; to avoid RPC-level coalescing, we choose keys such that their primaries are on different machines. Figure 5.7 shows FaSST's performance on the object store workloads.

5.5.1.1 Single-key read-only transactions

With $O(1, 0)$ FaSST achieves 11.0 million transactions per second (Mtps) per machine on CX3. FaSST is bottlenecked by the ConnectX-3 NIC: this throughput corresponds to 11.0 million RPC requests per second (Mrps), which is 96.5% of the NIC's maximum RPC throughput in this scenario.

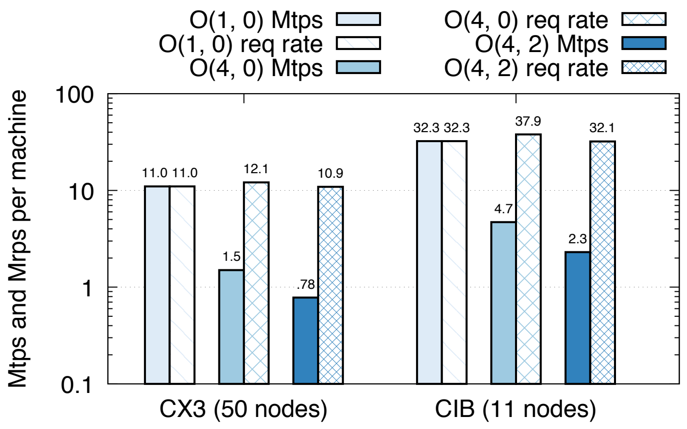


Figure 5.7: Object store performance. The solid and patterned bars show transaction throughput and RPC request rate, respectively. The Y axis is in log scale.

On CIB, FaSST achieves 32.3 Mtps/machine and is CPU-limited. This is because $O(1, 0)$ does not allow Doorbell batching for requests, leading to low per-core throughput. Although CIB's CPUs can saturate the NIC without request Doorbell batching for an RPC microbenchmark that requires little computation (Section 5.2.3.1), they cannot do so for $O(1, 0)$ which requires key-value store accesses.

Comparison. FaRM [39] reports performance for the $O(1, 0)$ workload. FaRM uses larger, 16 B keys and 32 B values. Our FaSST implementation currently supports only 8 B keys, but we use larger, 40 B values to keep the key-value item size identical. Using 16 B keys is unlikely to change our results.[3]

FaRM achieves 8.77 Mtps/machine on a 90-node cluster with $O(1, 0)$. It does not saturate its two ConnectX-3 NICs and is instead bottlenecked by its 16 CPU cores. FaSST achieves 1.25x higher per-machine throughput with 50 nodes on CX3, which has close to half of FaRM's hardware resources per node (Table 5.1). *Although $O(1, 0)$ is well-suited to FaRM's design goal of remote CPU bypass (i.e., no transaction involves the remote CPU), FaRM performs worse than FaSST.* Note that with FaRM's hardware—two ConnectX-3 NICs and 16 cores—FaSST will deliver higher performance; based on our CIB results, we expect FaSST to saturate the two ConnectX-3 NICs and outperform FaRM by 2.5x.

5.5.1.2 Multi-key transactions

With multi-key transactions, FaSST reduces per-message CPU use by using Doorbell batching for requests. With $O(4, 0)$, FaSST achieves 1.5 and 4.7 Mtps/machine on CX3 and CIB, respectively. (The decrease in Mtps from $O(1, 0)$ is because the transactions are larger.) Similar to $O(1, 0)$, FaSST is NIC-limited on CX3. On CIB, however, although FaSST is CPU-limited

[3]On a single node, FaSST's data store (MICA) delivers similar GET throughput (within 3%) for these two key-value size configurations. Throughput is *higher* with 16 B keys, which could be because MICA's hash function uses fewer cycles.

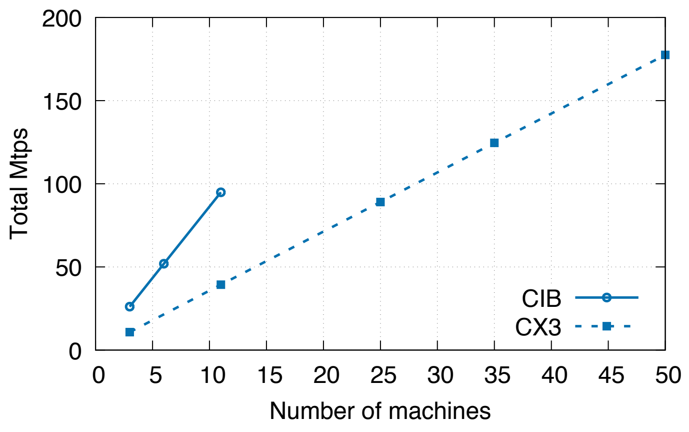


Figure 5.8: TATP throughput

with $O(1, 0)$, it becomes NIC-limited with $O(4, 0)$. With $O(4, 0)$ on CIB, each machine generates 37.9 Mrps on average, which matches the peak RPC throughput achievable with a request batch size of 4.

With multi-key read-write transactions in $O(4, 2)$, FaSST achieves 0.78 and 2.3 Mtps/machine on CX3 and CIB, respectively. FaSST is NIC-limited on CX3. On CIB, the bottleneck shifts to CPU again because key-value store inserts into the replicas' data stores are slower than lookups.

Comparison. FaRM does not report object store results for multi-key transactions. However, as FaRM's connected transport does not benefit from Doorbell batching, we expect the gap between FaSST's and FaRM's performance to increase. For example, while FaSST's RPC request rate increases from 32.3 Mrps with $O(1, 0)$ to 37.9 Mrps with $O(4, 0)$, the change in FaRM's READ rate is likely to be negligible.

5.5.2 Performance on the TATP benchmark

We scale the TATP database size by using one million TATP "subscribers" per machine in the cluster. We use all CPU cores on each cluster and increase the number of machines to measure the effect of scaling. Figure 5.8 shows the throughput on our clusters. On CX3, FaSST achieves 3.6 Mtps/machine with 3 nodes (the minimum required for 3-way replication), and 3.55 Mtps/machine with 50 nodes. On CIB, FaSST's throughput increases to 8.7 Mtps/machine with 3–11 nodes. In both cases, FaSST's throughput scales linearly with cluster size.

Comparison. FaRM [39] reports 1.55 Mtps/machine for TATP on a 90-node cluster. With a smaller 50-node cluster, however, FaRM achieves higher throughput ($\approx$ 1.9 Mtps/machine) [1]. On 50 nodes on CX3, FaSST's throughput is 87% higher. Compared to $O(1, 0)$, the TATP performance difference between FaSST and FaRM is higher. TATP's write transactions require using FaRM's RPCs, which deliver 4x lower throughput than FaRM's one-sided READs, and up to 8x lower throughput than FaSST's RPCs (Section 5.3.5).

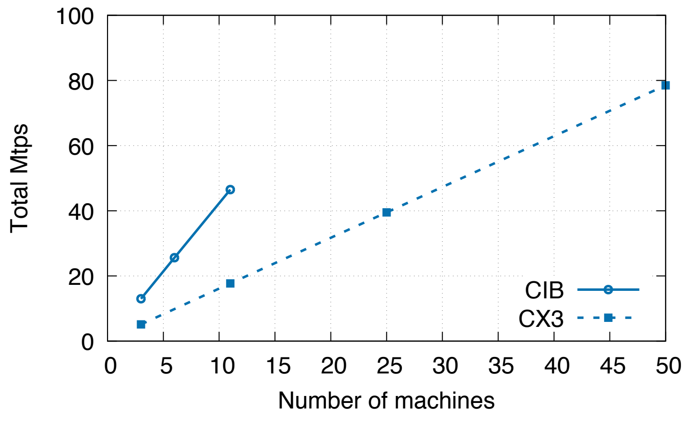


Figure 5.9: FaSST's SmallBank throughput

5.5.3 Performance on the SmallBank benchmark

To scale the SmallBank database, we use 100,000 bank accounts per thread. 4% of the total accounts are accessed by 90% of transactions. (Despite the skew, the workload does not have significant contention due to the large number of threads, and therefore "bank accounts" in the workload/cluster.) This configuration is the same as in DrTM [154]. Figure 5.9 shows FaSST's performance on our clusters. FaSST achieves 1.57–1.71 Mtps/machine on CX3, and 4.2–4.3 Mtps/machine on CIB, and scales linearly with cluster size.

Comparison. DrTM+R [26] achieves 0.93 Mtps/machine on a cluster similar to CX3 (Table 5.1), but with more powerful CPUs. FaSST outperforms it by over 1.68x on CX3, and over 4.5x on CIB. DrTM+R's lower performance comes from three factors. First, DrTM+R's use of RDMA's atomic operations lead to a fundamentally slower protocol. For example, excluding logging and replication, for a write-set key, DrTM+R uses four separate messages to read, lock, update, and unlock the key; FaSST uses only two messages. Second, as discussed in Guideline 3.7.2, RDMA's atomic operations perform poorly (up to 10x worse than READs) on the ConnectX-3 NICs used in DrTM+R; evaluation on Connect-IB NICs may yield better performance, but is unlikely to outperform FaSST because of the more expensive protocol. Third, DrTM+R does not use connection sharing, so their reported performance may be affected by NIC cache misses.

5.5.4 Transaction latency on TATP

For brevity, we discuss FaSST's latency only for TATP on CIB. Figure 5.10 shows FaSST's median and 99^{th} percentile latency for successfully committed TATP transactions. To plot a throughput-latency graph, we vary the request load by increasing the number of worker coroutines per thread from 1 to 19; each machine runs 14 threads throughout. With one worker coroutine per thread, the total transaction throughput is 19.7 Mtps with 2.8 µs median latency and 21.8 µs 99^{th} percentile latency. Since over 50% of committed transactions in TATP are single-key reads, FaSST's median latency at low load is close to the network's RTT. This shows that our batching

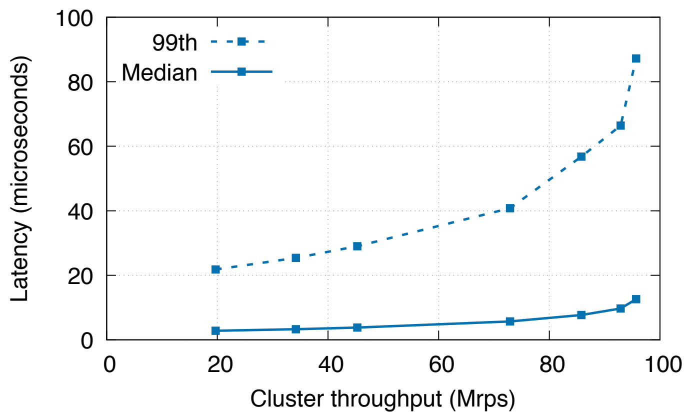


Figure 5.10: FaSST's TATP latency on CIB

optimizations do not add noticeable latency. With 19 worker coroutines per thread, cluster throughput increases to 95.7 Mtps, and median and 99th percentile latency increase to 12.6 µs and 87.2 µs, respectively.

5.6 Discussion

5.6.1 Dynamically Connected Transport

A key limitation of one-sided RDMA on current commodity hardware is its low scalability. This limitation itself comes from the fundamentally connection-oriented nature of the Virtual Interface Architecture (Section 3.5.2). One notable attempt to work around the need for connections is InfiniBand's Dynamically Connected Transport (DCT). DCT preserves its core connection-oriented design, but dynamically creates and destroys one-to-one connections. This provides software the illusion of using one QP to communicate with multiple remote machines, but at a prohibitively large performance cost for our workloads: DCT requires three additional network messages when the target machine of a DCT queue pair changes: a disconnect packet to the current machine, and a two-way handshake with the next machine to establish a connection [30]. In a high fanout workload such as distributed OLTP, this increases the number of packets associated with each RDMA request by around 1.5x, reducing performance.

A detailed evaluation of DCT on CIB is available in FaSST's source code repository. Here, we discuss DCT's performance in the READ rate benchmark used in Section 5.2.3.1. We use six machines and 14 threads per machine, which issue 32 B READs to machines chosen uniformly at random. We vary the number of outstanding READs per thread, and the number of DCT QPs used by each thread. (Using only one DCT QP per thread limits its throughput to approximately one operation per multiple RTTs, since a QP cannot be used to READ from multiple machines concurrently. Using too many DCT QPs causes cache NIC misses.) We achieve only up to 22.9 Mrps per machine—55.3% lower than the 51.2 Mrps achievable with standard READs over the RC transport (Figure 5.1).

5.6.2 Comparison with hybrid RPC-RDMA approaches

FaSST takes an extreme approach took by using only RPCs, demonstrating that high-performance transactions do not necessarily require CPU bypass, and that a design using optimized RPCs can provide better performance. It might seem that if scalable one-sided RDMA becomes commonly available in the future, the highest-performing design will likely be hybrid of RPCs and remote bypass, with RPCs used for accessing data structures during transaction execution, and scalable one-sided writes used for logging and replication during transaction commit. Wei et al. [155] design such a hybrid transaction processing system called RTX. However, FaSST's RPC-only design outperforms RTX's hybrid design.

In the evaluation results reported by Wei et al. [155], RTX outperforms a highly-modified version of FaSST that uses RTX's transaction protocol. RTX's protocol was designed with one-sided RDMA in mind; it is less efficient than FaSST's original protocol, which benefits from the general-purpose computation ability of CPUs. For example, FaSST's original protocol can read and lock a remote object in a transaction's write set in one RPC, whereas RTX's version of FaSST does so in two RPCs, emulating the two one-sided RDMA operations (RDMA read and atomic compare-and-swap) used in RTX's protocol. FaSST's original design and implementation outperforms RTX. On the SmallBank benchmark, FaSST achieves 4.2 Mtps per machine on CIB, whereas RTX achieves 2.85 Mtps per machine with similar CPU and NIC resources.[4]

In addition, remote-bypassing designs will have higher software complexity than RPC-only designs. For example, writing durably to remote non-volatile memory with RDMA writes is challenging [104].

5.6.3 Advanced one-sided RDMA

Future NICs may provide advanced one-sided RDMA operations such as multi-address atomic operations, and B-Tree traversals [132]. Both of these operations require multiple PCIe round trips, and will face similar flexibility and performance problems as one-sided RDMA (but over the PCIe bus) if used for high-performance distributed transactions. On the other hand, we believe that "CPU onload" networks such as Intel's 100 Gbps OmniPath [19] are well-suited for transactions. These networks provide fast messaging over a reliable link layer, but not one-sided RDMA, and are therefore cheaper than "NIC offload" networks such as Mellanox's InfiniBand. FaSST requires only messaging, so we expect our design to work well over OmniPath.

[4]RTX uses platforms with two CPUs and two NICs each. Because their evaluation isolates processes to one socket, we regard each platform as two machines, each of which has a similarly-powerful CPU and NIC as our CIB cluster.

5.7 Related work on distributed transactions

Like FaSST, FaRM uses primary-backup replication and optimistic concurrency control for transactions. FaRM's design (unlike FaSST) is specialized to work with their desire to use one-sided RDMA verbs. FaRM also provides fast failure detection and recovery, and a sophisticated programming model, which was not a goal of our work on FaSST. Several projects use one-sided RDMA atomics for transactions [17, 26, 154]. Though an attractive primitive, RDMA atomics can reduce performance because of in-NIC lock contention (Guideline 3.7.2), use of connected QPs, and additional round trips compared to an RPC-based approach (e.g., separate messages are needed to read and lock a key). Calvin [145] uses conventional networking without kernel bypass, and is designed around avoiding distributed commit. Designs that use fast networks, however, can use traditional distributed commit protocols to achieve high performance [39, 154].

5.8 Conclusion

FaSST is a high-performance, scalable, distributed in-memory transaction processing system that provides serializability and durability. FaSST achieves its performance using FaSST RPCs, a new RPC design tailored to the properties of modern datacenter networks. It rejects RDMA's CPU bypass feature to keep its communication overhead low and its system design fast, flexible, scalable, and simple. FaSST outperforms recent transactional systems that use one-sided RDMA by 1.68x–1.87x with fewer resources and making fewer workload assumptions. Finally, we provide the first large-scale study of InfiniBand network reliability, demonstrating the rarity of packet loss on such networks.

Our experience with FaSST led to two insights that form the basis of eRPC. First, since RPCs perform well in lossless networks with zero packet loss, they should also perform well most of the time in lossy networks, *if* we can somehow make packet loss rare. Second, since RPCs perform well with small messages and short-running handlers—which are workload characteristics of our target datacenter applications—perhaps we can provide good performance for these *common-case* workloads, while also supporting large messages and long-running handlers in less common cases. Combined, these insights lead to eRPC's end-to-end design that does not depend on in-network features, and eRPC's CPU-efficient support of a large, general-purpose feature set.

"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

Chapter 6

eRPC: A Fast and General-purpose RPC Library

This chapter presents the concluding RPC design of this thesis: eRPC. We build on top of results from previous chapters with a new understanding of packet loss and congestion control in datacenters to create an end-to-end design with a general-purpose feature set, while preserving speed close to HERD and FaSST RPCs. Unlike our prior RPC designs, eRPC does not depend on any in-network features; it works well with only UDP packets over lossy Ethernet without Priority Flow Control. eRPC adds support for a general-purpose feature set that is sufficient to build real applications, including support for congestion control, large messages, long-running request handlers, and node failures. In the past, researchers have hypothesized that supporting these features in software instead of NIC hardware would result in performance substantially lower than FaSST RPCs [40]. We show that with careful design, we can support all these features and still match FaSST's performance.

eRPC has two key insights. First, we optimize for common-case workloads and network conditions, i.e., when messages are small and RPC handlers are short (Section 2.2.4), and the network is congestion-free. Handling large messages, long-running RPC handlers, and network congestion 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, we show that 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.

Our contributions in this chapter are:

1. We describe the design and implementation of a high-performance, general-purpose RPC library for datacenter networks that 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. 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, with well-designed end-host networking software, lossy networks can provide state-of-the-art networking performance. 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 [23, 121] that is used in Intel's distributed object store [36] 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 [76]), FPGAs [72], and RDMA (DARE [129]).

6.1 Understanding packet loss in datacenter networks

The RPC designs in HERD and FaSST rely on a lossless link layer to avoid dealing with packet loss in an end-to-end fashion in software. Unfortunately, network losslessness comes with several drawbacks, including deadlocks, unfairness, and operational complexity (Section 2.1.3.2). eRPC uses new insights about packet loss in datacenter networks to provide high performance without lossy networks: We found that because datacenter switch buffers vastly exceed their network's BDP, restricting each flow to one BDP of outstanding data prevents most packet drops even on lossy networks.

Switch buffers in Ethernet datacenters. The increase in datacenter network 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 CX4 cluster that resembles real datacenters (Table 2.2), the RTT between two nodes connected to different top-of-rack 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 [6].

In contrast to the small BDP (~10 kB), switches have tens of megabytes of buffer. For example, the Mellanox Spectrum switches in CX4 have 12 MB in their dynamic buffer pool [75]. Therefore, the ToR 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 [43, 165]. Zhang et al. [161] have made a similar observation (i.e., buffer $\gg$ 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 [115, 165]. 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 example, 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.

6.2 eRPC overview

This section provides an overview of the interface that eRPC provides to developers, i.e., its API and threading model. While FaSST RPCs provide an interface specialized for transactions in OLTP systems, eRPC provides a general-purpose and more convenient interface. For example, we forego coroutines, and instead use event-loop based concurrency, which is easier to accommodate in existing codebases. And, eRPC provides the ability to run long-running RPC handlers in "worker" threads.

Similar to FaSST, eRPC implements RPCs on top of a transport layer that provides basic unreliable packet I/O, such as UDP over Ethernet, or InfiniBand's Unreliable Datagram transport. It requires a userspace NIC driver for good performance. eRPC's primary contribution is the design and implementation of end-host mechanisms and a network transport (e.g., wire protocol and congestion control) that support the RPC interface.

6.2.1 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 (§ 6.2.2).

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.

6.2.2 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 [124]. Other RPC systems such as FaRM, and FaSST and HERD RPCs, avoid this overhead by running all request handlers directly in dispatch threads. 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.

6.2.3 Evaluation clusters

We evaluate eRPC on the CX3, CX4, and CX5 clusters (Table 2.2), covering both lossy Ethernet and lossless InfiniBand networks. 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, 165], 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.) Because CX4 is shared with other CloudLab users, we use up to 100 nodes out of its 200 nodes. The six switches in CX4 are organized as five ToRs with 40 25 GbE downlinks and five 100 GbE uplinks, for a 2:1 oversubscription.

6.3 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.

6.3.1 Scalability considerations

We chose plain packet I/O instead of RDMA writes to send messages in eRPC. This decision is based on two insights. First, the connection-oriented nature of one-sided RDMA reduces scalability. eRPC replaces NIC-managed connection state with CPU-managed connection state. This is an explicit design choice, based upon fundamental differences between CPU and NIC architectures (Guideline 3.5.2).

Second, RPC systems that use RDMA writes, such as HERD and FaRM's RPCs, have another 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 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 (§ 6.3.3.1). 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 (Section 5.5), 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.

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 [38].

As we discuss in Section 6.3.3.1, eRPC's RQs must have sufficient descriptors for all connected sessions. The footprint of traditional RQs grows with the number of connected sessions that eRPC supports. 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 [111], 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 eight-entry CQs, which makes the contention negligible.

6.3.2 Challenges in zero-copy transmission

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

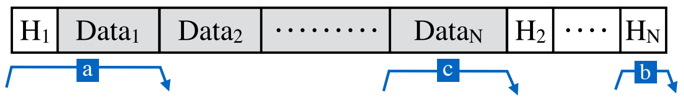


Figure 6.1: Layout of packet headers and data for an N-packet msgbuf. Blue arrows show NIC DMAs; the letters show the order in which the DMAs are performed for packets 1 and N.

6.3.2.1 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 6.1). Each eRPC packet has a header that contains the network 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% (Guideline 3.6.3).

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.

6.3.2.2 Message buffer ownership

Since eRPC transfers packets directly from application-owned msgbufs, eRPC must never use a msgbuf reference after returning ownership of the reference to the application. For brevity, we discuss msgbuf ownership issues for only clients; the process is similar but simpler for the server, since eRPC's servers are passive (Section 6.4). 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 (Section 6.4.2).

Maintaining this invariant is trivial 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 and the rate limiter next.

Msgbuf references in the 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 reduces message rates by up to 30% by using more NIC and PCIe resources (Guideline 3.6.3.2), 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.

Msgbuf references in the software rate limiter. On receiving the response for the first copy of a retransmitted request, we wish to ensure that eRPC's 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. Efficiently deleting references from the rate limiter turned out to be too complex (Section 6.4.2.1), so 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.

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

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.

6.3.2.3 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% (Section 6.5.2).

6.3.3 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 worker-mode RPC. We support a constant number of concurrent requests (default = 8) per session; eRPC queues additional requests transparently. This is inspired by how RDMA connections allow a constant number of operations [133]. 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 optimization avoids the overhead of dynamic memory allocation, and improves eRPC's message rate by up to 13% (§ 6.5.2).

6.3.3.1 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 (§ 6.4.2). Allowing BDP/MTU credits per session ensures that each session can achieve line rate. eRPC's BDP flow control is similar to the IRN proposal for RDMA NICs by Mittal et al. [116]. We compare eRPC to IRN in detail in Section 6.4.2.3.

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

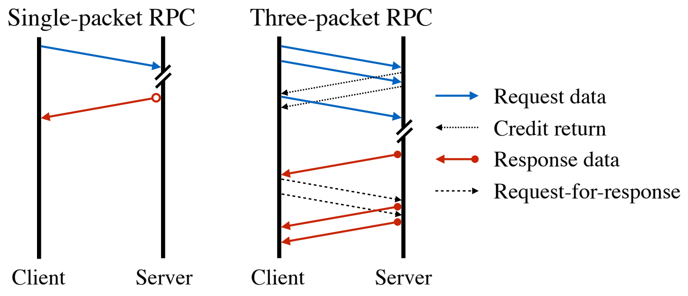


Figure 6.2: Examples of eRPC's wire protocol, with two credits per session

$|RQ|/C$ 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.

6.3.3.2 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.

6.4 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.

6.4.1 Protocol messages

Figure 6.2 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 for only 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.

6.4.2 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 [40]. 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 (§ 6.5.2).

6.4.2.1 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 [115], and ECN-based approaches such as DCQCN [165]. 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: clients use the congestion signals to adjust per-session sending rates. We implement Carousel's rate limiter [139], which is designed to efficiently handle a large number of sessions. We use Carousel's design as-is, so we omit the details.[1]

[1]We wished to support deleting enqueued packets in our Carousel implementation to simplify zero-copy trans-

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: The Ethernet switch in our private CX5 cluster does not support ECN marking [110, 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. 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.

6.4.2.2 Common-case congestion control optimizations

We use three congestion control 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. [138] report that 99% of all datacenter links are less than 10% utilized at one-minute timescales. Zhang et al. [162, 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 [115, 166].

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.

6.4.2.3 Comparison with IRN

IRN [116] 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 acknowledgments (SACKs) instead of simple go-back-N

mission (Section 6.3.2.2). However, doing so efficiently proved 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, which complicates rollback.

for packet loss recovery. Note that, unlike IRN, eRPC is a real system, and it does not require hardware support from RDMA NICs.

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.

6.4.3 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 [54, 162, 164] except during rare route churn events. Note that current RDMA NICs also drop reordered packets [116].

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.

6.5 Microbenchmarks

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

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 6.1: Comparison of median latency with eRPC and RDMA

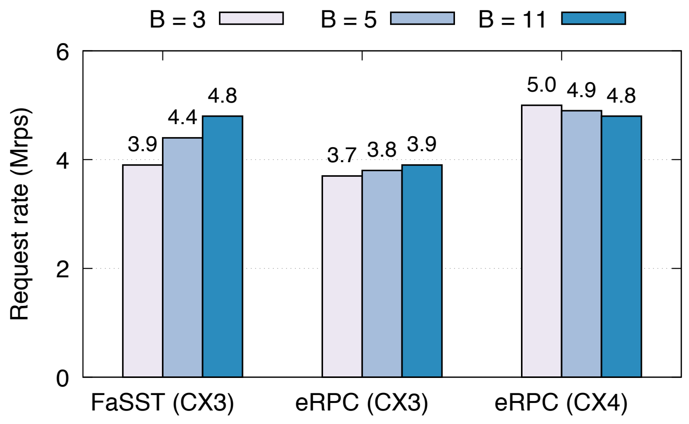


Figure 6.3: Single-core small-RPC rate with *B* requests per batch

6.5.1 Small RPC latency

How much latency does eRPC add? Table 6.1 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 [76, 127]. CX5's switch adds 300 ns to every layer-3 packet [147], meaning that end-host networking adds only ≈850 ns each at the client and server. This is comparable to the latency added by programmable switches. We discuss this further in § 6.6.1.

6.5.2 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 are *specialized* for single-packet RPCs in a lossless network, and they do not handle congestion.

We mimic FaSST's experiment setting (Section 5.3.5): one thread per node in an 11-node cluster, each of which 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

Action	RPC rate	% loss
Baseline (with congestion control)	4.96 M/s	–
Disable batched RTT timestamps (section 6.4.2)	4.84 M/s	2.4%
Disable Timely bypass (section 6.4.2)	4.52 M/s	6.6%
Disable rate limiter bypass (section 6.4.2)	4.30 M/s	4.8%
Disable multi-packet RQ (section 6.3.1)	4.06 M/s	5.6%
Disable preallocated responses (section 6.3.3)	3.55 M/s	12.6%
Disable 0-copy request processing (section 6.3.2.3)	3.05 M/s	14.0%

Table 6.2: Impact of disabling optimizations on small RPC rate (CX4)

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 6.3 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%.

Factor analysis. How important are eRPC's common-case optimizations? Table 6.2 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 (§ 6.4.2.2) 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.*

6.5.3 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 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)

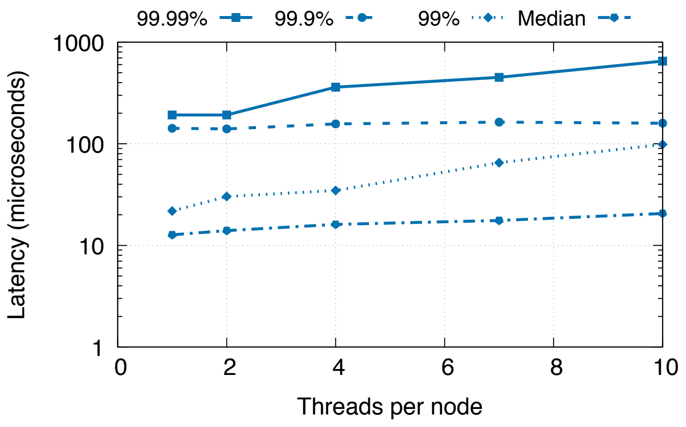


Figure 6.4: Latency with increasing threads on 100 CX4 nodes

workloads, which operate on small data items (Section 2.2.4).

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 6.4 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 6.1 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.

6.5.4 Large RPC bandwidth

We evaluate eRPC's bandwidth using one 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 the `perftest` benchmarking tool.

On the clusters listed in Table 2.2, eRPC saturates the network's per-port bandwidth with one CPU core. To understand eRPC's per-core performance limit, we connect two nodes in the CX5 cluster to a 100 Gbps switch via ConnectX-5 InfiniBand NICs. (We use CX5 as a 40 GbE cluster in the rest of this thesis.) Figure 6.5 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.

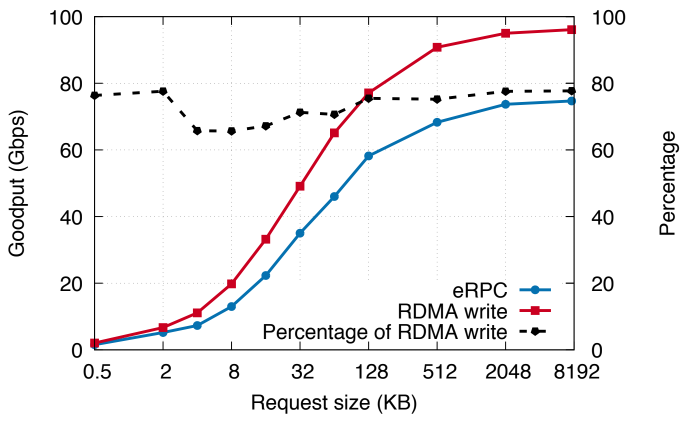


Figure 6.5: Throughput of large transfers over 100 Gbps InfiniBand

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

Table 6.3: eRPC's 8 MB request throughput with packet loss

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 [42, 45]. 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 (Section 6.4.1) can reduce packet processing overhead.

Table 6.3 shows the throughput with $R = 8$ MB, 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%) [165], likely because of support for negative acknowledgments, which eRPC currently lacks.

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 6.4: Effectiveness of congestion control (cc) during incast

6.5.5 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 [115].

Table 6.4 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. [166, § 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 [166].

Incast with background traffic. Next, we augment the setup above to mimic an experiment from Timely [115, 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.

6.6 Full-system evaluations

This section describes our experience with building real systems with eRPC, and the implementation and evaluation of these systems. We show that eRPC brings the speed of modern datacenter networks to unmodified existing storage software: We build a state machine replication system using an open-source implementation of Raft [121], and a networked ordered key-value store using Masstree [105].

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 6.5: Latency comparison for replicated PUTs

6.6.1 Raft over eRPC

We have implemented a highly-available replicated in-memory key-value store on top of eRPC. We avoid the complexity of implementing a state machine replication system from scratch (Section 2.2.3): we use an existing implementation of the Raft protocol [23]. (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.

We compare against recent consistent replication systems that are built from scratch for two specialized hardware types. First, NetChain [76] implements chain replication over programmable switches. Other replication protocols such as conventional primary-backup and Raft are too complex to implement over programmable switches [76]. Therefore, despite the protocol-level differences between LibRaft-over-eRPC and NetChain, our comparison helps understand the relative performance of end-to-end software-based designs and switch-based designs for in-memory replication. Second, Consensus in a Box [72] (called ZabFPGA here), implements ZooKeeper's atomic broadcast protocol [62] on FPGAs. eRPC also outperforms DARE [129], which implements SMR over RDMA; we omit the results for brevity.

Workloads. We mimic NetChain and ZabFPGA's experiment setups for latency measurement: we use three-way replication, 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 [97]. We compare eRPC's performance on CX5 against their published numbers because we do not have the hardware to run NetChain or ZabFPGA. Table 6.5 compares the latencies of the three systems.

6.6.1.1 Comparison with NetChain

NetChain's key assumption is that software networking adds 1–2 orders of magnitude more latency than switches [76]. However, we have shown that eRPC adds 850 ns, which is only around 2x higher than latency added by current programmable switches (400 ns [10]).

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.

6.6.1.2 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, highlighted by ZabFPGA's negligible leader latency variance. This advantage does not carry over directly to end-to-end latency [72] because storage systems built with specialized hardware are eventually accessed by clients running on commodity workstations.

6.6.2 Masstree over eRPC

Masstree [105] 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 query 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 [114]. 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.

6.7 Conclusion

eRPC is the first general-purpose communication library that provides near-network-speed performance in modern lossy Ethernet datacenters. eRPC's speed comes from the observation that switch buffer capacity far exceeds datacenter BDP, prioritizing common-case performance, and carefully combining a wide range of old and new optimizations. eRPC delivers performance that was until now believed possible only with lossless RDMA fabrics or specialized network hardware. It provides a flexible, end-to-end alternative to putting more functions in network hardware, and specialized system designs that depend on these functions. 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.

"How dull it is to pause, to make an end"

Alfred Lord Tennyson

Chapter 7

Lessons learned, limitations, and looking forward

We conclude this thesis with discussion about the high-level lessons from our research, the limitations of our systems' evaluation, and avenues for future research.

7.1 Lessons learned

CPU architecture is well-matched to running high-performance distributed systems. At the time of writing, there is a tremendous push in the systems and networks research community towards placing distributed system logic into network devices (i.e., "in-network computing"). Our work suggests that CPUs are, in fact, a good match for building distributed systems, in terms of performance, scalability, flexibility, and simplicity. The general-purpose programmability of CPUs allows easy innovation, debugging, testing, deployment, and addition of new features and technologies. These factors typically do not get sufficient attention by the research community while evaluating the costs and benefits of in-network computing for distributed systems.

Although this thesis does not explore power efficiency, our systems deliver higher performance with equal or fewer hardware resources, indicating that they have higher performance per watt, too. In addition, our prior work demonstrated that CPUs are competitive with FPGAs for in-memory key-value storage [94], which is a workload with characteristics similar to those explored in this thesis.

Stronger CPU baselines are crucial for in-network computing research. Many in-network computing projects demonstrate higher performance and CPU savings in comparison to slow software baselines that do not represent the highest performance achievable by software-only designs. Our work on eRPC provides a way forward for better comparisons: comparing against a fast baseline allows zooming-in on fundamental differences between CPUs and in-network devices, such as their hardware architecture, and placement in the network. As an extreme example, NetChain [76] compares a highly-optimized switch-based state machine replication system against Apache ZooKeeper, which uses the kernel's network stack. Our work suggests that much of the improvements observed in these projects can also be achieved by improving the software baseline with state-of-the-art, software-only techniques. For example, we showed in Chapter 6 that a state machine replication system running on a kernel-bypass communication library outperforms NetChain.

Low-level optimizations are sufficiently critical that they can change the relative performance of high-level designs. We are used to thinking about low-level optimizations as contributing a few percentage points to overall system performance. However, due to the complexity and feature-richness of modern hardware, the effect of low-level optimizations is often big enough to change which high-level system design is faster or more scalable. These optimizations are often obscure and challenging to get right, which is why prior work sometimes missed speedup opportunities. For example, in the context of their performance measurements that missed some low-level optimizations, the designers of Pilaf and FaRM-KV made the correct decision to use multiple RDMA reads over one RPC; their RPCs were slow, but not fundamentally so.

Consider end-to-end designs early. Although eRPC builds on top of many recent research results, it was (in theory) possible to create eRPC six years ago when we started this work. For several years, we mistakenly—but for good reason—believed that one-sided RDMA and/or lossless networks were necessary for good performance. We and other researchers did not consider end-to-end designs because we believed that such designs would not perform well, in part because we had not found all the required low-level optimizations. In the end, we re-discovered an essential lesson from the end-to-end arguments paper: *"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."*

7.2 Limitations

Limited NIC model variety. At the time of writing, Mellanox is the only major vendor of high-speed NICs with userspace device drivers. NICs from other vendors add several microseconds of latency, or require going through the OS kernel, or are not easily purchasable. As a consequence, our experiments use only Mellanox NICs, although we have tested our findings on NICs released over the course of a decade (Table 2.2). The selection of fast NICs is likely to improve soon, with new 100 GbE NICs from Intel and Broadcom on the horizon.

Evaluation in large datacenters. We conducted our research in an academic setting, where we had access to clusters with up to 200 hosts, but lacked access to large datacenters with thousands of hosts. In addition to large scale, networks in large datacenters also carry traffic from other applications that use communication protocols different from eRPC. Although we have not tested eRPC at very large scale and with co-located traffic, we believe that it will work well in such settings. A key determinant of performance in large, co-located environments is the congestion control protocol, and we deliberately chose to use an existing congestion control protocol (i.e., Timely) that has been deployed in hyperscale datacenters at Google. In addition, datacenters often confine communication-intensive workloads to pods [95]. Pods contain a few hundred hosts and each pod runs one workload, which is a setting that we have shown eRPC performs well in.

Co-located workloads. To improve resource utilization, datacenter operators often co-locate workloads on hosts. For example, a latency-sensitive in-memory key-value store application may run alongside a low-priority data analytics application on the same host. During a period in which the key-value store is idle, the host's operating system may use all CPU cores to run analytics. As a result, incoming RPC requests for a key-value item will suffer from the overhead of interrupts and process context switching. We designed eRPC for use in more exclusive settings, where some cores or the entire machine is available to the application using eRPC. Recent work from Kaffes et al. [78] and Ousterhout et al. [123] shows promising results for achieving both high utilization and high performance in such co-located settings, which can apply to eRPC.

7.3 Future work

It is an especially exciting time for high-performance networking research: fast networks are now commonplace in datacenters, and non-volatile memory has finally broken the storage latency barrier. The software bloat and inefficiency in current implementations of networking software can no longer hide behind slow networks and storage.

This thesis investigated how far existing hardware architectures can go towards achieving the speed of the underlying network in these systems. We now understand the limits of CPUs, and are in a more informed position to investigate specialized hardware solutions. For example, eRPC shows that line rate on current 100 Gbps networks (with one core) is achievable without in-network support, but we likely cannot reach terabit-per-second speeds. What functionality, if any, should we put into networks to reach terabit network speeds in applications?

Our work opens several new avenues in datacenter networking that are ripe for investigation, discussed next.

7.3.1 Protocol and API design for datacenter networks

One of eRPC's biggest contributions was showing that we can support reliability and congestion control with low CPU overhead. It is therefore unclear why existing datacenter transport protocols such as TCP require much higher CPU cycles per packet: One CPU core can handle 15 million RPCs per second with eRPC, but only around 1.5 million packets per second with optimized userspace TCP implementations. Early pioneers of TCP have shown that TCP's original datapath is simple and can be implemented with low overhead [28]. Is this still the case? If so, a likely cause for low performance could be the POSIX send/recv API that handles only one connection per call. In contrast, eRPC's API is explicitly designed to permit batched processing of multiple RPCs, on possibly different connections. Such batching allows several optimizations, such as faster network I/O and prefetching memory locations. These results, together with the increasing need for the underlying hardware to be parallel, suggest that a batching-centric redesign of operating system APIs may yield substantial benefits.

7.3.2 Towards a full-fledged networking library

Developers often desire features that are currently not provided by eRPC, such as encryption, and marshalling and unmarshalling of RPC arguments. These features are considered expensive, but we hypothesize that much of that perceived cost arises due to implementation inefficiencies and mismatches between hardware requirements and current API designs. For example, we believe that we can provide efficient secure communication in an end-to-end fashion without in-network hardware devices such as FPGAs and on-NIC accelerators. Authenticated encryption on modern CPUs is as fast as 0.64 cycles per byte [53], meaning that one CPU core may send or receive encrypted data over the network at 40 Gbps on a modern 3.5 GHz processor. Another example of the usefulness of batched APIs is that, due to intra-core hardware parallelism, these instructions are even more effective if multiple messages are available for simultaneous processing.

The database community, in the heyday of in-memory and NoSQL databases, tried to understand what makes a full-featured database slow [57]. They analyzed which components are slow fundamentally, and which due to implementation and design. A similar analysis for datacenter networking is needed to establish a grounded basis for the development of future systems that successfully balance utility and efficiency.

More generally, by better leveraging existing hardware, redesigning APIs with a grounded understanding of what limits efficiency, and carefully choosing hardware specializations, we can lay the groundwork for creating future systems that match the performance of ever-faster networks.

Bibliography

[1] Private communication with FaRM's authors.

[2] Private communication with Mellanox.

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

[4] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote memory in the age of fast networks. In *Proc. 8th ACM Symposium on Cloud Computing (SOCC)*, Santa Clara, CA, September 2017.

[5] Mohammad Al-Fares, Alex Loukissas, and Amin Vahdat. A scalable, commodity, data center network architecture. In *Proc. ACM SIGCOMM*, Seattle, WA, August 2008.

[6] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pfabric: Minimal near-optimal datacenter transport. In *Proc. ACM SIGCOMM*, Hong Kong, China, August 2013.

[7] Bob Alverson, Edwin Froese, Larry Kaplan, and Duncan Roweth. Cray XC series network. *Cray Inc., White Paper WP-Aries01-1112*, 2012.

[8] Amazon. Amazon Prime Day 2019 - Powered by AWS. https://aws.amazon.com/blogs/aws/amazon-prime-day-2019-powered-by-aws/.

[9] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the SIGMETRICS'12*, June 2012.

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

[11] AWS Nitro System. https://aws.amazon.com/ec2/nitro/.

[12] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei, and John D. Davis. CORFU: a shared log design for flash clusters. In *Proc. 9th USENIX NSDI*, San Jose, CA, April 2012.

[13] Barefoot Networks: Use Cases. https://www.barefootnetworks.com/use-cases/.

[14] Brian W Barrett, Ron Brightwell, Scott Hemmert, Kevin Pedretti, Kyle Wheeler, Keith Underwood, Rolf Riesen, Arthur B Maccabe, and Trammell Hudson. The Portals 4.0 network programming interface November 14, 2012 draft.

[15] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and

Edouard 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.

[16] Carsten Binnig, Ugur Çetintemel, Andrew Crotty, Alex Galakatos, Tim Kraska, Erfan Zamanian, and Stanley B. Zdonik. The end of slow networks: It's time for a redesign. *CoRR*, abs/1504.01048, 2015. URL http://arxiv.org/abs/1504.01048.

[17] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. The end of slow networks: It's time for a redesign. In *Proc. VLDB*, New Delhi, India, August 2016.

[18] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 1984.

[19] Mark S. Birrittella, Mark Debbage, Ram Huggahalli, James Kunz, Tom Lovett, Todd Rimmer, Keith D. Underwood, and Robert C. Zak. Intel Omni-path architecture: Enabling scalable, high performance fabrics. In *Proceedings of the 2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, 2015.

[20] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *Proc. ACM SIGCOMM*, Hong Kong, China, August 2013.

[21] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. TCP vegas: New techniques for congestion detection and avoidance. In *Proc. ACM SIGCOMM*, London, England, August 1994.

[22] Broadcom Ethernet Network Adapters. https://www.broadcom.com/products/ethernet-connectivity/network-adapters.

[23] C implementation of the Raft Consensus protocol. https://github.com/willemt/raft.

[24] Brent Callaghan, Theresa Lingutla-Raj, Alex Chiu, Peter Staubach, and Omer Asad. NFS over RDMA. In *Proceedings of the ACM SIGCOMM Workshop on Network-I/O Convergence: Experience, Lessons, Implications*, 2003.

[25] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *Proc. 7th USENIX OSDI*, Seattle, WA, November 2006.

[26] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. Fast and general distributed transactions using RDMA and HTM. In *Proc. 11th ACM European Conference on Computer Systems (EuroSys)*, April 2016.

[27] D. Clark. The Design Philosophy of the DARPA Internet Protocols. In *Proc. ACM SIGCOMM*, pages 109–114, Stanford, CA, August 1988.

[28] D. C. Clark, V. Jacobson, J. Romkey, and H. Salwen. An Analysis of TCP Processing Overhead. *IEEE Communications Magazine*, June 1989.

[29] Brian Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears.

Benchmarking cloud serving systems with YCSB. In *Proc. 1st ACM Symposium on Cloud Computing (SOCC)*, Indianapolis, IN, June 2010.

[30] 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.

[31] Andy Currid. TCP offload to the rescue. *Queue*. doi: 10.1145/1005062.1005069. URL http://doi.acm.org/10.1145/1005062.1005069.

[32] Jeffrey Dean. Keynote address: Designs, lessons and advice from building large distributed systems. In *LADIS*, 2009.

[33] Matt DeBergalis, Peter Corbett, Steve Kleiman, Arthur Lent, Dave Noveck, Tom Talpey, and Mark Wittle. The direct access file system. In *Proceedings of the 2Nd USENIX Conference on File and Storage Technologies*, 2003.

[34] Guiseppe DeCandia, Deinz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swami Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. 21st ACM Symposium on Operating Systems Principles (SOSP)*, Stevenson, WA, October 2007.

[35] Saïd Derradji, Thibaut Palfer-Sollier, Jean-Pierre Panziera, Axel Poudes, and François Wellenreiter Atos. The BXI interconnect architecture. In *Proceedings of the 2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, 2015.

[36] Distributed Asynchronous Object Storage Stack. https://github.com/daos-stack.

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

[38] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. FaRM: Fast remote memory. In *Proc. 11th USENIX NSDI*, Seattle, WA, April 2014.

[39] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *Proc. 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, October 2015.

[40] Aleksandar Dragojevic, Dushyanth Narayanan, and Miguel Castro. RDMA reads: To use or not to use? *IEEE Data Eng. Bull.*, 2017.

[41] Dave Dunning, Greg Regnier, Gary McAlpine, Don Cameron, Bill Shubert, Frank Berry, Anne Marie Merritt, Ed Gronke, and Chris Dodd. The virtual interface architecture. *IEEE Micro*, pages 66–76, 1998.

[42] Michael Dalton et al. Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In *Proc. 15th USENIX NSDI*, Renton, WA, April 2018.

[43] 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/.

[44] Bin Fan, David G. Andersen, and Michael Kaminsky. MemC3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Proc. 10th USENIX NSDI*,

Lombard, IL, April 2013.

[45] 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.

[46] Wu-chun Feng, Pavan Balaji, Chris Baron, Laxmi N Bhuyan, and Dhabaleswar K Panda. Performance characterization of a 10-gigabit Ethernet TOE. In *13th Symposium on High Performance Interconnects (HOTI'05)*. IEEE, 2005.

[47] Daniel Firestone et al. Azure accelerated networking: SmartNICs in the public cloud. In *Proc. 15th USENIX NSDI*, Renton, WA, April 2018.

[48] Mario Flajslik and Mendel Rosenblum. Network interface design for low latency request-response protocols. In *Proc. USENIX Annual Technical Conference*, San Jose, CA, June 2013.

[49] S. Floyd. TCP and Explicit Congestion Notification. *ACM Computer Communications Review*, 24(5), October 1994.

[50] Sebastian Gallenmüller, Paul Emmerich, Florian Wohlfart, Daniel Raumer, and Georg Carle. Comparison of frameworks for high-performance packet io. In *ANCS*, 2015.

[51] Garth Gibson, Gary Grider, Andree Jacobson, and Wyatt Lloyd. PRObE: A Thousand-Node Experimental Cluster for Computer Systems Research.

[52] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient memory disaggregation with Infiniswap. In *Proc. 14th USENIX NSDI*, Boston, MA, March 2017.

[53] Shay Gueron, Adam Langley, and Yehuda Lindell. Aes-gcm-siv: Specification and analysis. 2017.

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

[55] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over commodity Ethernet at scale. In *Proc. ACM SIGCOMM*, Florianopolis, Brazil, August 2016.

[56] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. PacketShader: a GPU-accelerated software router. In *Proc. ACM SIGCOMM*, New Delhi, India, August 2010.

[57] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. OLTP through the looking glass, and what we found there. In *Proc. ACM SIGMOD*, Vancouver, BC, Canada, June 2008.

[58] Simon Hauger, Thomas Wild, Arthur Mutter, Andreas Kirstaedter, Kimon Karras, Rainer Ohlendorf, Frank Feller, and Joachim Scharf. Packet processing at 100 Gbps and beyond - challenges and perspectives. In *Photonic Networks, 2009 ITG Symposium on*, 2009.

[59] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. IronFleet: Proving practical distributed systems correct. In *Proc. 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey,

CA, October 2015.

[60] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. Hopscotch hashing. In *Proceedings of the 22Nd International Symposium on Distributed Computing*, 2008.

[61] Jian Huang, Xiangyong Ouyang, Jithin Jose, Md. Wasi ur Rahman, Hao Wang, Miao Luo, Hari Subramoni, Chet Murthy, and Dhabaleswar K. Panda. High-Performance Design of HBase with RDMA over InfiniBand. In *IPDPS*, 2012.

[62] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: wait-free coordination for internet-scale systems. In *Proc. USENIX Annual Technical Conference*, Boston, MA, June 2010.

[63] IEEE 802.1Qbb – Priority-based Flow Control. https://1.ieee802.org/dcb/802-1qbb/.

[64] InfiniBand Architecture Specification Volume 1. https://cw.infinibandta.org/document/dl/7859.

[65] Intel 82599. Intel 82599 10 Gigabit Ethernet Controller: Datasheet. http://www.intel.com/content/www/us/en/ethernet-controllers/82599-10-gbe-controller-datasheet.html, 2013.

[66] Intel Atom Processor C2000 Product Family for Microserver. http://www.intel.in/content/dam/www/public/us/en/documents/datasheets/atom-c2000-microserver-datasheet.pdf.

[67] Intel Optane DC Persistent Memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[68] Intel Xeon Phi Processor Knights Landing Architectural Overview. https://www.nersc.gov/assets/Uploads/KNL-ISC-2015-Workshop-Keynote.pdf.

[69] Intel Xeon Processor D-1500 Product Family. http://www.intel.in/content/dam/www/public/us/en/documents/product-briefs/xeon-processor-d-brief.pdf.

[70] Intel Xeon Processor E5-1600/2400/2600/4600 (E5-Product Family) Product Families. http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xeon-e5-1600-2600-vol-2-datasheet.pdf.

[71] Intel Xeon Processor E5-1600/2400/2600/4600 v3 Product Families. http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xeon-e5-v3-datasheet-vol-2.pdf.

[72] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolic. Consensus in a box: Inexpensive coordination in hardware. In *Proc. 13th USENIX NSDI*, Santa Clara, CA, May 2016.

[73] Zsolt István, David Sidler, and Gustavo Alonso. Caribou: Intelligent distributed storage. August 2017.

[74] EunYoung Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: A highly scalable user-level TCP stack for multicore systems. In *Proc. 11th USENIX NSDI*, Seattle, WA, April 2014.

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

[76] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. NetChain: Scale-free sub-RTT coordination. In *Proc. 15th USENIX NSDI*, Renton, WA, April 2018.

[77] Jithin Jose, Hari Subramoni, Krishna Kandalla, Md. Wasi-ur Rahman, Hao Wang, Sundeep Narravula, and Dhabaleswar K. Panda. Scalable memcached design for infiniband clusters using hybrid transports. In *Proc. CCGRID*, 2012.

[78] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for microsecond-scale tail latency. In *Proc. 16th USENIX NSDI*, Boston, MA, February 2019.

[79] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using RDMA efficiently for key-value services. In *Proc. ACM SIGCOMM*, Chicago, IL, August 2014.

[80] Anuj Kalia, Dong Zhou, Michael Kaminsky, and David G. Andersen. Raising the bar for using GPUs in software packet processing. In *Proc. 12th USENIX NSDI*, Oakland, CA, May 2015.

[81] Anuj Kalia, Michael Kaminsky, and David G. Andersen. FaSST: Fast, scalable and simple distributed transactions with two-sided RDMA datagram RPCs. In *Proc. 12th USENIX OSDI*, Savannah, GA, November 2016.

[82] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663, New York, NY, USA, 1997. ACM.

[83] Daehyeok Kim, Amirsaman Memaripour, Anirudh Badam, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Shachar Raindel, Steven Swanson, Vyas Sekar, and Srinivasan Seshan. HyperLoop: Group-based NIC-offloading to accelerate replicated transactions in multi-tenant storage systems. In *Proc. ACM SIGCOMM*, Budapest, Hungary, August 2018.

[84] 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.

[85] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32(4), December 2001.

[86] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical Paxos and primary-backup replication. Technical report, Microsoft Research, 2009.

[87] Steen Larsen and Ben Lee. Platform io dma transaction acceleration. In *CACHES*. ACM, 2011.

[88] Changhyun Lee, Chunjong Park, Keon Jang, Sue Moon, and Dongsu Han. Accurate latency-based congestion feedback for datacenters. In *Proc. USENIX Annual Technical Conference*, Santa Clara, CA, June 2015.

[89] Viktor Leis, Alfons Kemper, and Thomas Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, 2013.

[90] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. KV-Direct: High-performance in-memory key-value store with programmable NIC. In *Proc. 26th ACM Symposium on Operating Systems Principles (SOSP)*, Shanghai, China, October 2017.

[91] Bojie Li, Tianyi Cui, Yibo Wang, Wei Bai, and Lintao Zhang. SocksDirect: Datacenter sockets can be fast and compatible. In *Proc. ACM SIGCOMM*, Beijing, China, August 2019.

[92] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. Just say no to Paxos overhead: Replacing consensus with network ordering. In *Proc. 12th USENIX OSDI*, Savannah, GA, November 2016.

[93] Jialin Li, Ellis Michael, and Dan 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, October 2017.

[94] Sheng Li, Hyeontaek Lim, Victor W. Lee, Jung Ho Ahn, Anuj Kalia, Michael Kaminsky, David G. Andersen, O. Seongil, Sukhan Lee, and Pradeep Dubey. Architecting to achieve a billion requests per second throughput on a single key-value store server platform. In *ISCA*, 2015.

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

[96] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. SILT: A memory-efficient, high-performance key-value store. In *Proc. 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, October 2011.

[97] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *Proc. 11th USENIX NSDI*, Seattle, WA, April 2014.

[98] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[99] Feilong Liu, Lingyan Yin, and Spyros 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)*, April 2017.

[100] Jiuxing Liu, Jiesheng Wu, and Dhabaleswar K Panda. High performance RDMA-based MPI implementation over InfiniBand. *International Journal of Parallel Programming*, 2004.

[101] Jiuxing Liu, Jiesheng Wu, and Dhabaleswar K Panda. High performance RDMA-based MPI implementation over InfiniBand. *International Journal of Parallel Programming*, 2004.

[102] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, and Simon Peter. Offloading distributed applications onto SmartNICs using iPipe. In *Proc. ACM SIGCOMM*, Beijing, China, August 2019.

[103] Xiaoyi Lu, Nusrat S. Islam, Md. Wasi ur Rahman, Jithin Jose, Hari Subramoni, Hao Wang, and Dhabaleswar K. Panda. High-performance design of hadoop RPC with RDMA over InfiniBand. In *ICPP*, 2013.

[104] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. Octopus: an RDMA-enabled distributed persistent memory file system. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017.

[105] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proc. 7th ACM European Conference on Computer Systems (EuroSys)*, Bern, Switzerland, April 2012.

[106] Marvell LiquidIO 2 Smart NICs. https://www.marvell.com/ethernet-adapters-and-controllers/liquidio-smart-nics/.

[107] Mellanox BlueField Multicore System on a Chip. http://www.mellanox.com/products/bluefield-overview/.

[108] Mellanox ConnectX-4 Product Brief. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_ConnectX-4_VPI_Card.pdf.

[109] Mellanox Innova-2 Flex Open Programmable SmartNIC. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_Innova-2_Flex.pdf.

[110] Mellanox MLNX-OS User Manual for Ethernet. http://www.mellanox.com/related-docs/prod_management_software/MLNX-OS_ETH_v3_6_3508_UM.pdf.

[111] 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.

[112] Memcached. Memcached: A distributed memory object caching system. http://memcached.org/, 2011.

[113] Christopher Mitchell, Yifeng Geng, and Jinyang 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.

[114] Christopher Mitchell, Kate Montgomery, Lamont Nelson, Siddhartha Sen, and Jinyang Li. Balancing CPU and network in the Cell distributed B-Tree store. In *Proc. USENIX Annual Technical Conference*, Denver, CO, June 2016.

[115] Radhika Mittal, Terry Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. TIMELY: RTT-based congestion control for the datacenter. In *Proc. ACM SIGCOMM*, London, UK, August 2015.

[116] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. Revisiting network support for RDMA. In *Proc. ACM SIGCOMM*, Budapest, Hungary, August 2018.

[117] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-tolerant software distributed shared memory. In *Proc. USENIX Annual Technical Conference*, Santa Clara, CA, June 2015.

[118] Netronome Agilio SmartNICs. https://www.netronome.com/products/smartnic/overview/.

[119] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *Proc. 10th USENIX NSDI*, Lombard, IL, April 2013.

[120] Oak Ridge Leadership Computing Facility - Summit. https://www.olcf.ornl.gov/summit/.

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

[122] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast crash recovery in RAMCloud. In *Proc. 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, October 2011.

[123] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *Proc. 16th USENIX NSDI*, Boston, MA, February 2019.

[124] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. The RAMCloud storage system. *ACM TOCS*, 2015.

[125] P4 Language Consortium. https://p4.org.

[126] R. Pagh and F.F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, May 2004.

[127] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. NetBricks: Taking the V out of NFV. In *Proc. 12th USENIX OSDI*, Savannah, GA, November 2016.

[128] PCI Express Base Specification Revision 3.0. https://pcisig.com/specifications/.

[129] Marius Poke and Torsten Hoefler. DARE: High-performance state machine replication on RDMA networks. In *HPDC*, 2015.

[130] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. Designing distributed systems using approximate synchrony in data center networks. In *Proc. 12th USENIX NSDI*, Oakland, CA, May 2015.

[131] Ken Raffenetti, Abdelhalim Amer, Lena Oden, Charles Archer, Wesley Bland, Hajime Fujita, Yanfei Guo, Tomislav Janjusic, Dmitry Durnov, Michael Blocksome, Min Si, Sangmin Seo, Akhil Langer, Gengbin Zheng, Masamichi Takagi, Paul Coffman, Jithin Jose, Sayantan Sur, Alexander Sannikov, Sergey Oblomov, Michael Chuvelev, Masayuki Hatanaka, Xin Zhao, Paul Fischer, Thilina Rathnayake, Matt Otten, Misun Min, and Pavan Balaji. Why is mpi so slow?: Analyzing the fundamental limits in implementing MPI-3.1. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, 2017.

[132] Shlomo Raikin, Liran Liss, Ariel Shachar, Noam Bloch, and Michael Kagan. Remote transactional memory, 2015. US Patent App. 20150269116.

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

[134] Redis. http://redis.io.

[135] David P. Reed. Invited talk: End-to-end arguments: The Internet and beyond. In *USENIX Security*, 2010.

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

[137] Luigi Rizzo. netmap: a novel framework for fast packet I/O. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, June 2012.

[138] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the social network's (datacenter) network. In *Proc. ACM SIGCOMM*, London, UK, August 2015.

[139] Ahmed Saeed, Nandita Dukkipati, Vytautas Valancius, Vinh The Lam, Carlo Contavalli, and Amin Vahdat. Carousel: Scalable traffic shaping at end hosts. In *Proc. ACM SIGCOMM*, Los Angeles, CA, August 2017.

[140] J. Saltzer, D. Reed, and D. Clark. End-to-end Arguments in System Design. *ACM Transactions on Computer Systems*, 2:277–288, November 1984.

[141] Yee Jiun Song, Marcos K. Aguilera, Ramakrishna Kotla, and Dahlia Malkhi. RPC chains: Efficient client-server communication in geodistributed systems. In *Proc. 6th USENIX NSDI*, Boston, MA, April 2009.

[142] James W. Stamos and Flaviu Cristian. Coordinator log transaction execution protocol. *Distrib. Parallel Databases*, 1(4):383–408, October 1993. ISSN 0926-8782. doi: 10.1007/BF01264014. URL http://dx.doi.org/10.1007/BF01264014.

[143] Tyler Szepesi, Bernard Wong, Benjamin Cassell, , and Tim Brecht. Designing a low-latency cuckoo hash table for write-intensive workloads. In *WSRC*, 2014.

[144] Tyler Szepesi, Benjamin Cassell, Bernard Wong, Tim Brecht, and Xiaoyi Liu. Nessie: A decoupled, client-driven, key-value store using RDMA. Technical Report CS-2015-09, University of Waterloo, David R. Cheriton School of Computer Science, Waterloo, Canada, June 2015.

[145] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, May 2012.

[146] Tofino: World's fastest P4-programmable Ethernet switch ASICs. https://barefootnetworks.com/products/brief-tofino/.

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

[148] TPC-C. TPC benchmark C. http://www.tpc.org/tpcc/, 2010.

[149] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proc. 24th ACM Symposium on Operating Systems Principles (SOSP)*, Farmington, PA, November 2013.

[150] Understanding Performance of PCI Express Systems. http://www.xilinx.com/support/documentation/white_papers/wp350.pdf.

[151] Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *Proc. 6th USENIX OSDI*, San Francisco, CA, December 2004.

[152] Yandong Wang, Xiaoqiao Meng, Li Zhang, and Jian 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, November 2014.

[153] Yandong Wang, Li Zhang, Jian Tan, Min Li, Yuqing Gao, Xavier Guerin, Xiaoqiao Meng, and Shicong 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.

[154] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using RDMA and HTM. In *Proc. 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, October 2015.

[155] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. Deconstructing RDMA-enabled distributed transactions: Hybrid is better! In *Proc. 13th USENIX OSDI*, Carlsbad, CA, October 2018.

[156] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. 5th USENIX OSDI*, pages 255–270, Boston, MA, December 2002.

[157] Jiesheng Wu, Pete Wyckoff, and Dhabaleswar K. Panda. PVFS over InfiniBand: Design and performance evaluation. In *Ohio State University Tech Report*, 2003.

[158] Reza Zamani and Ahmad Afsahi. Communication characteristics of message-passing scientific and engineering applications. In *International Conference on Parallel and Distributed Computing Systems*, 2005.

[159] Erfan Zamanian, Carsten Binnig, Tim Harris, and Tim Kraska. The end of a myth: Distributed transactions can scale. In *Proc. VLDB*, Munich, Germany, August 2017.

[160] Huanchen Zhang, David G. Andersen, Andrew Pavlo, Michael Kaminsky, Lin Ma, and Rui Shen. Reducing the storage overhead of main-memory OLTP databases with hybrid indexes. In *Proc. ACM SIGMOD*, San Francisco, USA, June 2016.

[161] Jiao Zhang, Fengyuan Ren, Xin Yue, Ran Shu, and Chuang Lin. Sharing bandwidth by allocating switch buffer in data center networks. *IEEE Journal on Selected Areas in Communications*, 2014.

[162] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. High-resolution measurement of data center microbursts. In *Proceedings of the 2017 Internet Measurement*

Conference, IMC '17, 2017.

[163] Dong Zhou, Bin Fan, Hyeontaek Lim, David G. Andersen, and Michael Kaminsky. Scalable, High Performance Ethernet Forwarding with CuckooSwitch. In *Proc. 9th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, December 2013.

[164] Junlan Zhou, Malveeka Tewari, Min Zhu, Abdul Kabbani, Leon Poutievski, Arjun Singh, and Amin Vahdat. WCMP: Weighted cost multipathing for improved fairness in data centers. In *Proc. 9th ACM European Conference on Computer Systems (EuroSys)*, April 2014.

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

[166] Yibo Zhu, Monia Ghobadi, Vishal Misra, and Jitendra Padhye. ECN or delay: Lessons learnt from analysis of DCQCN and TIMELY. In *Proc. CoNEXT*, December 2016.