DISS. ETH NO. 28141

Data management in modern RDMA-capable networks

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZURICH
(Dr. sc. ETH Zurich)

presented by

Konstantin Taranov

MSc in Computational Engineering,
Lappeenranta University of Technology

born on 18.02.1992

accepted on the recommendation of

Prof. Dr. Torsten Hoefler (ETH Zurich), examiner
Prof. Dr. Gustavo Alonso (ETH Zurich), co-examiner
Dr. Miguel Castro (Microsoft Research Cambridge), co-examiner

2022

Abstract

Current trends in modern hardware create many opportunities and challenges for data management systems. More specifically, novel high-performance network devices with the capability of high-throughput packet processing offer the potential of significant improvements in the performance of networked systems. However, advanced in-network acceleration and offload capabilities, including RDMA, often have strong constraints which force developers to limit the functionality of designed systems or even sacrifice their security.

This dissertation aims to address challenges in the design and implementation of the data management protocols that efficiently use the offload capabilities offered by modern network accelerators in the context of various data storage and data analytics systems. Particularly, I address challenges in data management for key-value stores, shared message queues, and remote memory systems regarding storage reliability, performance, and memory fragmentation.

Second, I propose a serialization-free communication library for Java virtual machines that allows applications to send on-heap objects through RDMA connections. I show how the library unlocks RDMA networking to Java virtual machines hiding all the burden of low-level RDMA programming from the users.

Finally, I propose an extension to the InfiniBand architecture that enables authentication and encryption for RDMA networking to prevent information leakage and message tampering. I show how providers can implement RDMA secure channels with minimal changes to the existing InfiniBand protocol and with minor performance overheads.

I conclude by discussing future research directions which arise from the work presented in this dissertation, and highlighting the potential of in-network processing for modern data management platforms.

Zusammenfassung

Die aktuellen Hardware Entwicklungen eröffnen zahlreiche Möglichkeiten und Herausforderungen für Datenverwaltungsanwendungen. Insbesondere neuartige Hochleistungs-Netzwerkkarten mit der Fähigkeit zur Paketverarbeitung mit hohem Durchsatz bieten das Potenzial, die Leistung vernetzter Systeme erheblich zu verbessern. Allerdings sind fortschrittliche netzinterne Beschleunigungs- und Offload-Funktionen, einschliesslich RDMA, oft mit starken Einschränkungen verbunden, die Entwickler dazu zwingen, die Funktionalität der entworfenen Systeme einzuschränken oder sogar ihre Sicherheit zu opfern.

Diese Dissertation befasst sich mit den Herausforderungen beim Entwurf und der Implementierung von Datenverwaltungs Protokollen, die die von modernen Netzwerk Beschleunigern gebotenen Offload-Fähigkeiten im Zusammenhang mit verschiedenen Datenspeicher- und Datenanalyse Systemen effizient nutzen. Insbesondere befasse ich mich mit Herausforderungen bei der Datenverwaltung für Key-Value-Stores, Shared Message Queues und Remote-Memory-Systeme in Bezug auf Speicherzuverlässigkeit, Leistung und Speicherfragmentierung.

Zweitens schlage ich eine serialisierungsfreie Kommunikationsbibliothek für Javas virtuelle Maschinen vor, die es Anwendungen ermöglicht, On-Heap-Objekte über RDMA-Verbindungen zu senden. Ich zeige, wie die Bibliothek die RDMA-Vernetzung für virtuelle Java-Maschinen erlaubt, indem sie den Benutzern die gesamte Last der Low-Level-RDMA-Programmierung abnimmt.

Schließlich schlage ich eine Erweiterung der InfiniBand-Architektur vor, die Authentifizierung und Verschlüsselung für RDMA-Netzwerke ermöglicht, um Informationsverluste und Manipulation von Nachrichten zu verhindern. Ich zeige, wie Anbieter sichere RDMA-

Kanäle mit minimalen Änderungen am bestehenden InfiniBand-Protokoll und mit geringem Leistungsverlust implementieren können.

Abschließend erörtere ich zukünftige Forschungsrichtungen, die sich aus der in dieser Dissertation vorgestellten Arbeit ergeben, und zeige das Potenzial der Netzwerk-Internen Verarbeitung für moderne Plattformen zur Datenverwaltung auf.

Acknowledgements

First, I would like to express my gratitude to my adviser Torsten Hoefler for supervising my PhD studies here at ETH Zurich. He was always ready to give insightful advice, feedback, and ideas on my projects. His scientific guidance was essential to address the right research questions and helped to widen the scope of my work. I want to thank my co-adviser Gustavo Alonso for all his guidance in projects that we undertook together. His supervision was essential for my growth as a researcher and the success of my PhD study. I am also extremely grateful to Zaheer Chothia for his crucial help during first years of my PhD study. He was always ready to help me with any technical difficulties that I had in my first steps in the field of computer science research.

I acknowledge the generous financial support from ETH Zurich, and Microsoft Research through its Swiss Joint Research Centre. My research has been also funded from the European Research Council under the European Union Horizon 2020 programme (grant agreement DAPP, No. 678880). Without all that support, writing this work in the current shape would not have been possible. I value the contributions of all my co-authors and collaborators on published works. I especially enjoyed working with Rodrigo Bruno, Salvatore Di Girolamo, Ingo Müller, and Benjamin Rothenberger. I am deeply thankful to Steve Bean, Virendra Marathe, Adrian Perrig, Aleksandar Dragojevic, Dushyanth Narayanan, and Miguel Castro for inspiring discussions and various forms of help.

My special thanks goes out to people that I met in Zurich and during my internships for the fun moments we had together. Especially, I would like to thank Eric, Gaia, Beatrice, Ahmed, and Lisandra, who have been one of the greatest highlights of my life during PhD study. Last but not least, I would like to thank my wife, Anastasia Taranova, and my family for supporting me during all these years. Without their unconditional support in my every undertaking, writing this thesis would not have been possible.

Contents

1 Introduction **1**

1.1 Background on RDMA network accelerators 3

1.2 Challenges and Contributions 4

1.3 Dissertation Outline 7

1.4 Related publications 8

2 Per-item resilience in key-value stores **11**

2.1 Motivation 12

2.2 Ring applications 14

2.3 Storage schemes 16

2.3.1 Replication 16

2.3.2 Reed-Solomon coding 16

2.3.3 Stretched Reed-Solomon coding 18

2.4 System architecture 23

2.4.1 API 24

2.4.2 Data layout, memgests 24

2.4.3 Strong consistency 27

2.4.4 Erasure coded and replicated memgests 29

2.4.5 Balancing 29

2.4.6 Membership and handling failures 30
2.5 Evaluation 31
2.5.1 Latency 32
2.5.2 Move requests 34
2.5.3 Throughput 36
2.5.4 Failures and recovery 38
2.6 Related work 40
2.7 Summary and Discussion 40

3 Compactable Remote Memory over RDMA 43
3.1 Motivation 44
3.2 Background on Shared Memory Systems 45
3.2.1 Concurrent memory allocators 46
3.2.2 RDMA-accelerated DSM systems 48
3.3 CoRM 50
3.3.1 Memory allocation and compaction 51
3.3.2 Pointer release 57
3.3.3 Probability of compaction 57
3.3.4 Preserving RDMA access 59
3.4 Evaluation 60
3.4.1 Operations Latency 61
3.4.2 Read throughput 63
3.4.3 Compaction performance 66
3.4.4 Compaction overheads and benefits 69
3.5 Related Work 74
3.6 Summary and Discussion 74

4 Zero-copy Data Access for Apache Kafka over RDMA **75**

4.1 Motivation 76

4.2 Background on Publish-Subscribe systems 78

4.3 RDMA design for Kafka 79

4.3.1 Network Layer 81

4.3.2 Produce datapaths 82

4.3.3 Replication datapaths 87

4.3.4 Consume datapaths 89

4.4 Evaluation 91

4.4.1 The effect of RDMA on produce datapath 93

4.4.2 The effect of RDMA on replication 95

4.4.3 The effect of RDMA on consume datapath 98

4.4.4 Improving Data Processing Applications. 101

4.5 Related work 102

4.6 Summary and Discussion 103

5 Serialization-free RDMA networking in Java **105**

5.1 Motivation 106

5.2 Background on Object Serialization 108

5.3 System Overview 111

5.3.1 Object Graph Traversal 112

5.3.2 Network exchange of on-Heap Objects 115

5.3.3 Object Graph Recovery 119

5.3.4 Overlapping network and graph traversal 122

5.4 Evaluation 123

5.4.1 Serializing Java Data Structures 124

5.4.2 Accelerating applications with Naos 131

5.5 Summary and Discussion 133

6 Efficient NIC-based Authentication and Encryption for RDMA 135

6.1 Motivation 136

6.2 Background on InfiniBand Transport 137

6.2.1 IBA Memory Protection 138

6.3 Problem Definition 139

6.3.1 Desired Security Properties 139

6.3.2 Adversary Model 139

6.4 Secure RDMA System Design 140

6.4.1 Assumptions 141

6.4.2 Secure Reliable Connection Queue Pair 142

6.4.3 Header Authentication 143

6.4.4 Packet Authentication and Encryption 144

6.4.5 PD-level Protection 144

6.4.6 Extended Memory Protection 145

6.5 Implementation 146

6.5.1 Notation and Experimental Setup 147

6.5.2 Implementation of the Secure QP 147

6.5.3 sRDMA requests 149

6.6 Evaluation 150

6.6.1 Authentication performance 150

6.6.2 Evaluation modes 151

6.6.3 Latency 152

6.6.4 Bandwidth 154

6.6.5 Mixed write/read workload 157

6.6.6 Key-value store workload 158

6.7 Related Work on Securing IBA 159

6.8 Summary and Discussion 161

7 Conclusions and future work **163**

7.1 Future work 164

Bibliography **167**

1 Introduction

Modern supercomputers have made large steps in improvements in networking, computation, and energy efficiency. The ending of Moore's Law has resulted in a growing need for specialized hardware like network accelerators and FPGA. Thus, we can see the current trend to take advantage of modern hardware to build highly scalable and efficient data centers, high-performance computing, and data processing systems. New hardware includes new capabilities and offers new architectural opportunities that may require a complete redesign of the algorithms and data structures to obtain the necessary performance and satisfy the client's requirements. These technical problems are often encountered in high-performance data management systems.

Today's data management systems are designed to provide an efficient way of storing and accessing data across a network. They incorporate decades of academic and industrial research and intense software development to satisfy customer needs. Because of the enormous diversity in use-cases and access patterns, there is a multitude of different systems exploiting numerous data structures and offering different access interfaces, ranging from SQL and key-value interfaces to direct memory accesses, to users. In addition to the immense assortment of user interfaces, memory management of applications is af-

fected by the programming languages in which they are implemented. Managed languages such as Java perform automatic memory management with the use of garbage collection algorithms, thereby hindering system developers from having full control over memory structures. This profusion of memory, access, and programming models raises diverse design challenges to the efficient adoption of modern network accelerators.

For the last decades, networked systems have tried to partially offload network and data management protocols to network accelerators with the use of advanced remote direct memory access (RDMA) programming techniques. RDMA interconnects are already provided in public clouds, such as Microsoft Azure [74], Oracle Cloud [109], and Alibaba Cloud [35]. The widespread availability of fast low-cost networks with RDMA capabilities has encouraged modern database management systems to adapt RDMA for improving query performance [20, 131]. RDMA already empowers replication [173, 147, 80], index structures [177], distributed transactions [164, 31, 172], and processing of analytical workloads [91, 17]. The emergence of RDMA has further sparked interest in distributed shared memory systems that combine memory of interconnected nodes as a shared remotely-accessible memory space [47, 28, 3, 2, 106]. In-memory databases [27], caching services [54], and ephemeral storage [145] are only some examples of systems enabled by this paradigm.

Despite the success of RDMA, RDMA features have created many design challenges in protocols for memory and data management. On the one hand, RDMA-capable network controllers bring new memory access patterns that completely bypass the operating system and the CPUs of communicating machines. Therefore, applications that exploit RDMA communication have the potential to eliminate many copy operations within the application logic, further increasing performance. On the other hand, these direct memory accesses are not fully consistent with the concurrent accesses from host CPUs, thereby forcing researchers to design specialized access protocols to prevent inconsistent data accesses. Besides, the design of RDMA-capable network protocols has been mainly focused on performance rather than security, bringing potential security implications and dangers to RDMA-enabled systems.

In this dissertation, I explore and address various challenges in networked data management systems to enhance their capabilities. Particularly, I address data management challenges in key-value stores, shared message queues, and remote memory systems, managed programming languages regarding storage reliability, performance, and memory fragmentation. In addition, I explore security challenges posed by RDMA networking and propose a security extension to the existing InfiniBand protocol.

1.1 Background on RDMA network accelerators

RDMA is a mechanism allowing one machine to directly access data in the memory of remote machines across the network. Memory accesses are performed using RDMA-capable network controllers (RNICs) without any CPU intervention or context switches. The offload of memory accesses decreases CPU usage on both the initiator and the target, thereby decreasing latency. RDMA includes the concept of one-sided operations where the CPU at a target node is unaware of incoming RDMA requests.

RDMA is offered by several network architectures [10, 126, 22, 5, 4, 42]. In this study, we focus on the InfiniBand standard and its reliable RDMA connection type called *reliably connected queue pair (RC QP)*. An RC transport establishes connected queue pairs (QP) between the two communicating applications. A queue pair is a bi-directional message transport engine that empowers applications, besides reliably sending and receiving messages, to directly read or write data from remote nodes using one-sided operations. Applications make use of offloaded RDMA communication by directly posting asynchronous work requests to an RNIC, bypassing the operating system. Upon completion of a work request, the RNIC generates a corresponding completion event that is placed on a completion queue created by the application.

In this dissertation, we primarily focus on the following RDMA work requests. *RDMA Send* allows an application to send a buffer to the remote endpoint similar to a classical TCP/IP socket. The sender is unaware of where the data will be written in the remote machine. The remote RNIC will write the data to the buffer specified in the corresponding receive work request posted by the receiver. *RDMA Write* is a one-sided operation that allows the sender to write a buffer to a remote virtual address without notifying the receiving side. To notify the receiver about an incoming Write, InfiniBand supports *WriteWithImm* operation that generates a completion event at the receiver. Unlike *Send* operation, *WriteWithImm* allows the sender to choose the destination memory address. *RDMA Read* allows the initiator to read the content of a remote buffer without the involvement of the remote CPU. RNICs also support one-sided remote atomic operations that can atomically modify an eight byte value at a remote address: *Compare-and-Swap*, and *Fetch-and-Add*.

1.2 Challenges and Contributions

In this dissertation, I address the following data management challenges in the context of networked systems:

Fault Resilience in key-value stores

The first part of the thesis explores the design and architecture of a key-value store that allows users to dynamically set the level of fault resilience for each individual key-value pair while still maintaining overall consistency and without compromising efficiency. For that, we propose an efficient data management scheme, Ring, that utilizes a combination of replication and Reed-Solomon storage schemes to allow users to choose the best reliability-overhead-performance trade-off for every single data item.

Our novel data layout guarantees a stable key-to-server mapping regardless of the storage schemes that are used, thereby allowing clients to change dynamically the storage scheme of key-value pairs independently from other clients. In other words, our storage design avoids key remapping when keys are moved across storage schemes. Ring ensures the highest possible performance for finding keys with an unknown storage scheme, as clients can use a hash mapping to locate a requested key-value pair regardless of the storage scheme used.

To achieve stable key-to-node mapping in Ring, all distinct deployed storage schemes must have the same key-to-node mapping (i.e., the same key shards). For the replication scheme, we achieve that by partitioning the keyspace into a fixed number of shards. However, the conventional Reed-Solomon scheme does not support partitioning, thereby preventing the use of erasure codes in such data layout. Therefore, for erasure codes, we introduce a new storage encoding scheme called Stretched Reed-Solomon that allows us to partition data blocks into a fixed number of shards. Stretched Reed-Solomon redistributes the data chunks of Reed-Solomon codes to maintain a stable key-to-node mapping regardless of erasure code used.

Compaction in remote memory systems

In the second part, I discuss the problem of memory fragmentation in remote memory systems. Even though memory fragmentation increases memory usage of in-memory data

stores by up to 69% (e.g., Redis, MongoDB, and VoltDB) [84, 98, 176, 115, 114] and has a negative impact on their performance due to memory sparsity [84], RDMA-accelerated remote memory systems that utilize one-sided RDMA requests do not provide memory compaction and are exposed to memory fragmentation. In fact, remote objects are accessed via RDMA by specifying their virtual addresses at the remote host: if the remote host relocates an object, its virtual address might change, requiring propagating this update to the other nodes.

To address this issue, we propose CoRM, a remote memory system that exploits RDMA for fast remote accesses and supports memory compaction. The high-level idea of our compaction algorithm is to find two blocks with low utilization and copy objects from a source block to the target block, then CoRM exploits RDMA-aware memory remapping to silently move objects across physical pages, preserving their base virtual addresses and RDMA access keys. To facilitate compaction, an in-memory object is uniquely identified by the block address and the block-local object ID, allowing CoRM to move compacted objects to new offsets within the target block, but still allowing clients to find the object within the block. During compaction, CoRM attempts to store all compacted objects at the same offset as in the source block to preserve the full virtual addresses of compacted objects. Therefore, our compaction algorithm does not alter the virtual address of the majority of compacted objects, thereby preserving direct access to them via RDMA. For the rest, we propose an additional action, called pointer correction, that recovers direct virtual pointers for clients.

Zero-copy Data Access for Log-structured data stores

In the third part I explore the most efficient way of using existing RDMA features to accelerate Apache Kafka [8, 82], a publish-subscribe system, which performance is currently constrained by overheads in the existing TCP datapaths in the form of RPC infrastructure, CPU wakeup latency, and superfluous buffering of data. The design of our system, KafkaDirect, is inspired by the fact that general-purpose request processing is expensive due to excessive data copies. Since zero-copy request processing is crucial for CPU-intensive systems, such as Kafka, we remove data copies introduced by the TCP/IP stack and general-purpose request processing by offloading CPU-intensive operations to RNICs.

Our design empowers clients to write records directly to storage using RDMA. KafkaDirect

can ensure consistent writes to the same topic from multiple producers by employing RDMA atomic operations. Consumers in KafkaDirect exploit RDMA Reads to directly read records from subscribed topics and to get notified about new records, completely bypassing the CPU of Kafka brokers and thereby significantly reducing their CPU usage. Overall, our system outperforms the existing Kafka systems in terms of both bandwidth and latency for all datapaths.

Serialization-free RDMA networking in Java

In the fourth part, I explore data management and networking in the context of managed language. Java virtual machines do not allow users to directly access on-heap objects, forcing developers to employ expensive serialization libraries to extract objects and copy them to RDMA accessible buffers, thereby preventing zero-copy RDMA networking for on-heap objects. To address these questions, we developed Naos, a JVM-based library that allows objects to be sent serialization-free from a local heap to a remote one with minimal CPU involvement and over RDMA networks. As Naos eliminates the need to copy and transform objects, it offers significant speedups compared to state-of-the-art serialization libraries.

To the best of our knowledge, Naos is the first *serialization-free* communication library for JVM that allows applications to send objects directly through RDMA or TCP connections. Naos unlocks efficient asynchronous RDMA networking to JVM users hiding all the burden of low-level RDMA programming from the users, thereby facilitating the adoption of RDMA. For that, Naos solves several complex design issues such as sending unmodified memory segments across Java heaps without employing intermediate buffers, and interacting with concurrent garbage collection without compromising JVM's memory safety. For the first issue, we propose a novel algorithm that writes objects directly to the remote heap and makes them valid on the receiver's address space. For the second one, we propose techniques preventing a concurrent JVM garbage collector from moving unsent objects that may be accessed by RNIC and from accessing unrecovered received objects.

Efficient NIC-based Authentication and Encryption for RDMA

In the last part, I explore the security property of InfiniBand interconnects and propose sRDMA, a protocol that provides secure transport for RDMA networking. The main shortcoming of the current InfiniBand architecture is that it lacks any form of cryptographic

authentication or encryption. Instead, its RDMA mechanisms provide a weak form of protection by including access tokens in each message. If an adversary is able to obtain control over a machine in an RDMA network, it can fabricate and inject arbitrary packets. If the adversary can guess or acquire memory protection tokens (which are transmitted in plaintext), it can read and write memory locations that have been exposed using RDMA on any machine in the network, leading to a powerful attack vector for lateral movement in a data center network.

We propose a security extension to InifniBand protocol that provides authentication and secure channels for InifniBand-based RDMA-capable protocols. Our extension introduces secure reliable connection queue pairs that use symmetric cryptography for source and data authentication. Since symmetric cryptography introduces per-connection memory overhead and memory on network controllers is constrained, we augment our proposed mechanisms using protection domain-level keys and efficient dynamic key derivation, which eliminates the need for storing QP-level keys and drastically reduces the memory overhead on RDMA-capable NICs.

1.3 Dissertation Outline

The dissertation is organized as follows:

Chapter 2: Per-item fault resilience in key-value stores

In this chapter, we present the design and architecture of Ring, a key-value store that empowers its users to explicitly manage storage schemes, including replication and erasure codes, on a key-value pair basis.

Chapter 3: Compactable Remote Memory over RDMA

This chapter of the dissertation describes the memory fragmentation problem in data management systems and proposes how RDMA-accelerated remote memory systems can support compaction to reduce memory fragmentation. Our RDMA-aware compaction algorithm ensures strict consistency while providing one-sided RDMA accesses even to compacted objects.

Chapter 4: Zero-copy Data Access for Apache Kafka over RDMA

The material presented in this chapter focuses on the effective use of offloaded RDMA networking for log-structured storage systems. We show how to achieve true zero-copy communication that avoids intermediate buffering for appending data to and retrieving data from a shared message queue.

Chapter 5: Serialization-free RDMA networking in Java

In this chapter, we present Naos, a library that allows Java objects to be sent serialization-free across Java virtual machines with minimal CPU involvement and over RDMA networks. We show how Naos eliminates the need to transform on-heap objects, thereby offering significant speedups compared to state-of-the-art serialization libraries.

Chapter 6: Efficient NIC-based Authentication and Encryption for RDMA

In Chapter 6, we show the security shortcomings of the InfiniBand transport protocol and propose an extension to the InfiniBand architecture that enables authentication and encryption for RDMA networking. We also demonstrate how providers can implement our RDMA secure channels with minimal changes to the existing InfiniBand protocol.

Finally, in Chapter 7 we conclude the dissertation with a short summary and outline a few opportunities for future work.

1.4 Related publications

A part of the work in the dissertation has already been covered in the following publications:

- Konstantin Taranov, Gustavo Alonso, and Torsten Hoefler. Fast and Strongly-consistent Per-item Resilience in Key-value Stores. In *Proceedings of the 13th EuroSys Conference*, EuroSys'18, pages 39:1-39:14. Association for Computing Machinery, 2018 [148].

- Konstantin Taranov, Rodrigo Bruno, Gustavo Alonso, and Torsten Hoefler. Naos: Serialization-free RDMA networking in Java. In *Proceedings of the 2021 USENIX*

Annual Technical Conference, USENIX ATC'21, pages 1-14. USENIX Association, 2021 [149].

- Konstantin Taranov, Salvatore Di Girolamo, and Torsten Hoefler. CoRM: Compactable Remote Memory over RDMA. In *Proceedings of the 2021 ACM International Conference on Management of Data*, SIGMOD'21. Association for Computing Machinery, 2021 [150].

- Konstantin Taranov, Benjamin Rothenberger, Adrian Perrig, and Torsten Hoefler. sRDMA - Efficient NIC-based Authentication and Encryption for Remote Direct Memory Access. In *Proceedings of the 2020 USENIX Annual Technical Conference*, USENIX ATC'20, pages 691-704. USENIX Association, 2020 [151].

2

Per-item resilience in key-value stores

In-memory key-value stores (KVSs) provide different forms of fault resilience through basic r-way replication and complex erasure codes such as Reed-Solomon. Each storage scheme exhibits different trade-offs in terms of reliability and resources used (network load, latency, storage required, etc.). Most KVSs support only a single such storage scheme, forcing developers to employ different KVSs for different applications. In this chapter, we argue that it is possible to create one storage system that can meet the needs of numerous applications, by allowing users to alter the storage scheme of each key-value pair.

The work in this chapter explores the following research questions:

- How can the dynamic management of storage schemes of stored items improve the utilization of cluster resources?
- How can we efficiently update the storage scheme of each key-value pair?
- What applications can benefit from explicit and dynamic storage management?

To address these questions we have designed an in-memory KVS, Ring, that empowers clients to explicitly manage storage schemes for each key-value pair, in the same way, that they would manage conventional system resources such as memory or processor time, while still maintaining overall consistency and without compromising efficiency. At the heart of Ring lies a novel encoding scheme, Stretched Reed-Solomon coding, that combines hash key distributions of heterogeneous replication and erasure coding schemes. Ring utilizes RDMA to ensure low latencies and offload communication tasks. Our work demonstrates how future applications that consciously manage the fault resilience of key-value pairs can reduce the overall operational cost and significantly improve the performance of KVS deployments.

The content of this chapter has been published at the Thirteenth EuroSys Conference in 2018 [148]. The work in this chapter was done in collaboration with Torsten Hoefler and Gustavo Alonso.

2.1 Motivation

Key-value stores (KVSs) were designed as an alternative to conventional database engines to bypass the cost of imposing a schema and the scalability limitations inherent in the transactional and relational models used in database engines. KVSs can achieve outstanding performance and scalability while providing fault resilience through different storage schemes. One serious downside of existing KVSs, however, is that the degree of resilience is typically fixed per engine and/or volume. Many features may affect the choice of a particular KVS, such as consistency, performance, reliability, or memory cost. These features are usually determined by the underlying storage schemes employed by the KVS [56]. As a result, each application needs to use its own KVS to match its requirements, leading to a proliferation of engines and a significant deployment and maintenance complexity in real settings. The following table illustrates the trade-offs between performance, reliability, and storage overheads for three schemes: Simple storage (no replication), three-fold replication, and a Reed-Solomon coding scheme.

Scheme	**Reliability**	**Put Latency**	**Put Throughput**	**Storage Cost**
Simple	None	1x	1x	1x
Rep(3)	1 failure	2x	0.5x	3x
RS(3,2)	2 failures	3.4x	0.31x	1.66x

Databases have addressed parts of this problem by offering different consistency guarantees at the SQL level, allowing each application to determine the degree of consistency it wants to achieve, while the engine still preserves the correctness of the data and mediates among all applications to provide strong consistency. In the cloud, several approaches have been proposed to classify data according to its importance and assign different levels of consistency to each one of them [81, 90, 156], showing the importance of having a flexible approach to deciding how much replication is needed for each data item. With the exception of some initial ideas from research [117], we are not aware of any KVS offering similar functionality.

In this chapter, we explore the design and architecture of Ring, a KVS allowing users to set the level of fault resilience on a key-value pair basis while still maintaining overall consistency and without compromising efficiency. The key feature of Ring is that all keys live in the same *strongly consistent* namespace, and a user does not need to specify the storage scheme when looking up a key. Users can update a key's storage scheme during the key insertion or at arbitrary points during execution and still be strongly consistent.

Challenges. At a first glance, it might seem that implementing Ring requires nothing but just adding various storage schemes to a known KVS. However, this seemingly easy step entails subtle technical challenges. The first one is to ensure strong consistency across storage schemes, so that updating the storage scheme of a key remains consistent and atomic. By strong consistency we mean that updates occur atomically and requests need to be seen by all clients in the same order regardless of failures in the system. However, naively combining different KVSs makes key updates in different storages either slow or does not guarantee strong (sequential) consistency.

The second challenge is to guarantee the highest possible performance for finding keys with an unknown storage scheme. This goal requires minimizing communication during the lookup phase. Ring achieves that with a fully decentralized design. As opposed to traditional stores [168, 52, 139], it does not rely on a central server to manage the location of key items. Instead, it relies on a novel allocation scheme that guarantees a stable key-to-server mapping regardless of the storage schemes that are used.

The last challenge is to support memory efficient erasure coding schemes while maintaining strong consistency. Ring utilizes a combination of replication and optimal erasure codes to allow users to choose the best reliability-overhead-performance trade-off for every single data item.

Overview of Ring. Ring's storage abstraction relies on *memgests*, which are different storage schemes with various fault resilience and performance properties. The name *memgest* is derived from the Latin term *gestus*, which can be roughly translated as *carry or bear*. Ring's memgests range from Reed-Solomon (RS) codes to flexible forms of full replication.

Conceptual Contributions. We design a new storage encoding scheme called *Stretched Reed-Solomon* (SRS) to maintain a stable key-to-node mapping regardless of resilience levels used. SRS redistributes the data chunks of optimal RS codes and simple replication schemes among nodes such that distinct interleaved layers of erasure coding and replication always share the same distribution for key hashes. In this way, Ring allows users to change storage schemes (memgests) transparently and independently from the client. SRS coding is our key tool to address all three challenges outlined above, and we show how to combine it with versioning to guarantee strong consistency.

Technical Contributions. In addition to the SRS scheme itself, we implement Ring as a fully-functional resilient high-performance in-memory KVS. We utilize Remote Direct Memory Access (RDMA) networking to provide low latencies and offload communication tasks from the CPUs. Our extensive experimental analysis shows that Ring reaches a low remote read latency of 5 μs and an aggregate throughput of more than 1.5M put requests/sec for unreliably stored 1 KiB key-value pairs. For reliably stored 1 KiB key-value pairs, Ring achieves 800K put requests/sec for three-fold replication and more than 300K put requests/sec for $RS(3,2)$ Reed-Solomon coding.

2.2 Ring applications

There are a wide range of scenarios where per-key management of the storage trade-offs provides a powerful abstraction enabling a more efficient utilization of expensive DRAM memory. To fully utilize the overall system, Ring users can flexibly change the storage scheme for each key at any time during operation. Changing the storage scheme influences both the network and server CPU loads, which determine the overall performance of the KVS, an aspect that we will study in the paper through four use cases.

Transparent multi-temperature data management. Data in warehouses is often classified according to its *temperature*. Frequently accessed data is considered *hot* and must be available at the highest performance. Rarely accessed *cold* data permits higher

response times. Ring can transparently place hot data in high-performance replicated storage while keeping cold data in low-overhead erasure-coded storage using standard temperature-tracking schemes [67]. Ring's SRS storage scheme enables flexible temperature management by moving data from cold storage to hot storage fully transparently to users while ensuring strong consistency. It can lead to significant cost savings while maintaining the highest performance.

Heavy updates. KVSs often experience highly varying load over time. For example, items in online auctions or limited sales become very popular during the final stages of the sale. The last seconds of an auction are usually the ones of highest interest for a bidder and the system may receive millions of requests per second. A crash and the corresponding period of unavailability may be disastrous, even if the data is stored reliably. The designer of an online auction with heavy updates can use Ring to move items to high-performance, less reliable storage when a high workload is observed to increase throughput. Even if it seems counter-intuitive, the overall reliability is not reduced, for two reasons: First, the data is only stored less reliably for a short period of time, thus reducing the probability of data loss. Second, Ring supports *versioning*, allowing each key to have multiple versions in different storage schemes, preserving previous reliable copies of the data.

Importance of the data. The importance of data may change over time according to the intrinsic nature of the data. For instance, in iterative algorithms such as PageRank, the time to recover data increases as the computation progresses because losing data at a late stage requires expensive recomputation from the start. In other words, the intermediate page ranks at iteration $i + 1$ are more important than the ones at iteration i. In general, Ring can be useful for algorithms where the temporary data has to become persistent, since it dynamically increases the reliability of given key-value pairs.

Temporary blob storage. Our last use case concerns typical cloud storage schemes providing write-commit or write-modify-commit patterns. Such patterns are typical in block blobs on Azure Storage and others, where users may upload blobs and after that decide on whether to store them persistently or not [103]. For example, many services for uploading pictures allow users to apply filters and then either commit or discard the changes. Blobs are deleted by the session management if they have not been committed within a predefined time. Objects should be stored in less reliable, high-performance memgests before a final decision is made on their persistence. This use case generalizes to other user-facing storage systems, with Ring providing a convenient interface to manage them.

2.3 Storage schemes

We now explain how Stretched Reed-Solomon (SRS) codes are built and their advantages over classical RS codes. First, we briefly summarize key aspects of the replication (Rep) and Reed-Solomon (RS) schemes required to build SRS codes. In all our analyses, we assume a standard fail-stop failure model [137].

2.3.1 Replication

Replication is the simplest and most widely used approach for fault tolerance [165]. Numerous methods for data replication such as primary-backup replication [26], quorum-based replication [68, 23], and chain replication [160] exist. Primary r-fold replication provides simple reliability and availability, as any copy of the data can be read independently, but causes an $(r-1)$-fold increase in memory overhead. Strong consistency is often provided by a distinguished leader that is responsible for serving client requests. To commit a put request, the leader has to replicate the request to a majority of nodes. Therefore, we consider that availability and reliability of the r-fold quorum-based replication are guaranteed when less than or equal to $\lfloor \frac{r-1}{2} \rfloor$ nodes are faulty. Conversely, basic fully synchronous replication can tolerate $r-1$ failures, but the unavailability in case of failures is higher because of the synchronous communication with worker nodes.

2.3.2 Reed-Solomon coding

The alternative, (k, m) partial replication through erasure coding (e.g., Reed-Solomon), uses m additional parity blocks to secure k data blocks on different servers [119]. Maximum distance separable erasure codes can tolerate up to m simultaneous failures in a group of $k+m$ blocks, which is the theoretically optimal storage overhead [119]. RS codes achieve the maximum distance separable property and provide a flexible choice of parameters k and m. The memory overhead is only proportional to the expected number of failures m, but requires accessing at least k data blocks during recovery. The performance of erasure coding is affected by faults, as the lost data cannot be immediately accessed and should be recovered first. Therefore, systems using erasure codes are less available than ones using replication schemes in the presence of failures.

A common way to use RS schemes is to split data of size C into k blocks of size C/k. According to $RS(k, m)$ encoding scheme, m additional parity blocks are calculated from the k original data blocks. These blocks are stored on separate nodes and are grouped to form a stripe with k data blocks (denoted by $[D_1, ..., D_k]$) and m parity blocks (denoted by $[P_1, ..., P_m]$). We refer to nodes which store parity blocks as parity nodes, and nodes which store data blocks as data nodes. We also refer to data on data nodes as primary data, and data on parity nodes as parity data. This arrangement allows recovery from any combination of up to m simultaneous failures. The choice of the parameters k and m influences the fault tolerance, memory overhead, and recovery time. In the case of failures, the decoding operation reads any k out of the $k + m$ blocks to recover the lost blocks. When failures are frequent, the system performance degrades dramatically due to data recovery.

An RS encoding operation can be represented as a matrix-vector multiplication where the vector of k data blocks is multiplied by a particular matrix $H = \begin{bmatrix} I \\ \hdashline G \end{bmatrix}$ of size $(k+m) \times k$, (see Eqn. (2.1)). Here, I is the identity matrix and G is called the generator matrix, and yields the maximum distance separable property.

$$\underbrace{\begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 1 \\ g_{11} & g_{12} & g_{13} & \cdots & g_{1k} \\ g_{21} & g_{22} & g_{23} & \cdots & g_{2k} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ g_{m1} & g_{m2} & g_{m3} & \cdots & g_{mk} \end{bmatrix}}_{H: k+m \times k} \times \begin{bmatrix} D_1 \\ D_2 \\ \vdots \\ D_k \end{bmatrix} = \begin{bmatrix} D_1 \\ D_2 \\ \vdots \\ D_k \\ P_1 \\ P_2 \\ \vdots \\ P_m \end{bmatrix} \tag{2.1}$$

The matrix G can be constructed from a Vandermonde matrix ($g_{ij} = j^{i-1}$), where the elements are calculated according to Galois Field (GF) arithmetic [43]. In the GF with 2^n elements, where n is a positive integer, addition is equivalent to a bitwise XOR operation. Multiplying blocks by a scalar constant (such as the elements of H) is equivalent to multiplying each GF word component by that constant. The matrix G ensures the main property of the matrix H: any set of k rows of the matrix H is linearly independent. It

means that the data can be recovered if at least k blocks out of $k + m$ data and parity blocks are available.

Recovery

Lost data blocks can be reconstructed by solving the reduced system of linear equations obtained by removing the rows corresponding to lost blocks in Eqn. (2.1). Reconstruction of a parity block is the same as an encoding operation, and involves multiplying the corresponding row of H by the data vector. Data block reconstruction takes two steps. The first step is to calculate a decoding matrix. The decoding matrix is built choosing any k linearly independent surviving rows of H, and then taking the inverse of them. The second step involves multiplication of the previously selected combination of data and parity blocks by the corresponding row of the decoding matrix to the missing block.

Update

If data on any server is updated, all corresponding parity blocks must be updated as well. Fortunately, it is not necessary to recalculate the whole data in parity blocks, as only recomputation of the concerned pieces of information is required. An update operation first calculates the difference between the old and the new data items, then replicates the update operation to parity nodes. Finally, at the parity nodes, the stored parity block is XORed with the update operation multiplied by a corresponding coefficient from the encoding matrix H.

2.3.3 Stretched Reed-Solomon coding

In this section, we introduce our Stretched Reed-Solomon (SRS) codes, which are based on RS codes. A common way of load balancing $RS(k, m)$ codes among k data nodes is to put an object with a key to data node $i = (h(key) \mod k)$, where $h(key)$ is a hash function. The major problem in this mapping, and other distribution schemes, is the coupling between the hash key distribution and the number of data blocks k. For instance, a storage system based on $RS(2, 1)$ has 2 primary data nodes, and thus has 2 key shards, whereas $RS(3, 1)$ includes 3 data nodes and 3 key shards. As a result, they cannot both be accessed with the same key-to-node mapping. Even worse, when the storage scheme is

changed to a different k, the keys need to be remapped and migrated. Hence a new erasure code is needed to avoid key remapping when keys are moved across storage schemes.

The main idea behind SRS codes is to ensure the same key-to-node mapping for a range of RS codes with different k. This is a key feature of Ring that enables efficiently locating a node responsible for any key with the unified mapping, irrespective of the storage scheme (memgest) chosen for the data.

Derivation

SRS codes are defined by parameters k, m, and s. The parameters k and m are inherited from RS codes such that $SRS(k, m, s)$ codes apply a $RS(k, m)$ coding algorithm to the data. Yet, instead of storing k data blocks on k nodes, the blocks are spread or stretched over s machines ($s \geq k$). As a result, we have s data nodes and m parity nodes, but the data on them is encoded according to $RS(k, m)$. Note that $SRS(k, m, k)$ is identical to $RS(k, m)$.

Having $s \geq k$ data nodes for all RS codes enables them to share identical key-to-node mappings regardless of the erasure codes. For instance, if we stretch $RS(2, 1)$ over 3 data nodes and obtain $SRS(2, 1, 3)$ as in Figure 2.1, then it will have the same number of data nodes as $RS(3, 1)$. Hence, $SRS(2, 1, 3)$ and $SRS(3, 1, 3)$ can share a key-to-node mapping, and their data nodes can be stored on the same physical machines. When resilience requirements for a key is updated from $SRS(2, 1, 3)$ to $SRS(3, 1, 3)$, the key can be moved locally from one coding scheme to the other, since the masters of two schemes reside on the same physical node.

We build a family of $SRS(k, m, s)$ codes as follows:

---

1. Encode data according to $RS(k, m)$ coding scheme with k data blocks and m parity blocks.
2. Compute the least common multiple (l) of k and s.
   $l = lcm(k, s)$
3. Divide the original data into l blocks: $\left[\tilde{D}_1, ..., \tilde{D}_l\right]$.
4. Distribute l data blocks over s data nodes such that each data node stores l/s blocks.
5. Parity blocks are stored on m nodes as in $RS(k, m)$.

The encoding matrix H for $RS(k,m)$ from Eqn. (2.1) can be expanded to a stretched matrix H_{exp} of size $l + \frac{lm}{k} \times l$:

$$\underbrace{\begin{bmatrix} I_{\frac{l}{k}} & 0_{\frac{l}{k}} & 0_{\frac{l}{k}} & \dots & 0_{\frac{l}{k}} \\ 0_{\frac{l}{k}} & I_{\frac{l}{k}} & 0_{\frac{l}{k}} & \dots & 0_{\frac{l}{k}} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0_{\frac{l}{k}} & 0_{\frac{l}{k}} & 0_{\frac{l}{k}} & \dots & I_{\frac{l}{k}} \\ g_{11}I_{\frac{l}{k}} & g_{12}I_{\frac{l}{k}} & g_{13}I_{\frac{l}{k}} & \dots & g_{1k}I_{\frac{l}{k}} \\ g_{21}I_{\frac{l}{k}} & g_{22}I_{\frac{l}{k}} & g_{23}I_{\frac{l}{k}} & \dots & g_{2k}I_{\frac{l}{k}} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ g_{m1}I_{\frac{l}{k}} & g_{m2}I_{\frac{l}{k}} & g_{m3}I_{\frac{l}{k}} & \dots & g_{mk}I_{\frac{l}{k}} \end{bmatrix}}_{H_{\text{exp}}:l+\frac{lm}{k}\times l} \times \begin{bmatrix} \tilde{D}_1 \\ \tilde{D}_2 \\ \vdots \\ \tilde{D}_l \end{bmatrix} = \begin{bmatrix} \tilde{D}_1 \\ \tilde{D}_2 \\ \vdots \\ \tilde{D}_l \\ \tilde{P}_1 \\ \tilde{P}_2 \\ \vdots \\ \tilde{P}_{\frac{lm}{k}} \end{bmatrix} \tag{2.2}$$

where each element is a matrix: I_n is the identity matrix of size n, and 0_n is the zero matrix of size n.

For $RS(k,m)$, H_{exp} is a coding matrix corresponding to encoding l data blocks using an $RS(k,m)$ code. It distributes l data blocks among k data nodes so that each node stores l/k data blocks. It also calculates lm/k parity blocks and distributes them among m parity nodes so that each node stores l/k parity blocks each. The matrices H and H_{exp} are equivalent in terms of data encoding, since they produce the same output. The matrix H_{exp} can also be calculated as entry-wise product of H and an expansion matrix E:

$$H_{\text{exp}} = H \circ E = E \circ H, \tag{2.3}$$

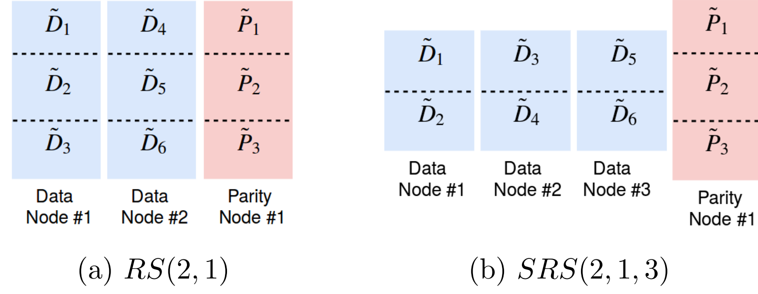
where $E_{ij} = I_{\frac{l}{k}}$, H and E are of the same dimensions.

According to classical Reed-Solomon the i-th data node D_i and j-th parity node P_j are comprised of the following blocks from Eqn. (2.2):

$$D_i = \left[\tilde{D}_{\frac{(i-1)l}{k}+1}, ..., \tilde{D}_{\frac{il}{k}}\right], \qquad P_j = \left[\tilde{P}_{\frac{(j-1)l}{k}+1}, ..., \tilde{P}_{\frac{jl}{k}}\right]$$

To obtain the SRS code, we reassign data blocks to s nodes instead of k. Thus, every data node is responsible for l/s instead of l/k chunks of data, whereas parity nodes are not involved in stretching and are kept the same. Therefore, nodes of $SRS(k,m,s)$ store data as follows:

$$D_i = \left[\tilde{D}_{\frac{(i-1)l}{s}+1}, ..., \tilde{D}_{\frac{il}{s}}\right], \qquad P_j = \left[\tilde{P}_{\frac{(j-1)l}{k}+1}, ..., \tilde{P}_{\frac{jl}{k}}\right]$$



(a) $RS(2,1)$ (b) $SRS(2,1,3)$

Figure 2.1: Block distribution for $RS(2,1)$ and $SRS(2,1,3)$

Example of building SRS(2, 1, 3). In this paragraph we introduce an example of creating Stretched Reed-Solomon over three data nodes. The coding matrix H for $RS(2,1)$ is presented in Eqn. (2.5). The least common multiple of 2 and 3 is 6, so the data is divided into 6 blocks in order to spread it over 3 nodes. Afterwards, we expand the coding matrix of $RS(2,1)$ to 6 blocks by multiplying it by E according to Eqn. (2.3), where $E_{ij} = I_3$ (see Eqn. (2.5)). As a result, the data is encoded in Figure 2.1(a) as follows:

$$\tilde{P}_1 = \tilde{D}_1 \oplus \tilde{D}_4 \qquad \tilde{P}_2 = \tilde{D}_2 \oplus \tilde{D}_5 \qquad \tilde{P}_3 = \tilde{D}_3 \oplus \tilde{D}_6 \tag{2.4}$$

In $RS(2,1)$ every server is responsible for $l/k = 6/2 = 3$ blocks as shown in Figure 2.1(a). In $SRS(2,1,3)$, we assign $l/s = 6/3 = 2$ data blocks for every data node (Figure 2.1(b)), and the data on them is encoded with the matrix from Eqn. (2.5) as in Eqn. (2.2).

$$H_{\text{exp}} = \underbrace{\begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix}}_{H} \circ \underbrace{\begin{bmatrix} I_3 & I_3 \\ I_3 & I_3 \\ I_3 & I_3 \end{bmatrix}}_{E} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix} \tag{2.5}$$

Properties of Stretched Reed-Solomon codes. An important feature of $SRS(k,m,s)$ is that it preserves the coding properties of the original $RS(k,m)$. First, restoring a lost block still requires collecting k corresponding blocks over $s+m$ nodes, since data is still encoded according to $RS(k,m)$. Second, when a data server receives a put request, the request has to be propagated to m parity nodes. It however leads to memory imbalance as a parity server is responsible for more data than a data node. Furthermore, it can be observed that $SRS(k,m,s)$ can tolerate at least m and sometimes more simultane-

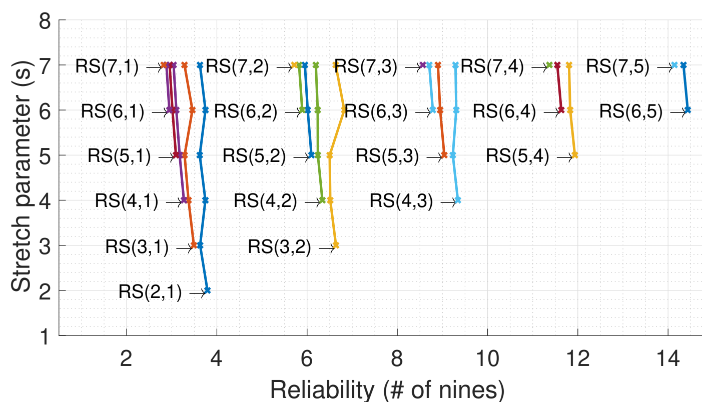


Figure 2.2: Reliability of SRS codes with different parameters.

ous failures. For instance, $SRS(2, 1, 4)$, can tolerate two simultaneous failures when two independent data servers are failed.

The main benefit of SRS codes is that we can combine many distinct storage schemes that fit the node layout, and access them with a unified key-to-node map. Keys can hence be transparently moved from one scheme to another, since all primary data of all schemes are present at each node. For example, with $s = 4$, Ring can support the following families of SRS coding schemes: $SRS(2, m, 4)$, $SRS(3, m, 4)$, and $SRS(4, m, 4)$, where m is an arbitrary integer greater than one. In practice, the number of parity nodes in RS schemes is bounded by the number of data nodes, i.e., $m < k$. Taking that into account, the total number of different erasure coded storage schemes with given s equals to $\frac{s(s-1)}{2}$. Besides, we can include replication schemes by partitioning them into s shards in order to have the same key hash distribution as $SRS(k, m, s)$. It enables, even for moderate s, a very large number of possible storage schemes (memgests) in a single KVS.

Reliability and Availability of SRS codes. In this paragraph we show that $RS(k, m)$ and $SRS(k, m, s)$ provide a comparable level of resilience. $SRS(k, m, s)$ distributes the data across more nodes and thus seems more liable to failures. However, the sparser distribution leads to a lower data loss after a node failure. We investigate the overall reliability using Markov models for reliability and availability.

Figure 2.2 indicates reliabilities of RS codes from which we derive stretched versions. The diagram shows a vertical line for each stretched code. The lowest point with the label is $RS(k, m) = SRS(k, m, k)$ and the connected points above are different stretching factors. The results show that stretching maintains approximately the same level of reliability. For example, the family of $SRS(3, 1, s)$ codes provides reliability around 3.5 nines.

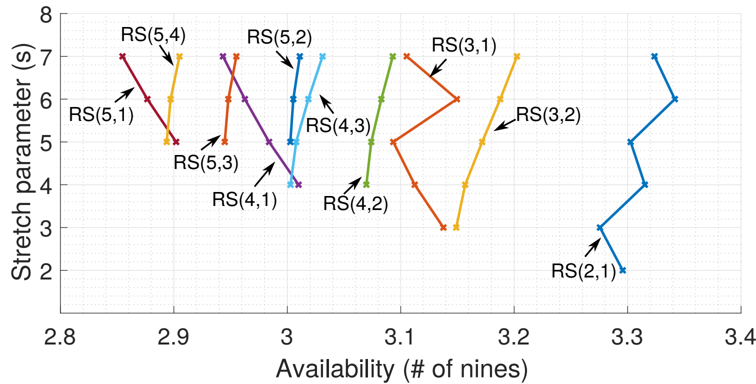


Figure 2.3: Availability of SRS codes with different parameters.

An interesting observation is that the reliability sometimes increases when the data is stretched. For instance, $SRS(3, 2, 6)$ is more reliable than $RS(3, 2)$. One reason for that is that each data node of a stretched version stores less data than a data node of the original one, which results in less data requiring recovery in the event of data node failure. For example, if the system stores 600GiB, in case of a data node failure $RS(3, 2)$ loses 200GiB and $SRS(3, 2, 6)$ loses only 100GiB. Faster recovery increases reliability because the system can tolerate more failures if the data has been recovered before the next failure. In addition, $SRS(k, m, s)$ is sometimes able to tolerate more than m simultaneous failures. It happens when failed data nodes store independent data blocks, still allowing the system of equations in Eqn. (2.2) to be solved without these nodes.

Figure 2.3 indicates estimated availabilities of RS and SRS codes. Lines represents different stretching factors of $SRS(k, m, s)$ codes that share the same parent code $RS(k, m)$. It can be observed that all RS schemes and their stretched variations have availability less than 3.4 nines. In addition, the number of nodes in the stripe decreases the availability. Maximal availability has been observed for the family of $SRS(2, 1, s)$ codes and stands at approximately 3.35 nines.

2.4 System architecture

After establishing the concepts of Stretched Reed-Solomon coding, we proceed to describe the architecture and the key components of Ring. We will discuss how Ring combines SRS coding to unify a flurry of RS coding and replication modes into a high-performance, fault-tolerant, strongly-consistent in-memory KVS.

2.4.1 API

Ring allows clients to access objects identified by a key through the three standard KVS operations `get`, `put`, and `delete`. The `put(key, object, memgestID)` operation determines where the object should be placed based on the associated `key`, and writes the object to the memgest of Ring that determines the storage mode. Ring also supports the standard `put(key, object)` call which stores the data in a configurable default memgest. The `get(key)` operation locates the object master associated with the `key` in the storage system and returns the object. The `delete(key)` operation deletes the object associated with a given `key`. Ring also supports `move`, which provides additional flexibility in storing the data by allowing objects to be moved between memgests.

```
/* Conventional KVS requests */
object_t* get(const key_t)
int put(const key_t, const object_t*)
int delete(const key_t)
/* Resilience management */
int put(const key_t, const object_t*, const id_t)
int move(const key_t, const id_t)
/* Storage scheme management */
id_t createMemgest(const descriptor_t*)
int deleteMemgest(const id_t)
int setDefaultMemgest(const id_t)
descriptor_t* getMemgestDescriptor(const id_t)
```

To manage memgests, clients can dynamically add and remove memgests from Ring. Clients can create a memgest by sending a `createMemgest(descriptor)` request. The `descriptor` contains parameters of the storage scheme. It adds additional flexibility to Ring, since users may tune the KVS by deploying different resilience levels. Memgests can be removed with the `deleteMemgest` command. Finally, the default storage scheme for new keys can be specified with `setDefaultMemgest(memgestID)`.

2.4.2 Data layout, memgests

Ring's storage abstraction relies on memgests, which are different storage schemes with various resilience, overhead, and performance properties. Each memgest corresponds to either

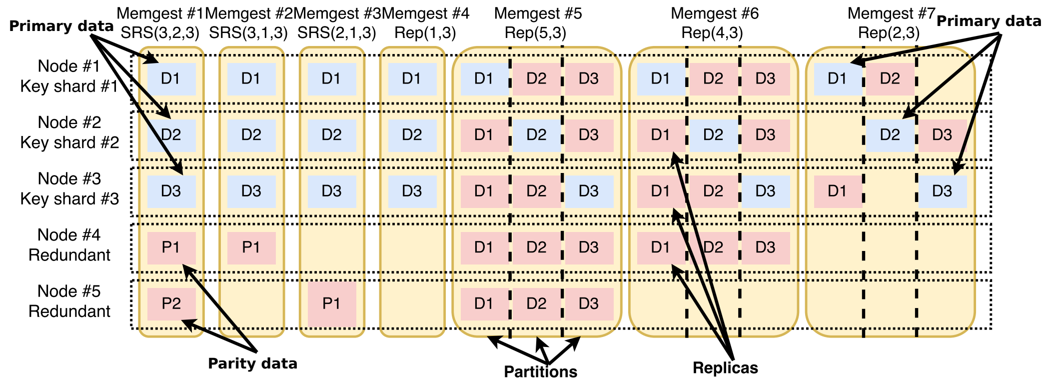


Figure 2.4: Various memgests for a 5 node system with 3 coordinators and 2 redundant nodes. The group provides 7 resilience levels: $SRS(3,2,3)$, $SRS(3,1,3)$, $SRS(2,1,3)$, $Rep(1,3)$, $Rep(5,3)$, $Rep(4,3)$, and $Rep(2,3)$. Note that stretching only affects the (blue) data blocks and parity or replica blocks can be allocated arbitrarily.

erasure coded $SRS(k,m,s)$ or replicated $Rep(r,s)$ schemes. An $SRS(k,m,s)$ memgest contains s data nodes that handle requests to data blocks and m parity nodes that receive update requests from the data nodes. A $Rep(r,s)$ memgest contains s data partitions which are replicated r times (Figure 2.4). Ring also supports memgests that are not fault-tolerant and have no additional storage overheads ($Rep(1,s)$). They can be used for storing temporary or recomputable data. The unreliable memgest offers highest update performance because it does not replicate `put` requests and can immediately acknowledge them.

A set of memgests which share the same number of key shards s constitutes a memgest group with s coordinator nodes and d redundant nodes (Figure 2.4). We refer to a server with an assigned shard as a coordinator node, since it coordinates the shard and all memgests sharing the shard. The parameter d is also an upper bound on the number of parities m in $SRS(k,m,s)$ memgest, and $s+d$ is an upper bound on the replication factor r in $Rep(r,s)$ memgest. Ring is configured for particular s and d parameters and does not allow having families with other parameters. However, our SRS codes allow transforming any storage scheme to have s data shards and build the memgest group from any number of unified storage schemes.

Ring has a distinguished leader, which is responsible for processing `createMemgest` requests. It decides how to distribute primary and parity data across nodes, and then inform other nodes about its decision.

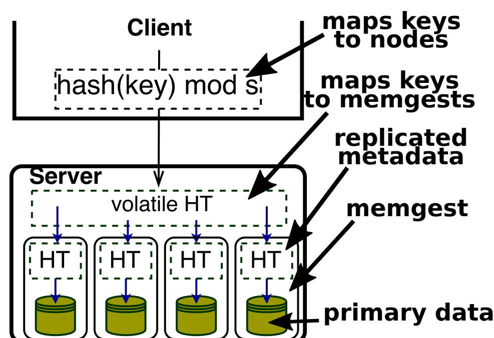


Figure 2.5: Request direction mechanism. The figure depicts only the server's coordinator-side data, i.e. without parity and replica related data.

We use a simple key-to-node mapping $i = (h(key) \mod s)$ to partition the whole key range into s shards, where each coordinator node handles one key shard within a memgest group. Thus, on a single node, *memgests share the same key shard*, but with different resilience and performance requirements (blue rectangles in Figure 2.4). As a result, our design allows `get` and `put` requests to a certain key to be performed by a single node only. The key thus suffices to directly retrieve data despite the fact that data can be stored in one of multiple memgests. Moreover, our storage design avoids the need for distributed transactions when the data moves across memgests since all required data is stored locally. In Ring every object has a version (see Section 2.4.3), which is incremented when the object is modified or moved across memgests. Each coordinator is responsible for assigning versions for all objects in its shard. It helps to design a strongly consistent KVS, where only one instance of the key of a certain version exists across all memgests.

When a client sends a KVS request, it first applies $i = (h(key) \mod s)$ to map a key to node i, which is responsible for storing the key as in Figure 2.5. Afterwards, the request is performed locally at the requested node. For example, Figure 2.5 depicts a data server that supports 4 different resilience levels. When a server receives a `get` request, it first looks the requested key up from a volatile hashtable, which maps the key to the list of pairs $\langle version, memgestID \rangle$. Each coordinator node only has keys in its volatile hashtable that are mapped to it by $(h(key) \mod s)$ mapping. The volatile hashtable is used to quickly retrieve the *memgestID* of the memgest that stores the highest *version* of the object. Then the requested object is looked up from that memgest using $\langle key, version \rangle$ pair.

Querying of a memgest is done through its metadata hashtable. The metadata hashtable is a part of each memgest, and is replicated to survive failures (except the unreliable

memgest). In contrast, the volatile hashtable that solely acts as an interface to the memgests is not replicated at all. It can be reconstructed by combining metadata hashtables of all local memgests. During normal operations the volatile hashtable is kept consistent with the memgests' hashtables.

2.4.3 Strong consistency

Strong consistency requires Ring to employ a range of techniques to ensure existence of only one instance of a key of a certain version across all memgests. First of all, the volatile hashtable and all metadata hashtables are write-ahead on the master node, that is all modifications are written to them before they are committed. The write-ahead approach does not violate consistency since, in case of a data node failure, only committed entries will be recovered. We also postpone requests that read uncommitted objects. Second, Ring exploits *versioning* and increments the key version when a key migrates across memgests. Ring retrieves the highest version of the key (even uncommitted) and continues writing with a higher version. Old versions are removed from the system periodically. It can be tuned to trigger removing of old versions of a key after every committed put request to it. It is the main mechanism for strong consistency in case of failures. It prevents having two distinct copies of the key with the same version number after failures, which would lead to an inconsistent state.

Write-ahead and versioning approaches allow serving requests to distinct memgests independently without waiting. For instance, if multiple clients want to put new values to the same key simultaneously, then their requests can be served together and independently from each other. An interesting case is when two clients put values to the same key but to different resilience levels. Since Ring allows committing the different versions of keys independently, then higher versions may be committed earlier than lower versions. It can happen when one client puts to a fast storage scheme, while another client is writing the same key to slower storage scheme. The diagram in Figure 2.6 illustrates this case for clients **A** and **B**. It also shows that client **C** gets the highest value committed by **B** regardless the success of client **A**'s request. In other words, the highest version depends on the last writer only.

A request becomes committed when it is replicated within the requested memgest; hence puts to unreliable memgests are always committed immediately. Each memgest has a special replicated log to propagate updates generated from client requests within itself.

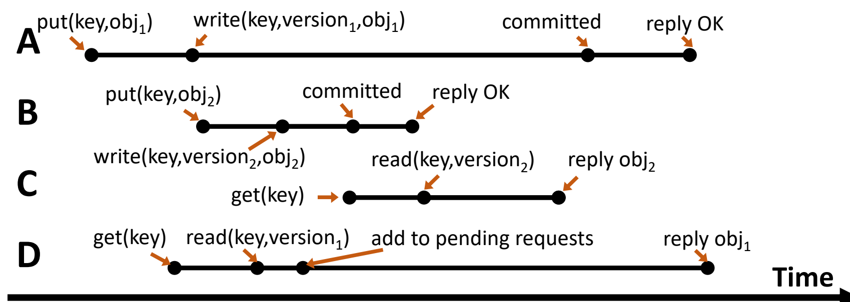


Figure 2.6: Put/get scenario with multiple clients.

To provide strong consistency, it is allowed to get data from committed entries only. Thus, the response to the client will be postponed until the requested entry is committed, as with client **D** in Figure 2.6. When the request from **D** was received, the highest version of the requested key was 1. Ring read the uncommitted version of the object and prepared a reply to the client **D**. The reply was sent once the version 1 had become committed. To ensure the proper order, the metadata hashtables store a list of pending get requests and *committed* flag for each object.

$$key, version \rightarrow data, length, \underbrace{committed, requests}_{volatile}$$

This information is volatile and can be lost in case of failures. When an entry becomes committed, the flag is flipped and all pending replies are sent. It ensures that a get request returns the version (even uncommitted at that moment) that was the highest when the request was received by Ring. Ring handles `move(key, memgestID)` requests similarly because the object has to be moved from the memgest with the highest version. Therefore, the `move` request will also be postponed if the requested object is not durable.

When clients send a `move(key, memgestID)` request, the key has to be moved from one memgest and written to another atomically, so partial updates have to be prevented. Here we benefit from a feature of Ring's design: the coordinator of memgests which are responsible for the same key is situated on a single physical machine due to the SRS coding. It allows us to avoid expensive locking and distributed transactions. In case of failures, Ring recovers all recoverable versions of keys from multiple memgests. Ultimately, several resilience requirements can be satisfied at almost no cost, since all storage management can be performed locally.

2.4.4 Erasure coded and replicated memgests

All memgests share a similar structure: they have a replicated log to replicate updates, replicated metadata hashtable, and the actual data, which is stored separately and treated according to its storage scheme.

Separating metadata from actual data provides a wide range of advantages. Firstly, the metadata suffices to serve `delete` requests. Secondly, it allows memgests to recover in two steps: metadata recovery, data recovery. What is more, data recovery can be postponed and only recovered on demand which is quite important for expensive erasure codes. Zhang et al. [174] proposed a similar approach for erasure codes. They replicated metadata of keys with a primary-backup replication scheme while actual values are erasure coded.

The `put` operation induces an update of the data of a memgest on the coordinator and redundant nodes. The coordinator thus replicates requests within the memgest. Coordinators of erasure coded memgests generate special parity updates and replicate them to m parity nodes to commit the operation, whereas the special processing of the request is not required for replication scheme, and can be replicated immediately. For replication, we implement quorum-based replication [122], where requests are replicated on the majority of nodes within a memgest to ensure their durability. The remaining nodes in replication scheme are updated asynchronously. Once the entry is properly replicated to redundant nodes, the put is committed, and the coordinator replies to the client with an acknowledgment.

Get requests do not alter the data, so they should not be replicated. To ensure that get requests do not return stale data, the coordinator node has to periodically verify its role in the system by reading the configuration from a replicated state machine [122].

2.4.5 Balancing

One issue caused by our design is that nodes within a memgest group occupy different amounts of main memory (see Figure 2.4). Unfilled rectangles represent unbalance in memory usage. It is mainly caused by the design of SRS memgests, where data is stretched over many servers. Additionally, every parity node stores metadata of all data nodes from the same SRS coding stripe. Therefore, a parity node stores more metadata than a data node. In our system, parity nodes are also responsible for recovering lost data and parity blocks and therefore require more memory to store blocks under recovery and recovery

metadata. Regarding workload, as parity nodes only need to participate in put operations, they may become idle for get-mostly workloads. In contrast, for put-mostly workloads, the parity nodes may become busy and may become a bottleneck of the KVS. Finally, during node failures, parity nodes are overloaded by data recovering processes.

To resolve these issues, we can create many memgest groups and assign them round-robin to fixed nodes, as has been done in [174]. It requires creating $s + d$ memgest groups, where s is the number of shards and d is the number of redundant nodes, since there are $s + d$ different ways of rotating the single memgest group. As we mentioned before, we use sharding to distribute keys among coordinators. Thus to support multiple memgest groups we also partition shards into $s + d$ memgest groups. It allows balancing workload and memory on each node. The shards on one node belong to different memgest groups, therefore a single node failure leads to a data loss on each memgest. However, since parity nodes are spread evenly across machines, the recovery workload also will be uniformly balanced.

2.4.6 Membership and handling failures

To deploy Ring, at least $s + d$ nodes are required, where s and d are coordinators and redundant nodes, respectively. Nonetheless, the overall system is comprised of $s + d + n$ machines, where n stands for *spare* nodes as in Figure 2.7. Thus, there are two types of nodes in the system: some that take part in serving requests and some that do not. Spare nodes are always ready to replace a failed node and immediately handle the requests for the node. The system has a leader, which is responsible for membership of nodes and, in case of failures, for reassigning the roles of failed nodes to healthy spare ones. The leader is elected according to a leader election protocol [122].

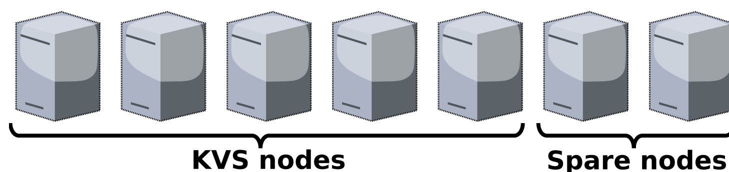


Figure 2.7: The system consists of spare nodes and KVS nodes.

While in spare mode, spare nodes incur minimal memory and CPU load. They use memory only for sending and receiving heartbeats to check membership and the log that replicates requests for changing roles. Once the leader recognizes that a node crashed, it replicates an

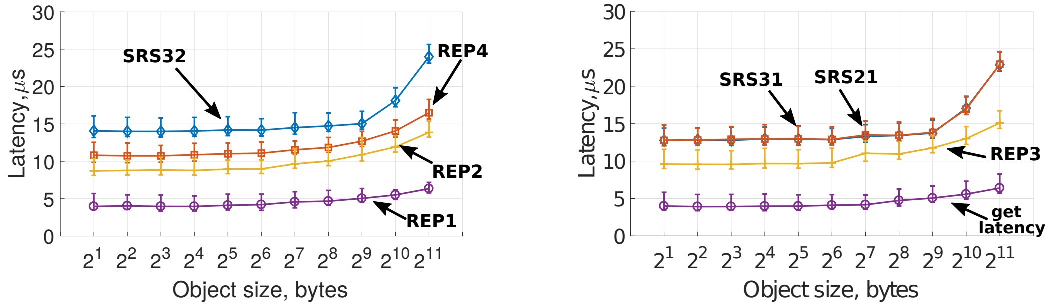
entry over the log, which consists of the new responsibilities for all of the nodes. Therefore, all servers know about all changes in the system.

To change the role and start processing requests, a spare node reads all necessary metadata from alive nodes. Once the metadata on the node is recovered and the volatile hashtable built, it starts providing services while performing data recovery in the background. If the requested data is lost, it will be recovered with an on the fly recovery algorithm with high priority. For replicated memgest, it will request a copy of the requested data from any available replica. For erasure coded memgest, it will start an online decoding algorithm similar to one introduced in [174]. Data node sends a recovery request to the parity node responsible for the lost block. Then the parity node starts block recovery by collecting k available corresponding blocks from a coding stripe according to $RS(k, m)$ and decoding them. Finally, the parity node sends the recovered block to the data node initiated the recovery.

Clients access the data by sending requests using a hash function to determine the required node. If a data node has failed and a request is not answered in a predefined period of time, clients re-send the request through multicast. The request will be serviced only by the node that is responsible for the requested key. The clients will then communicate with the new data node directly.

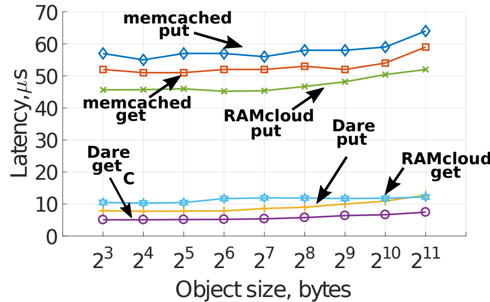
2.5 Evaluation

In this section, we evaluate the performance of replication and erasure coding approaches on RDMA networks. We use a 12-node InfiniBand cluster: each node has an Intel E5-2609 CPU clocked at 2.40GHz and WDC WD5003ABYX-01WERA1 internal hard drives. The cluster is connected with a single switch using a single Mellanox QDR NIC (MT27500) at each node. The nodes are running Linux, kernel version 3.18.14. Replication and erasure coding are implemented in C and rely on the following libraries: libibverbs, an implementation of the RDMA verbs for InfiniBand, and libev, a high-performance event loop; Jerasure [121], a library in C that supports erasure coding in storage applications; and GF-Complete [120], a library for Galois Field arithmetic. Each server is single-threaded, but can be potentially multi-threaded, e.g., by partitioning keys and assigning threads to partitions.



(a) Put latency of SRS32, REP4, REP2, REP1.

(b) Put latency of SRS31, SRS21, REP3, and get latency.



(c) Put and get latecy of baselines.

Figure 2.9: Latency of put and get requests for Ring and other systems.

2.5.1 Latency

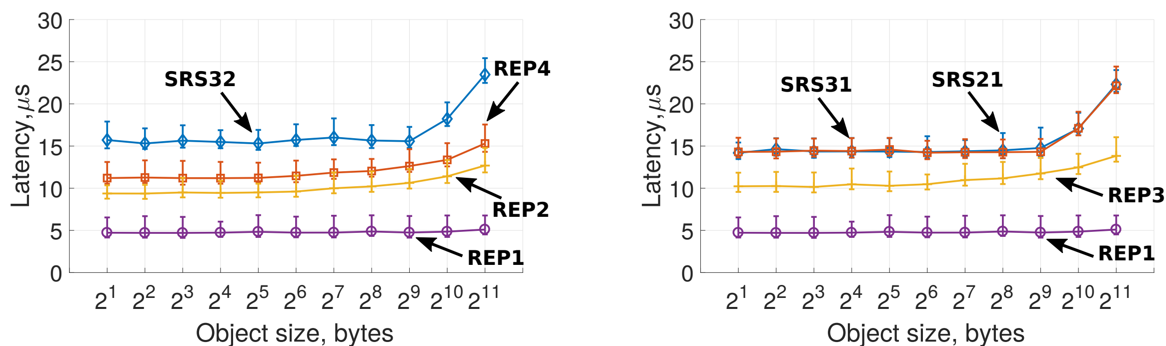
Ring is designed as a low latency KVS. Figure 2.9 shows the latency of put and get requests (split into two figures for readability). We deployed Ring on 5 nodes with 7 memgests: $SRS(3,2,3)$, $SRS(3,1,3)$, $SRS(2,1,3)$, $Rep(4,3)$, $Rep(3,3)$, $Rep(2,3)$, and unreliable $Rep(1,3)$. Since they all share the parameter s, which is equal to 3, we label them on graphs as SRS32, SRS31, SRS21, REP4, REP3, REP2, and REP1, respectively. In the benchmark, a single client gets and puts objects of varying size to/from the system. Each measurement is repeated 5,000 times, the figure reports the median and the 90th percentile. According to our implementation the get latencies of all memgests are the same, therefore we plot only one line in the figure for all studied schemes. It can be explained by all memgests using the same algorithm for retrieving data: They first ensure that they are allowed to reply to get requests by periodically checking the current node configuration, then respond to the client.

The main difference lies in put requests: The latency for $SRS(2,1,3)$ is the same as for

$SRS(3,1,3)$, regardless of whether they share the same number of coordinator nodes s in storage scheme. The reason is that they have to replicate update to one parity node only. $SRS(3,2,3)$ has the highest latency among the system because the data nodes are responsible for calculating updates and replicating them to two parity nodes, whereas other storage schemes are less compute and network intensive. The rationale is that an unreliable memgest writes directly to main memory, but erasure coded memgests have to read memory first to apply XOR operations to the data to build special updates and replicate them to all parity nodes. The size of the parity update is larger than the actual request, since the metadata must be replicated along with the update. Therefore, writing to replicated memgests is faster than to erasure coded ones, and the lowest put latency can be observed for the unreliable memgest $Rep(1,3)$.

We compare Ring with several state-of-the-art KVSs: the single-threaded caching KVS memcached [49], erasure coded Cocytus KVS [174]; a strongly-consistent RDMA KVS Dare with in-memory replication [122]; and a strongly-consistent RDMA KVS RAMCloud with disk-backed replication [111]. We were not able to reproduce experiments for Cocytus, hence we used data from their paper [174], evaluated on the cluster hardware comparable to ours. Both CPUs belong to Intel Xeon E5 family and have the same characteristics except the number of cores. However, both KVSs are single-threaded and it should not influence overall performance. Networks bandwidths are different: they used 10 Gbit/s NICs whereas our cluster is armed with 40 Gbit/s NICs. However, it should not influence the evaluations either, since RS codes are compute-bound rather than network-bound. Cocytus achieves throughput of 2.4 Gbit/s, which is less than 10 Gbit/s.

Memcached does not utilize RDMA to communicate with clients, therefore it provides higher get and put latencies at about 55 μs which is 10x higher than the $REP1$ memgest. Cocytus KVS with $RS(3,2)$ coding scheme for 1 KiB values has get latency 500 μs which is 100x slower than Ring implementation. Also Ring put latencies are 30x lower than Cocytus' ones for 1 KiB objects for the same coding scheme. Dare KVS with replication factor 3 provides the same get latency as Ring, which is not a surprise since they utilize RDMA for communication with clients. Ring also provides approximately the same put latency as Dare for comparable $REP3$ memgest. Another baseline is RAMCloud with 2 backup stores, which provides median 45 μs latency of putting objects up to 512 bytes using an unloaded server with a single client. The high latency is resulting from the fact that our cluster equipped with HDDs instead of SSDs. RAMCloud replicates a put request 2 times, therefore it corresponds to our $REP3$ scheme. RAMCloud has lower



(a) Move latency to SRS32, REP4, REP2, and REP1.

(b) Move latency to SRS31, SRS21, REP3, and REP1.

Figure 2.11: Latency of moves.

main memory storage overhead since it flushes data to disk on backup nodes, whereas Ring stores all data in main memory. Hence, Ring has lower put latency, and also is less subject to tail latency, which is typical for disk-backed systems. Potentially, Ring can also support disk-backed replication and still be strongly consistent.

2.5.2 Move requests

In Figure 2.11, we can see the results of our benchmarks regarding move operations. We split the data into two subfigures to improve readability. The graph shows only destination storage schemes because the source storage scheme does not impact performance due to the local availability of the data. An interesting observation is that moving the object to unreliable scheme $REP(1,3)$ has about the same latency for all object sizes. The reason is that the client does not send the object again and it is copied from high-bandwidth main memory. We can observe a similar pattern for other schemes since the time of moving the object from the unreliable memgest to the reliable is lower than putting directly to the second one. However, put operations are still dominated by building update requests and replicating them.

Benefits of using move requests. Move requests enable explicit resilience management and have a range of use cases discussed in Section 2.2. Next we outline storage benefits of employing move requests using the example of utilizing the unreliable memgest for the *Blob storage*. Let us estimate the footprint of an object in memory before it is committed. By commit operation we mean the decision of storing the object persistently. We denote

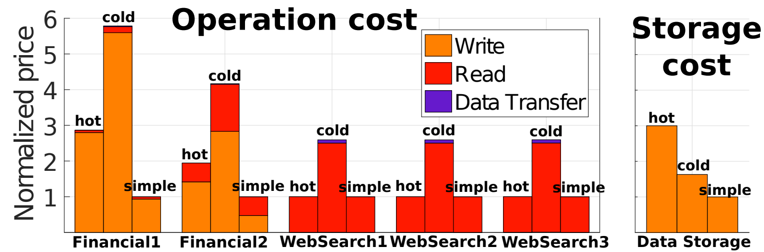


Figure 2.12: Storage Pricing for different I/O traces.

the time between the first write and commit operation as τ. The memory footprint of the object that was written to reliable storage is equal to the size of object S multiplied by storage overhead O and by the duration of time it was stored: $S \cdot O \cdot \tau$. When we write first to the memgest without additional overhead, the footprint is just $S \cdot \tau$. We can thus achieve a relative *memory reduction* of $\frac{1}{O}$ using the unreliable storage scheme. The cost of this significant memory reduction is a single move request, about 5 μs.

The unreliable memgest $REP(1, s)$ has also the lowest put latency among schemes under study. Its put latency is 3x times lower than latency of $SRS(3, 2, 3)$ and 2x lower than latency of $Rep(3, s)$ (see Figure 2.9). Therefore, Ring can provide significant *latency reduction* by employing its move requests and also reduce the load in the backbone network by decreasing the number of replications required to commit an update. Ring can also store a backup version of a key to withstand failures. This is beneficial in the *dynamic importance* use case.

The next advantage is that the unreliable $REP(1, s)$ memgest has the highest throughput among all coding schemes (see Figure 2.13). We can thus achieve a *considerable speedup in throughput* by moving objects to $Rep(1, 3)$ memgest and performing all put requests there. It corresponds to the *Heavy updates* use case.

Real-world applications show different access patterns: some applications are put-heavy, while others are get-heavy. To demonstrate the influence of storage scheme choice on real-world workloads, we estimated the price of operations from five traces obtained from the Storage Performance Council [142] for three storage schemes: $Rep(3)$ (hot), $SRS(3, 2, 3)$ (cold), and $Rep(1)$ (simple). The first two traces represent put-heavy OLTP applications running at a large financial institution. The remaining three are get dominant I/O traces from a popular search engine. Operation and storage costs for hot and cold schemes are obtained from Azure Blob Storage Pricing for Central US in February 2018 [102]. Azure Blob Storage does not provide a simple storage scheme, thus its price is assumed to be

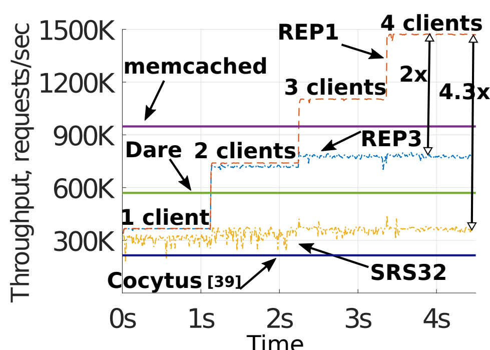


Figure 2.13: Put throughputs of memgests with 1 KiB value size.

the same as for *Rep*(3), but with 3x cheaper puts, as they are not replicated.

Figure 2.12 shows estimated prices for storing data at a constant capacity and performing traces with hot, cold, and simple storage schemes. The estimated costs are normalized and represent the price relative to no replication. As we can see the choice of storage schemes influences the price dramatically, depending on access pattern of traces and the volume of stored data. For example, cold storage is 5.5x more expensive than simple storage and 2x more than hot storage for the **Financial1** trace.

Ring enables *multi-temperature data management* by moving data across storage schemes. It can significantly reduce financial expenses compared to a KVS with a single storage scheme.

2.5.3 Throughput

Figure 2.13 shows throughput traces for $SRS(3,2,3)$, $Rep(1,3)$, and $Rep(3,3)$. In these experiments clients send requests to Ring with the same requests rates of 400K requests/sec. The length of the key is 8B, and value size is 1 KiB. Every second a new client is created, which starts sending requests to Ring. Ring achieves put throughput of almost 1.5M requests/sec under the load of 4 clients for 1 KiB objects with $Rep(1,3)$. $Rep(3,3)$ processes requests 2x times slower, and $SRS(3,2,3)$ 4.3x slower than $Rep(1,3)$. We also compare Ring throughput with throughputs of memcached, Dare, and Cocytus under the load of several clients. According to our experiments, comparable memgests achieve higher throughput than memcached, Dare, and Cocytus.

We also use the YCSB [37] benchmark to generate our workloads for Figure 2.14. The distribution of the key probability is Zipfian [37], with which some keys are hot and

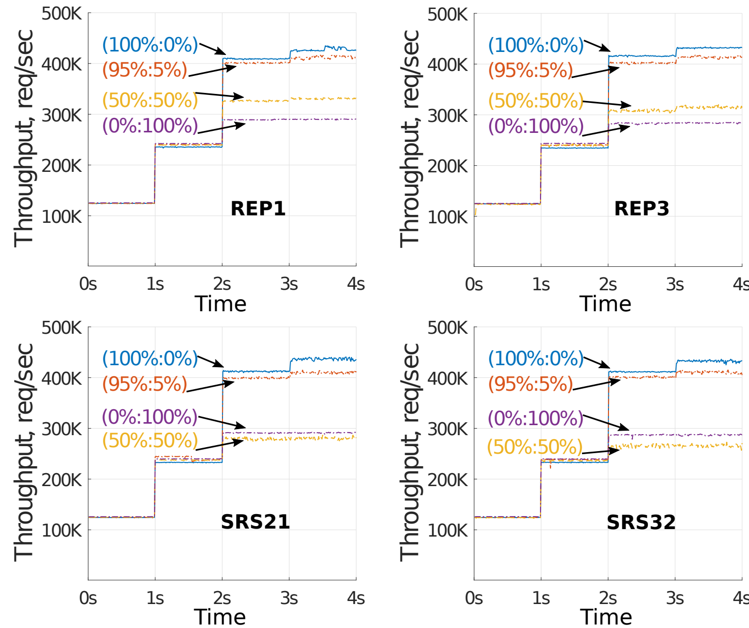


Figure 2.14: Single client throughputs of memgests under different (get:put) ratios workloads with 1 KiB value size.

some keys are cold. The length of the key is 8B, and value size is 1 KiB. We evaluate the systems with different (get:put) ratios, including equal-shares (50%:50%), get-mostly (95%:5%) and get-only (100%:0%).

Figure 2.14 shows traces for $SRS(2, 1, 3)$, $SRS(3, 2, 3)$, $Rep(1, 3)$, and $Rep(3, 3)$. In these traces a single client sends requests to Ring with different request rates. Every second the client doubles its request rate from 128K requests/sec until it reaches 1024K requests/sec.

In all experiments the requests were served by 3 coordinator nodes, and the client accesses them in order. As noted earlier, all memgests share the same implementation of how get requests are served, and therefore exhibit the same get throughput of 418K requests/sec. This number drops as the workload's put ratio is increased. Since our implementation is single threaded, we have not noticed a significant difference between different storage schemes. A small drop in throughput of erasure coded schemes for (50%:50%) workloads is due to differences in memory allocation algorithms for replicated and erasure coded memgests.

Figure 2.14 shows that the highest put throughput is achieved by the unreliable memgest $Rep(1, 3)$ at 290K requests/sec. Other schemes achieve a slightly lower throughput of 280K requests/sec, despite the fact that they have to replicate requests. This is due to

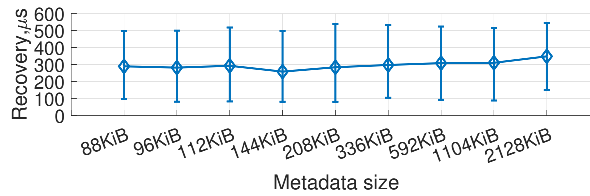


Figure 2.15: Recovery latencies depending on metadata size.

two effects: the replicated memgests employing quorum-based replication; and the current implementation being single-threaded. We believe that a multi-threaded implementation of memgests can achieve higher performance. Finally, Ring's throughput of $SRS(3,2,3)$ memgest is 1.5x-2x faster than a comparable Cocytus configuration, which achieves approximately 220K requests/sec for (100%:0%), (95%:5%), and (50%:50%) workloads [174].

2.5.4 Failures and recovery

We evaluate the recovery efficiency of Ring depending on recovered metadata size (see Figure 2.15). Each measurement is repeated 500 times, and the figure reports the median and the 90th percentile. The coordinator node failures are simulated by manually killing processes on the node. Our evaluations show that for all storage schemes the median recovery time is 300 μs for recovering the coordinator node after a failure with 1 MiB of metadata. Ring has to ensure strong consistency and therefore recover all metadata of all storage schemes within the system before answering client requests. Without this measure, there would be a risk for the system to reply with stale data, since the key with the highest version can be stored in an unrecovered memgest. The complexity of the recovery process results in a high variance of metadata recovery time, which includes:

1. The leader detects the failure and substitutes the failed node with a spare one.
2. The leader replicates the new configuration to all nodes.
3. Once all nodes have received the decision, they start in their new role, i.e., existing nodes connect to the new node.
4. The new node creates the required empty memgests and connects to alive nodes.
5. Once nodes are connected, the new node requests the metadata, followed by the logs, which store previous requests from clients to ensure strong consistency.
6. Once the coordinator nodes have all the metadata, they can rebuild the volatile hashtable. With only the metadata, the new node can serve put and delete requests

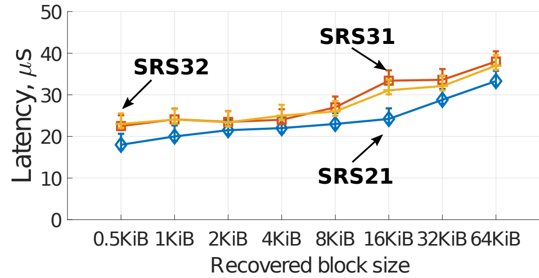


Figure 2.16: Recovery latencies for storage schemes $SRS(2, 1, 3)$, $SRS(3, 1, 3)$, $SRS(3, 2, 3)$, depending on recovered block size.

without actual data. To process get requests, however, the node needs data, which is either copied or recovered depending on the storage scheme (Rep or SRS, respectively).

Block recovery. We also evaluate the recovery time of erasure coded data blocks for different SRS memgests. Figure 2.16 reports the median and the 90th percentile. Time is measured from receiving a request from the client to when the block is fully recovered. As expected, the time of recovery is correlated to the size of the lost block. We can also see that the latency of recovering a block encoded with $SRS(3, 1, 3)$ takes longer than $SRS(2, 1, 3)$, despite the fact that they have the same number of coordinator nodes. As mentioned before, the data in $SRS(2, 1, 3)$ is encoded according to $RS(2, 1)$, and according to $RS(3, 1)$ in $SRS(3, 1, 3)$. It therefore requires collecting 2 blocks for $SRS(2, 1, 3)$ and 3 blocks for $SRS(3, 1, 3)$ to recover one lost block. At first glance, it seems that $SRS(2, 1, 3)$ and $SRS(3, 1, 3)$ are two identical schemes because they are allocated across 4 nodes and ensure the same throughput and latency. As it can be seen from the experiment, however, they have have different recovery rates and, therefore, different resilience. Since $SRS(2, 1, 3)$ recovers faster than $SRS(3, 1, 3)$, it provides higher reliability and availability guarantees.

Figure 2.16 shows that $SRS(3, 2, 3)$ and $SRS(3, 1, 3)$ have approximately the same latencies for all block sizes. This is because the number of parity nodes affects the number of failures the scheme can tolerate, while computation stays practically the same. $SRS(3, 2, 3)$ recovers data a little bit faster because the recovery master requires any 3 blocks out of 4 available ones, whereas only 3 blocks are available for reconstruction for $SRS(3, 1, 3)$. Therefore, $SRS(3, 2, 3)$ can recover faster in the case of a single failure.

2.6 Related work

To address opportunities of exploiting explicit resilience management, we give an overview of the existing technologies allowing users to modify storage schemes. In particular, one of the most well-known KVS, Redis [135], offers the option of determining the degree of replication of each volume, but it does not allow performing the update in a per-key manner. It also does not support erasure codes to reduce memory usage. Redis also does not ensure strong consistency since it employs asynchronous replication.

Erasure codes also support altering resilience of the data. For instance, it is well known that RAID6 can be easily converted to RAID5 and vice versa, but it can take days to decode all the data on the disks [32].

The known method to combine a wide range of storage schemes and be able to change them per object is to maintain a special name node, which stores all metadata and references to requested data [168, 139]. The name node can, however, become a bottleneck and be a single point of failure. In addition, this approach leads to additional hops in data center networks to read and write data.

Another approach of changing storage schemes with low computational overhead is introduced by the BlowFish distributed data store [79]. BlowFish stores data in a compressed format and enables dynamically changing the compression factor. A smaller compression factor indicates higher storage requirements, but also lower latency (and vice versa). However, to ensure resilience the data itself is still replicated, and changes in compression do not influence the reliability of the stored keys. Finally, the granularity for updating the compression factor is a single shard, whereas Ring supports per-key resilience management.

It is worth mentioning that Ring is not the first attempt to create a KVS that supports multiple resilience levels. For instance, Phanishayee et al. [117] suggest supporting multiple replication algorithms with different consistency levels [26, 23, 160]. However, this approach does not support erasure coding and is not strongly-consistent.

2.7 Summary and Discussion

Modern, in-memory KVSs are widely used because of their performance characteristics and their ability to provide fault tolerance and strong consistency. However, KVSs do not offer users the possibility to select the most suitable trade-off in terms of fault tolerance,

performance, and resource usage. In this chapter we have presented Ring, a distributed in-memory KVS that allows users to control the level of resilience on a per-key basis and, thus, control the resource usage of the KVS that better matches the application profile. The core of Ring is a novel encoding mechanism, Stretched Reed-Solomon (SRS), which enables strongly consistent systems to support different resilience levels on a per-key basis, allows dynamic changes, and does so transparently, without affecting performance or consistency. The experimental evaluation indicates Ring provides significant memory savings and allows choosing the best trade-offs between reliability, performance, and cost.

3

Compactable Remote Memory over RDMA

Distributed memory systems are becoming increasingly important since they provide a system-scale abstraction where physically separated memories can be addressed as a single logical one. This abstraction enables memory disaggregation, allowing systems as in-memory databases, caching services, and ephemeral storage to be naturally deployed at large scales. While this abstraction effectively increases the memory capacity of these systems, it faces additional overheads for remote memory accesses. To narrow the difference between local and remote accesses, low latency RDMA networks are a key element for efficient memory disaggregation. However, RDMA acceleration poses new obstacles to efficient memory management and particularly to memory compaction: network controllers and CPUs can concurrently access memory, potentially leading to inconsistencies if memory management operations are not synchronized. On the other hand, memory compaction is needed to lower memory requirements and avoid degraded performance. Nevertheless, RDMA-accelerated distributed memory systems do not provide memory compaction and are exposed to memory fragmentation.

The work in this chapter explores the following research questions:

- What are the main causes of memory fragmentation in remote memory systems?
- How can we provide memory compaction to RDMA-accelerated remote memory systems without compromising strict consistency?
- How can we ensure that a compacted object is still accessible via one-sided RDMA accesses?

To address these questions we introduce CoRM, an RDMA-accelerated remote memory system that supports memory compaction and ensures strict consistency while providing one-sided RDMA accesses. CoRM exploits RDMA-aware memory remapping to silently move objects across physical pages, preserving their base virtual addresses and RDMA access keys. We show how CoRM can enable compaction to reduce memory fragmentation and significantly decrease memory usage of RDMA-accelerated management systems, introducing minimal memory and communication overheads.

The content of this chapter has been published at the International Conference on Management of Data (SIGMOD) in 2021 [150]. The work in this chapter was done in collaboration with Salvatore Di Girolamo and Torsten Hoefler.

3.1 Motivation

RDMA-capable Network Interface Cards (RNICs) empower systems to access the main memory of remote peers without involving the host CPUs, providing up to 22x shorter latency [122], and up to 20x higher throughput [105], compared to traditional TCP/IP networking. However, the use of RDMA can prevent memory optimization strategies, such as memory compaction. In fact, remote objects are accessed by specifying their virtual addresses at the remote host: if the remote host relocates an object, its virtual address might change, requiring to propagate this update to the other nodes. To avoid this issue, some RDMA systems do not expose the virtual addresses of the stored objects, distributing instead objects' handles to the clients [111, 134]. While this indirection hides the process of updating pointers of relocated objects, it can significantly hinder performance because of the pointer chasing overheads [47, 46].

Memory fragmentation is a serious concern across the spectrum of modern computing platforms and databases [84, 124, 98]. Traditional memory allocators without compaction

can suffer from catastrophic memory fragmentation [130, 134]. Furthermore, while fragmentation increases memory usage of in-memory data stores by up to 69% (e.g., Redis, MongoDB, and VoltDB) [84, 98, 176, 115, 114], it also has a negative impact on their performance due to memory sparsity [84]. For distributed systems, the fragmentation problem is particularly severe as the memory space may consist of hundreds of physical nodes and the stored data can be replicated multiple times for fault tolerance. Each additional machine can potentially increase the amount of wasted fragmented memory (Section 3.2.1).

We propose CoRM, a remote memory system that exploits RDMA for fast remote accesses and supports memory compaction. Additionally, CoRM's compaction is RDMA-safe: objects are still accessible via RDMA (Section 3.3.4) and the user is guaranteed to observe their consistent state (Section 3.3.1) even if they have been relocated by the compaction algorithm. To facilitate compaction, objects in CoRM are associated with block-local object IDs that are randomly generated at object allocation time. Our compaction algorithm is probabilistic: two memory blocks can be compacted into one only if they are conflict-free, that is, the objects in the two blocks do not have the same IDs. This enables a trade-off between compaction probability and space overhead: the larger the object ID space, the higher the compaction probability (Section 3.3.3). In some cases, relocated objects can experience higher access times because of indirections. Clients can detect this situation and fix the pointers to recover efficient one-sided RDMA access (Section 3.3.1).

3.2 Background on Shared Memory Systems

Distributed Shared Memory (DSM) systems [30, 47, 28, 175] provide an abstraction where the memory of multiple different physical nodes is viewed as a single unified memory space. Applications running in a DSM can randomly access their local memory or the memory of remote nodes. To implement this abstraction, DSMs provide APIs for managing (i.e., allocating and freeing) and accessing (i.e., reading and writing) memory. Memory accesses are translated to load/stores if they target local memory, otherwise, they lead to requests that are sent over the network to the target nodes.

3.2.1 Concurrent memory allocators

In a DSM system, the memory of a process can be allocated by the application tasks running locally or by remote ones. Hence, a DSM node needs to manage concurrent memory allocations, as it would be in normal multi-threaded applications with the addition that now allocation requests can also come from the network.

Concurrent Memory Allocators (CMAs) have two main requirements: (1) scale with the number of threads managing memory; (2) maintain low memory fragmentation, maximizing memory efficiency. We define memory fragmentation as the ratio between the amount of memory granted by the operating system to a process and the amount of memory that the process is effectively using.

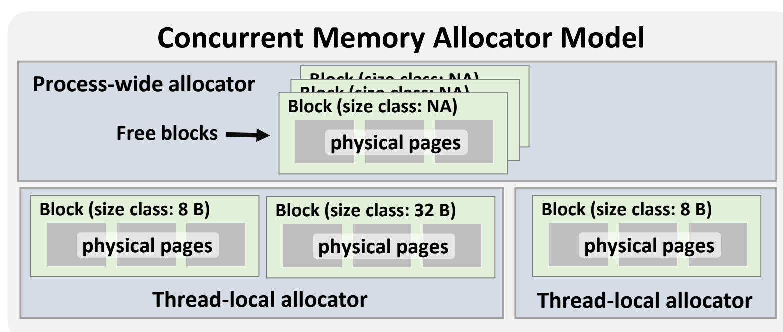


Figure 3.1: Concurrent Memory Allocator: each thread allocates memory from its thread-local allocator, which requests memory blocks from the process-wide allocator.

Scalability. To improve scalability, most CMAs [18, 47, 28, 124, 49, 89] adopt a two-level architecture, as depicted by Figure 3.1. In this model, each thread is served by a thread-local allocator that has its free memory heap: the memory allocation requests are served from this heap without the need for global synchronization. If a thread-local allocator runs out of memory, it requests new memory from the process-wide allocator. The process-wide allocator may allocate memory directly from the operating system.

To avoid frequent accesses to the process-wide allocator, which can potentially require synchronization, the thread-local allocators fetch more than one free page at a time. The set of free pages that are fetched from the process-wide allocator in a single access is defined as *block*. Blocks are used to store objects belonging to predefined size classes: a given memory block can be used only for storing objects of a certain size. An object is allocated in the smallest size class that can fit it. Therefore, the size classes must be carefully chosen to limit internal fragmentation.

The block-based approach introduces a trade-off between synchronization overhead and memory efficiency: the larger the block size, the less the number of accesses to the process-wide allocator, hence the better the scalability. On the other hand, larger blocks can lead to memory inefficiency if the allocated blocks are not fully utilized by the allocating thread.

Memory Fragmentation. High memory fragmentation can be caused by irregular allocation spikes or low usage of particular size classes [12]. Allocation spikes happen when an allocator experiences a high volume of allocations followed by only a partial set of deallocations. The issue is that it is not guaranteed that these deallocations will lead to memory being freed up: In fact, blocks containing at least one object cannot be released back to the process-wide allocator. In this scenario, the threads can be left with many blocks that are scarcely utilized and cannot be released, causing memory fragmentation.

Low occupancy of some size classes can also be a source of memory fragmentation. Consider an example where an application allocates an object of size s on each of its T threads: the thread-local allocators will allocate the object from their local memory, increasing the memory requirement to $(T \cdot s)$. However, if the thread-local allocators do not have blocks of the requested size-class, they will request a new block from the process-wide allocator, potentially allocating up to $(T \cdot B)$, where B is the block size. If objects of size s are uncommon, most of the newly allocated blocks will have very low occupancy: e.g., if there is only one object per thread of that size, the overall unused memory is $T \cdot (B - s)$ bytes.

Memory Compaction. To reduce memory fragmentation, CMA systems can adopt memory compaction strategies. In principle, these strategies consist of taking a set of scarcely utilized blocks and merging them into one, releasing the others. This process needs to preserve the accessibility of the compacted objects so clients could still access them. A common strategy is to employ an indirection table that maps object keys to their current memory location [111] allowing systems to move objects freely in memory by updating corresponding entries in the table. However, the use of indirection tables results in revoking direct RDMA access to stored objects (Section 3.2.2) leading to a 2x reduction in read throughput [47, 46].

An alternative approach has been recently proposed by Mesh [124]: it does not use indirection tables and, instead, exploits virtual memory functionality to compact memory without the need for changing virtual addresses of relocated objects. Mesh merges the content of scarcely utilized blocks into one block and then updates the virtual-to-physical mapping of the affected blocks making their virtual addresses to point to the single resulting block holding the relocated objects. Mesh requires the relocated objects to reside

at the same offset as they did in their original blocks (i.e., no conflicts) to preserve their virtual addresses. In Section 3.3.1, we discuss how this constraint limits the probability of compacting two blocks and show how our new compaction strategy can avoid this issue.

3.2.2 RDMA-accelerated DSM systems

To efficiently enable the shared memory abstraction, DSM systems must minimize the overheads of remote memory accesses. To narrow the performance gap between local and remote accesses, modern DSM systems employ RDMA to ensure low latency and high throughput [47, 28, 3, 2]. RDMA is a mechanism that allows one machine to directly access the memory of other remote machines across the network. The RNIC on the sender side reads data directly from the sender's memory and injects it into the network. On the receiver side, the RNIC receives the data and writes it directly to the host memory, bypassing the operating system and minimizing the end-to-end latency. RDMA-enabled hosts communicate through either reliable or unreliable Queue Pairs (QPs). In this chapter, we consider only reliable QPs, as it is the only type of QP that supports one-sided RDMA read operations.

In principle, all DSM operations (i.e., memory allocation and freeing, read and writes) can be implemented as RDMA one-sided operations. However, most of them would require multiple round-trips, hindering the performance gains given by RDMA: e.g., allocating memory with only RDMA one-sided operations would require to read, modify, and write back the allocation state of the target node (without considering that multiple nodes of the DSM can be targeting the same memory at the same time). We take the approach of FaRM [47], which accelerates remote reads with RDMA, while implementing other operations with Remote Procedure Calls (RPCs). We now describe how RDMA can be used to accelerate read operations and how it can help to have low-latency RPCs.

RDMA reads. To expose memory over the network and allow other nodes to read it, a DSM node must register the memory on its RNIC. Memory registration consists of pinning the associated memory pages in physical memory and copying the related page table entries to the Memory Translation Table (MTT) of the RNIC. The RNIC generates keys for local and remote accesses, namely *l_key* and *r_key*. The memory region can be accessed by any local QP which has the *l_key* and by remote endpoints having the *r_key*. Figure 3.2 shows an RDMA read example: when the RNIC receives an RDMA request, it translates the target virtual address using its MTT into the corresponding physical page

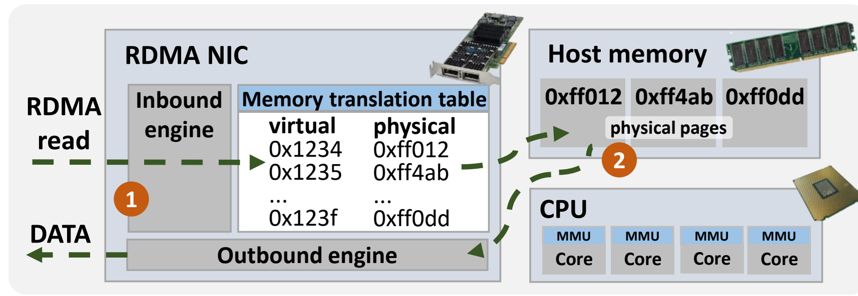


Figure 3.2: Accessing remote memory via RDMA read.

①, then it uses the computed physical address to issue a DMA read towards the host memory ②, sending back the read data.

RDMA reads and memory compaction. RDMA accesses are issued by specifying the virtual addresses of the data in the memory of the target machine. This characteristic often limits the memory compaction capabilities of DSM systems exploiting RDMA. Memory compaction requires to move allocated data in memory, changing the virtual-to-physical mapping and potentially changing the virtual addresses themselves. When this happens, remote peers cannot access that data via RDMA anymore because the virtual addresses they hold may become invalid. For this reason, DSM systems like FaRM [47], do not support compaction and therefore can suffer from memory fragmentation, which can be especially dangerous in scenarios like the ones discussed in Section 3.2.1.

RPC operations. Memory management operations and accesses can be handled via RPC. On a DSM node there is a number of worker threads that are in charge of running application-defined tasks and handling RPC calls. RPC requests are pushed into an RPC queue that is shared between the worker threads. The handling of RPC requests can be accelerated with RDMA by letting remote peers push the RPC requests directly to the RPC queue [72].

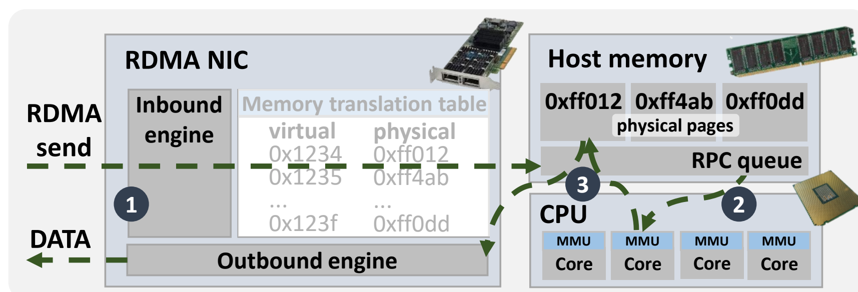


Figure 3.3: Accessing remote memory via RPC read.

Figure 3.3 shows an example of an RPC read operation: the memory read request arrives at the RNIC and is copied directly into the RPC queue ❶. The DSM worker threads regularly poll the RPC queue to check for new requests. When a thread gets a memory request ❷, it serves the request and sends the result back to the initiator ❸. In this case, the virtual-to-physical address translation is done by the MMU of the core where the worker thread is executing.

3.3 CoRM

CoRM is a memory management system that exploits RDMA to improve both latency and throughput of memory accesses. With CoRM, we show that it is possible to enable memory compaction in RDMA-accelerated DSM systems without introducing indirection to take full advantage of one-sided RDMA operations.

Relation to other systems. DSM systems like FaRM sacrifice memory compaction in order to employ RDMA to accelerate remote communications. CoRM is designed to provide memory compaction to RDMA-accelerated DSM systems such as FaRM without compromising strict consistency, but requiring storing extra metadata in object headers (Section 3.4.4). Since CoRM's API mimics FaRM's API and only adds a maintenance call for releasing unused virtual addresses (Section 3.3.2), we believe our compaction strategy can be integrated to FaRM without extra effort.

System	Type	RDMA	Mem. Compaction	Vaddr Reuse
Mesh [124]	Allocator	✗	✓	✗
FaRM [47]	DSM	✓	✗	-
CoRM	DSM	✓	✓	✓

Table 3.1: Comparison of FaRM, CoRM, and Mesh.

The compaction strategy of CoRM is similar to Mesh [124], which is a memory allocator supporting memory compaction for C/C++ applications. Unlike Mesh, CoRM's memory compaction strategy can additionally merge blocks having objects placed at the same offsets, improving the compaction probability (Section 3.3.3). Furthermore, Mesh does not solve the problem of virtual space exhaustion, which is addressed in CoRM's design by tracking the block in which each object was initially allocated (Section 3.3.2). For that,

CoRM requires users to perform additional actions, called pointer correction (Section 3.3.1) and pointer release (Section 3.3.2), that have little effect on performance (Section 3.4.3).

Table 3.1 recaps the characteristics of Mesh, FaRM, and CoRM. *RDMA* indicates if the system supports RDMA-accelerated remote accesses; *mem. compaction* indicates if the system supports memory compaction; *vaddr reuse* tells if the system can reuse virtual addresses after compaction, avoiding virtual address space exhaustion (Section 3.3.2). CoRM improves over Mesh by introducing a new compaction strategy that increases the probability that two memory blocks can be compacted and it extends FaRM by introducing support to memory compaction, making it resilient to memory fragmentation while still preserving strong consistency and one-sided RDMA accesses.

Interface. The API is shown in Table 3.2. Users can allocate and free objects using *Alloc* and *Free*. Allocations return 128-bit pointers that can be used to access objects. Those pointers include the actual 64-bit object address and RDMA-related metadata such as the *r_key*. Read operation can be used to read an object given its pointer. CoRM supports two types of reads: via RPC (*read*) and via one-sided RDMA (*DirectRead*). One-sided RDMA reads are lock-free and are performed without involving the remote CPU. The application is guaranteed to observe a consistent object state even in case of concurrent writes to the same object. To support lock-free consistent RDMA reads, we embed versioning information into the object itself [47]: a version number is stored with each cacheline, allowing the reader to check consistency by verifying that all cachelines that have been remotely read have the same version number. This strategy relies on cache-coherent DMA and requires cacheline-aligned allocation. If the consistency check fails, the RDMA read needs to be issued again. To update an object, the user can use the *Write* call to write a local buffer to a remote one.

3.3.1 Memory allocation and compaction

CoRM supports compaction that does not compromise the object pointers of the clients. Our system exploits RDMA-aware memory remapping to silently move objects across physical memory blocks while preserving their virtual addresses and RDMA access keys.

Allocation algorithm. CoRM uses a concurrent memory allocator as described in Section 3.2.1, similar to most memory systems [47, 28, 124, 49, 89]. The allocator supports a list of distinct 8-byte aligned sizes, that are chosen to reduce the average internal fragmentation due to round up to the nearest size class. The process-wide block allocator

Table 3.2: CoRM APIs

API	Type	Pointer Correction	Description
`ctx* CreateCtx(char* ip,int port)`	Initialization	N/A	connect to a remote memory allocator
`addr_t ctx::Alloc(u32 size)`	RPC	N/A	allocate object with a given *size*
`int ctx::Free(addr_t &addr)`	RPC	Yes	free object at a given *address*
`int ctx::Read(addr_t &addr,u8* buf,u32 size)`	RPC	Yes	read object to *buffer* with a given *size* using RPC
`int ctx::DirectRead(addr_t &addr,u8* buf,u32 size)`	RDMA	No	read object using one-sided RDMA read
`int ctx::ScanRead(addr_t &addr,u8* buf,u32 size)`	RDMA	Yes	read object by reading and scanning the whole block which contains the object
`int ctx::Write(addr_t &addr,u8* buf,u32 size)`	RPC	Yes	write content of *buffer* to the remote object using RPC
`int ctx::ReleasePtr(addr_t &addr)`	RPC	Yes	an explicit call to release old object *address* using RPC

in CoRM can allocate blocks with sizes that are multiples of 4 KiB (i.e., a normal-sized page). However, CoRM can easily be extended to work with huge pages to reduce the number of pages.

Block allocation is performed in two steps: first, we allocate a physical page using the *memfd_create* system call [99]; then the allocated physical page is mapped to virtual space using *mmap*. The process-wide allocator keeps track of all virtual-to-physical mappings. The *memfd_create* call creates an anonymous file that lives in RAM, which can be modified, truncated, and memory-mapped as a regular file. To reduce the number of allocated file descriptors, CoRM allocates files of 16 MiB and uniquely identifies physical blocks as a tuple of the file descriptor and the page offset in the file. The block allocator is also responsible for registering allocated blocks with the RNIC to enable remote access (Section 3.3.4).

Compaction algorithm. CoRM can compact blocks of the same size class belonging to the same machine. The high-level idea of the compaction algorithm is to find two blocks of the same class with low utilization and copy objects from a block (i.e., source) to the other (i.e., destination), as illustrated in Figure 3.4. Once all objects have been copied, the source block can be deallocated and its virtual address is remapped to the physical address of the destination block. At this point, we have two virtual addresses pointing to the same physical page. The mapping is updated also on the RNIC in order to preserve RDMA access to the objects of the source block (Section 3.3.4).

A similar approach to memory compaction has been proposed by Mesh [124]. However,

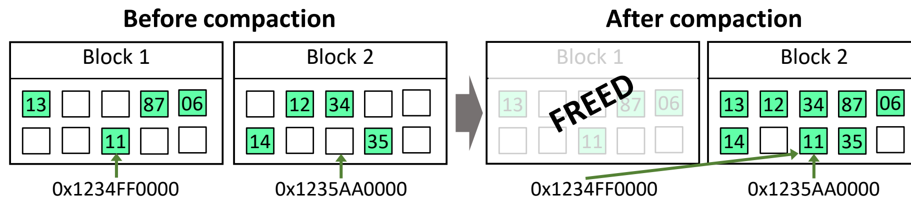


Figure 3.4: Compaction of two blocks without conflicts on object offsets or IDs. After compaction, the virtual addresses of block 1 point to block 2.

in Mesh compaction is only possible when no objects in the blocks occupy the same offsets. The limitation comes from the fact that Mesh is a plug-in replacement for malloc in C/C++ programs where the applications can freely read and write memory by using virtual addresses with load/store instructions. In that case, the page virtual address can be remapped transparently (i.e., the translation is performed by the MMU) but the object offsets cannot change. Thus, Mesh can compact blocks only if their objects do not conflict in offsets. In DSM systems, however, users always use explicit read/write functions to access memory: these are needed to resolve remote pointers and to enable concurrency control.

CoRM takes advantage of the DSM programming model and relaxes the requirement of the compacted objects keeping the same offsets after compaction. To achieve that, CoRM assigns identifiers (IDs) to each object in the block. The ID is unique only within a single block and is generated randomly using a uniform distribution. An object is uniquely identified in memory by the block address and the object ID, which is stored in the header of the object. This design choice allows CoRM to compact two blocks only if the objects in them do not have the same IDs. Differently from Mesh, where the compaction condition is based on offsets, in CoRM the object IDs are random and the IDs size can be tuned (16 bits by default), improving the compaction probability. In fact, while blocks can also have conflicts in the object IDs, they are less probable than offset conflicts (Section 3.3.3). Figure 3.5 shows an example where the blocks to compact have conflicting offsets. In this case, a Mesh-like approach would not be able to compact, while CoRM can move the conflicting objects to a different offset, preserving their IDs.

During compaction, it is preferable to preserve the offset of the objects as it preserves the virtual addresses of compacted objects. When it is not possible (i.e., because of offset conflicts), CoRM is free to move objects to new offsets within the block. The moved objects can still be found by looking for their object IDs which is included in the 128-bit

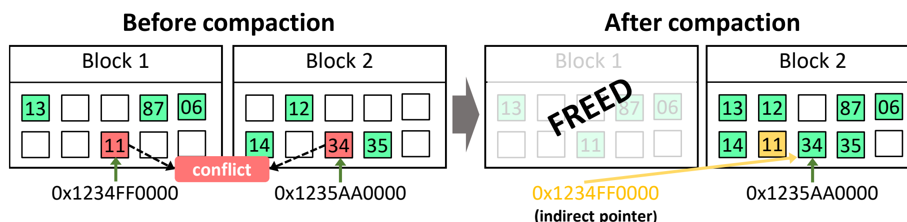


Figure 3.5: Compaction with offset conflicts. CoRM can compact blocks by moving objects. Accesses to moved objects with indirect pointers need additional pointer correction.

pointers returned by the *Alloc* function. A pointer that points to an object that has been moved to a different offset is defined as **indirect**. Instead, pointers to objects that have not been moved even after compaction are defined as **direct**. When accessing an object with an indirect pointer, the virtual address translation will point to the correct block but the object will not be found at the given offset. In this case, CoRM will need to perform an additional action, called *pointer correction* (Section 3.3.1), in order to retrieve the requested object. The pointer correction is implicitly performed during all API calls but one-sided *DirectRead* (see Table 3.2).

Compaction policy. CoRM calculates a fragmentation ratio for each size class. CoRM triggers compaction for a size class if its fragmentation ratio exceeds a fragmentation threshold. The fragmentation threshold can be tuned for each size class depending on its compaction probability (Section 3.3.3). CoRM can additionally start compaction when an allocation fails due to shortage of memory.

Compaction mechanism. In CoRM, each thread has its private memory allocator. Therefore, the blocks that can be compacted may belong to different threads, preventing efficient lockless memory compaction. To address this issue, CoRM selects one of the worker threads as a compaction leader, that performs compaction in two stages: *block collection* and *block compaction*. During the block collection, the leader broadcasts a collection request to all other threads, asking for sufficiently low-occupancy blocks of a certain size-class. In the second stage, after all threads reply to the collection request, the leader can start the compaction algorithm. Our two-stage design removes the need to have costly concurrent data structures since CoRM holds an invariant that any block is owned by at most one thread. CoRM tries first to compact the least utilized blocks, as they have fewer elements and induce fewer offset collisions. During compaction of two blocks, besides copying objects, CoRM also merges metadata of the affected blocks. The metadata is a hash table that keeps a mapping between object IDs and offsets used only

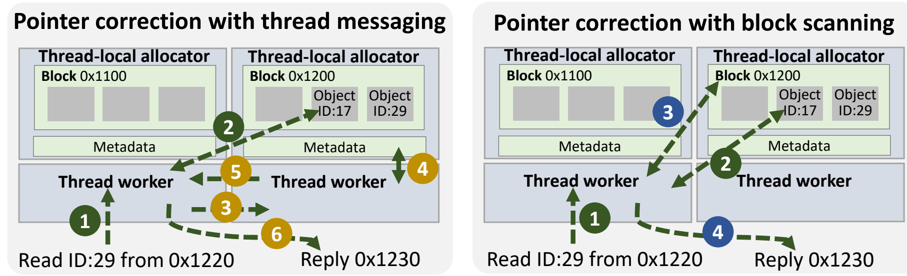


Figure 3.6: CoRM supports two approaches for pointer correction: thread messaging, and memory scanning. The worker received a request to read object ID:29 from address 0x1220, which belongs to block 0x1200, but the object has been moved to address 0x1230.

for fast pointer correction (Section 3.3.1).

Pointer correction As the compacted objects can move within memory after compaction, the virtual pointer held by the user may not directly point to the desired object. To avoid searching an object in its block, each user's object pointer contains the *offset hint* where the object is expected to be in the block. CoRM always optimistically accesses the object at the hinted offset (using a load instruction) and then checks its ID. If the ID of the accessed object does not match the one in the used pointer, then CoRM performs a search to find the requested object. Once the object is found, the hint inside the object pointer is updated to the new offset, making the pointer *direct*.

RPC calls. Pointer correction is transparent to the user when RPC-based calls are used to read and write objects. We show two approaches that can be used to find objects accessed with indirect pointers: the first approach uses inter-thread communication, while the second directly scans the block with the requested object in order to find it. Figure 3.6 illustrates both approaches. Whenever an RPC read call is served ❶, CoRM checks if the object ID of the hinted object matches the ID of the requested one ❷. If this check fails and the solution with inter-thread communications is employed (left), then the thread serving the RPC request forwards it to the thread owning the requested block ❸. In this way, the owner thread can quickly query metadata of the block to determine the position of the object ❹. For each block we keep a thread-local mapping between object IDs and offsets stored in it. Once the object is found, the owner thread sends the corrected pointer back to the thread handling the RPC request ❺ allowing it to complete the request and reply to the client ❻. While this approach is more efficient for large block sizes since it avoids expensive scans, it can delay the request processing if the owner thread is busy with other activities (e.g., compaction). Instead, the *block-scanning* approach does not require

inter-thread communications by letting the thread serving the RPC call scan the block with the requested object, even if it belongs to another thread. The thread compares the IDs of all allocated objects with the ID of the requested one to find the new offset ③. After the object is found, it is sent back to the client ④.

RDMA calls. RDMA read accesses are not served by the CoRM worker threads but are directly performed by the RNIC. This implies that the pointer correction mechanisms we described above cannot be applied for *DirectRead* calls and that we need to move the pointer-correction activity to the client side. Similarly to the RPC case, a client performing an RDMA read can detect if the read object is correct by comparing its ID to the one stored in the accessed pointer. If the two do not match, then the client has two options: (1) issue an RPC-read, triggering the pointer correction mechanisms of above; (2) issue an RDMA read of the entire block where the object is stored. With (2), the CoRM client-side library scans the block in order to find the requested object. We define (2) as *ScanRead*.

Consistency. Clients interfaced with CoRM are guaranteed to observe consistent objects even in the case reads are interleaved with writes or happen while memory compaction is in progress. While RPC calls can directly ensure consistency by employing explicit locking of object headers, this is not true for RDMA reads.

When performing a *DirectRead*, CoRM issues a one-sided RDMA read to retrieve the object and then checks if the read object is valid. Other than the case described in Section 3.3.1, there are two reasons for which a read object might be invalid: (1) the read object is being updated by a concurrent write (i.e., the object is corrupt); (2) the read object is under compaction. To detect the first case, we store the object version into the header of the object and in the first byte of each cacheline, as proposed by FaRM [47]. Writes to the object increase the object version. Clients can verify the validity of the read objects by checking that the version numbers match. To detect the second case, we store a lock state into the object header (2 bits). At the beginning of a compaction process, CoRM locks all objects that are going to be compacted. If a client reads a locked object, then the object is invalid. In case the read object is detected as invalid, then the read is repeated after a backoff period.

Fault Tolerance. The current implementation of CoRM is not fault tolerant. Thus, we assume that if any thread fails then the whole process fails. Fault tolerance is an interesting area of future work. CoRM could employ a fault-tolerant replication protocol (e.g., [48, 75, 66, 147]) to withstand failures.

3.3.2 Pointer release

The main effect of the CoRM compaction scheme is to reduce the physical memory utilization by reducing fragmentation. However, this does not automatically translate to lower utilization of the virtual address space since all virtual addresses are preserved after compaction. As a result, if virtual addresses are not released in the long run, solutions like CoRM or Mesh can run out of virtual space.

To address this problem, CoRM stores the address of the virtual block where the object has been initially allocated in the header of each object. This allows us to keep track of how many objects that have been moved out from an old virtual address are still valid and can still be accessed. Once there are no more of such objects, i.e., they have been deallocated with *Free* calls, then CoRM can safely assume that the virtual address can be reused.

Additionally, CoRM provides the *ReleasePtr* call allowing clients to explicitly release an old object pointer without actually freeing the corresponding object. This call can be used by the clients to communicate that all copies of the old pointer have been corrected and it is safe to reuse that address. The clients must ensure that the pointer is not reused to address the same object once it has released it. CoRM always notifies the user if it uses an old pointer. Note an old pointer can be corrected to become direct (i.e., having correct offset), but it will still reference the old block address. We expect *ReleasePtr* calls to be rarely used (only when virtual space is almost exhausted) as *Free* calls can implicitly free unused virtual space.

3.3.3 Probability of compaction

A key threat to CoRM's memory compaction capabilities are collisions in object IDs. This issue is similar to the one that Mesh has for allocation offsets, where a conflict in the offsets prevents two blocks from being compacted. We now define the probability of compaction of two blocks B_1 and B_2 depending on their occupancy.

We denote $p(B_1, B_2)$ as the probability of compacting the block B_2 into the block B_1. Note that the probability $p(B_1, B_2)$ is equal to $p(B_2, B_1)$. As only blocks of the same type are compactable, we denote s as the total number of objects that can be stored in a block. We denote n as the total number of different object identifiers a block can have. For Mesh, n is the number of objects a block can store, which is equal to s. For CoRM, n is the

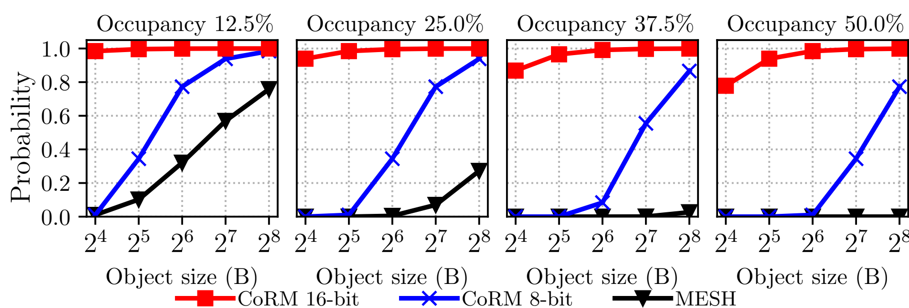


Figure 3.7: Compaction probability of two random blocks depending on occupancy and size class. CoRM probabilities are reported for 8-bit and 16-bit object IDs.

total number of possible object IDs, which is 2^x, where x is the number of bits used to store the object IDs. The value of x is a parameter of CoRM and can be used to tune the compaction probability. The larger object IDs, the lower the probability of ID conflicts but increases memory usage, as they are stored in the header of each object (Section 3.3.2). This is a key difference from Mesh, where n depends solely on the block and the object class size. E.g., for 16 byte objects, a 4 KiB block can store 256 objects, whereas for 128 byte objects the same block can fit only 32 objects. In this case, if CoRM would use 8-bit IDs, then it would have the same compaction probability of Mesh. However, already for larger size classes, the compaction probability of CoRM with 8-bit IDs will become higher than Mesh, because the number of offsets that Mesh can use would decrease.

We define b_1 and b_2 as the number of objects stored by B_1 and B_2, respectively. Assuming that object IDs are randomly generated with a uniform distribution, the probability of no collisions is:

$$p(B_1, B_2) = \begin{cases} \frac{\binom{n-b_1}{b_2}}{\binom{n}{b_2}}, & \text{if } b_1 + b_2 \leq s \\ 0, & \text{otherwise,} \end{cases}$$

where $\binom{n}{k}$ is the binomial coefficient, $\binom{n-b_1}{b_2}$ is the total number of blocks not using any ID of the objects stored in B_1, and $\binom{n}{b_2}$ is the total number of blocks that have b_2 allocated objects. When the sum of objects in two blocks is greater than the total number of slots, then the blocks are not compactable.

Figure 3.7 compares the probability of two random blocks of 4 KiB being compactable depending on their occupancy (four sub-figures) and size classes (x-axis). CoRM performs better than Mesh in all situations. In particular, for large object sizes, CoRM succeeds even for high occupancy using only 8-bits for identifiers, whereas Mesh has near-zero

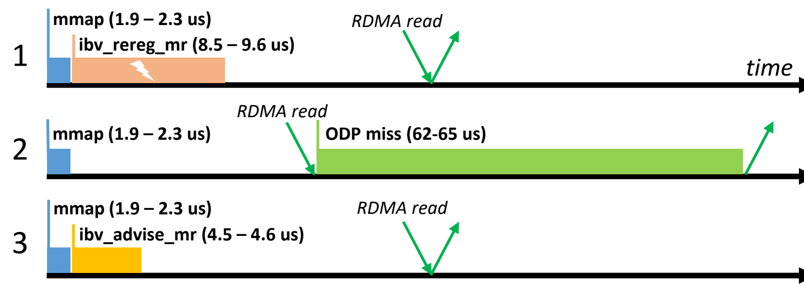


Figure 3.8: RDMA remapping latencies for three strategies.

probability. With 16-bit IDs, CoRM consistently provides a higher chance of compaction regardless of block occupancy. We conclude that CoRM is a better choice for memory compaction in DSM systems since it has a higher likelihood of compacting memory blocks even with 50% utilization.

3.3.4 Preserving RDMA access

When an RNIC receives an RDMA request, it translates the requested virtual address to a physical one using its Memory Translation Table (MTT), which contains virtual-to-physical page translation entries. Whenever a new memory region is registered, a new entry is installed in the MTT, enabling RDMA access to that region. However, if the page is remapped because of compaction (i.e., the virtual address is associated with a different physical page), then also the corresponding entry in the RNIC's MTT must be updated. Otherwise, RDMA accesses referencing that virtual address will access the wrong physical page. One possible solution is to re-register the pages every time they get remapped. However, according to the RDMA specification, this will cause the invalidation of the *r_key*: all clients would need to be informed of this event and update the *r_key* to the remote objects. A client making an RDMA access with an invalid *r_key* causes the RDMA QP disconnection, potentially leading to high overheads for recovering (e.g., re-establishing the connection), which can take few milliseconds.

To avoid these overheads, CoRM supports three approaches for restoring RDMA accesses after page remapping, both preserving the *r_key* of the original registration. The first approach relies on the *ibv_rereg_mr* call, which re-registers the memory and preserves its access keys. While this approach works on any commodity RNIC, we observed that RDMA accesses to memory regions under re-registration break the QP connection, which complies with the InfiniBand specification [10]. The second and third approaches rely

on the On-Demand-Paging (ODP) capabilities of RNICs. ODP does not compromise the connection but only works on modern RDMA devices. ODP is a technique that allows RNICs to implicitly request the latest address translation entries from the OS when pages are invalid in the MTT or if their mapping changed. ODP enables consistency between OS and RNIC translation entries. The second approach solely relies on ODP, whereas the third approach also exploits ODP prefetching to reduce the overhead of MTT misses.

Figure 3.8 shows the latency of the three solutions on a Mellanox ConnectX-5 card. In all cases, the address must be first mapped with an *mmap* call, which takes around 2 μs. With the first approach, the *ibv_rereg_mr* call takes approximately 9 μs to update the translation entry on the RNIC. During that time, the virtual address is unavailable and all remote accesses to it will cause QP disconnections. Using the second approach, the CPU does not need to explicitly fix entries in the MTT as the RNIC will resolve inconsistencies via ODP. However, we observe that the first RDMA read from a page that has been remapped incurs in a 63 μs overhead: this is the cost paid by ODP to invalidate and install the new mapping into the MTT. Subsequent reads have a 2 μs latency. To reduce the overhead of ODP MTT updates, verbs support the prefetching of MTT entries with the *ibv_advise_mr* call. The latency of prefetching is 4.5 μs, which can potentially save the ODP overhead if the read requests arrive after the prefetching completed. This is the default solution adopted by CoRM for memory registration and remapping.

3.4 Evaluation

We evaluate the performance of memory access and management operations with CoRM and its memory compaction capabilities. We first study the performance of CoRM with direct pointers and indirect pointers to evaluate performance degradation when objects move. Then we measure the performance of CoRM during compaction under synthetic and YCSB [37] workloads. Later, we evaluate CoRM's ability to compact memory and compare it with Mesh for synthetic workloads and real applications.

Experimental setup. The experiments are conducted on an isolated cluster with 12 machines interconnected with an FDR InfiniBand network. Each machine is equipped with ConnectX-3 InfiniBand network cards and two 3.40 GHz Intel Xeon E5-2630 v3 CPUs (16 hardware threads). CoRM is implemented in C++ and depends on: *libibverbs*, an implementation of the RDMA verbs; *librdmacm*, an implementation of the RDMA

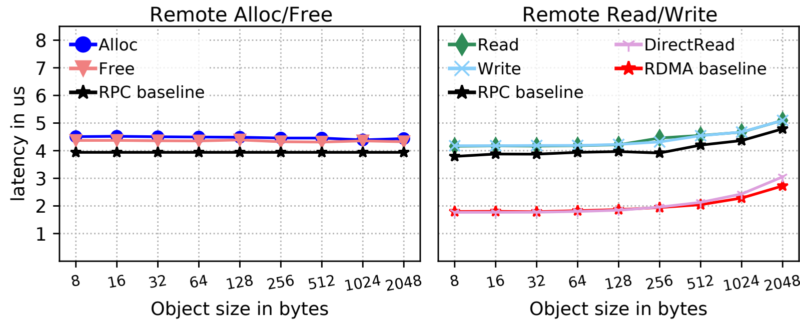


Figure 3.9: Median latency of CoRM with direct pointers.

connection manager; and *libev*, a high-performance event loop.

We deploy CoRM on a dedicated machine and spawn clients on other interconnected machines: all clients connect to CoRM remotely and send requests over the network. If not stated differently, we configure CoRM with blocks of 4 KiB and 8 worker threads.

3.4.1 Operations Latency

CoRM enables lock-free RDMA-accelerated DSMs with memory compaction capability. To achieve this, all memory accesses specified by the applications must be issued through the CoRM API and cannot directly use load/store instructions for local accesses or raw RDMA calls (e.g., ibverbs) for remote ones. We now discuss the latency of the memory access and management operations of CoRM. The operation latency is defined as the time observed by a client to complete the operation (i.e., round-trip time).

Latency of direct accesses. Figure 3.9 shows the median latency of different CoRM functions when all pointers are direct (i.e., no object has been relocated because of memory compaction). To show the overhead introduced by CoRM, we also report the round-trip latencies of RPC and raw one-sided RDMA reads. RPC operations are implemented using raw Send/Recv RDMA operations. The benchmark first loads CoRM with 10,000 objects of each size-class (≈40 MiB in total), then starts the client to issue the different operations.

The round trip latencies of RDMA requests are under 4 μs. For comparison, the TCP/IP traffic over the same link using IPoIB has a latency of 17 μs. Alloc and Free calls add about 0.5 μs to the base RPC call to manipulate the memory. However, this number accounts for the allocation case when the thread-local allocator always has a block of the requested size-class. If this is not the case, the thread-allocator needs to request a new block, increasing the allocation latency by an additional 5 μs (i.e., to allocate a new block

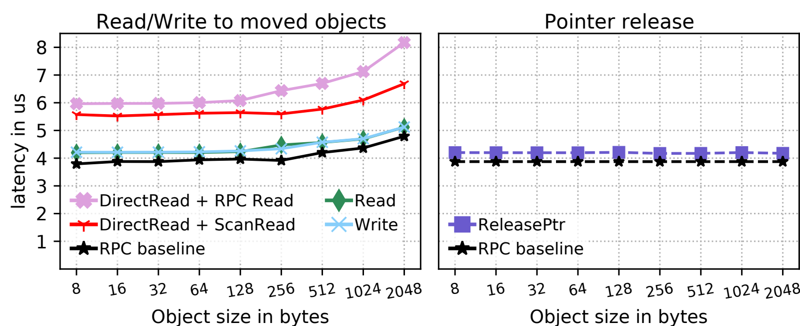


Figure 3.10: Median latency of CoRM with indirect pointers.

and register its memory on the RNIC). Read and write over RPC have similar performance as they communicate the same amount of data. The round-trip latency of the raw RDMA reads can be as low as 1.7 μs. The consistency protocol which checks the integrity of the read data in DirectReads only increases the latency for large objects. For objects smaller than 256 bytes, the DirectRead call has approximately the same latency as a raw RDMA read. These results show the benefit of using one-sided RDMA operations over Send/Recv calls for latency-sensitive read-only workloads.

Latency of indirect accesses. To measure the effects of indirect accesses, we measure the latency of read and write calls targeting compacted objects that have been moved to a different offset. In this case, the client is using an indirect pointer for accessing the objects (i.e., the pointer has an incorrect offset). Figure 3.10 (left) shows that there are no significant differences in latency between direct and indirect pointers for RPC requests. In throughput-intensive workloads, however, we observed a 5% drop in performance for RPC requests (Section 3.4.3). While RPC requests using indirect pointers can be fully handled by the CoRM workers, a failing DirectRead requires the client to perform an additional action to recover the requested object (i.e., pointer correction). We show the costs of the two pointer correction strategies: ScanRead and RPC read (see Section 3.3.1). We observe that with this configuration (i.e., blocks of 4 KiB) using an RPC call to backup a failed DirectRead is more expensive than a ScanRead. However, for large block sizes, the first approach can be more efficient because it avoids to move the whole block over the network but it would still require more CPU time on the CoRM workers in order to perform the fix.

Once a client realizes that an object has been moved and it has updated all its references to that object, it can communicate to CoRM that it is now safe to reuse the old object's virtual address (Section 3.3.2). This is done with the *ReleasePtr* call. Figure 3.10 (right)

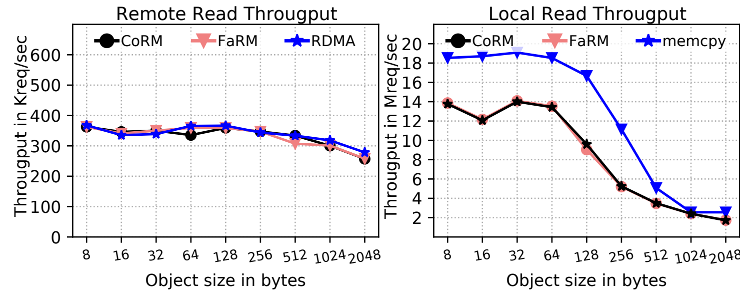


Figure 3.11: Read Throughput of CoRM, FaRM for remote and local accesses compared to direct memory accesses using RDMA and load/store instructions.

shows the latency of this operation and the one of an RPC call for a reference. The pointer release costs about 0.3 μs for indirect pointers and does not depend on the object size. This latency includes the time needed by CoRM to find the moved object (Section 3.3.1).

3.4.2 Read throughput

We now show the throughput achieved by CoRM when objects are remotely or locally accessed. We compare the read throughput achieved by CoRM with the one of FaRM[1] and the one achieved by load instructions for local reads and raw RDMA calls for remote accesses. The former scenario can be compared to an ideal Mesh-based DSM system where applications are free to use load/store instructions for local accesses. This case represents the best-case scenario for Mesh as it does not consider the additional synchronization overheads that should be introduced to keep consistency in case of concurrent read and writes.

Synthetic workload. We load the systems with a total of 8 GiB of data for each size class. The objects are accessed uniformly to ensure that data is accessed from DRAM and the clients have at most one outstanding request at a time. Figure 3.11 shows the results.

For remote accesses, raw RDMA shows the best performance (380K requests/sec for small objects) as clients do not need to verify cache versions. FaRM and CoRM have approximately the same performance as they both share the same approach for checking consistency. For small object sizes, the consistency check has negligible overhead, while it causes up to 2% slowdown w.r.t. raw RDMA for large objects. This is expected as

[1]FaRM is not open-source, therefore, we emulated FaRM (including its cacheline consistency check) following the publicly available information.

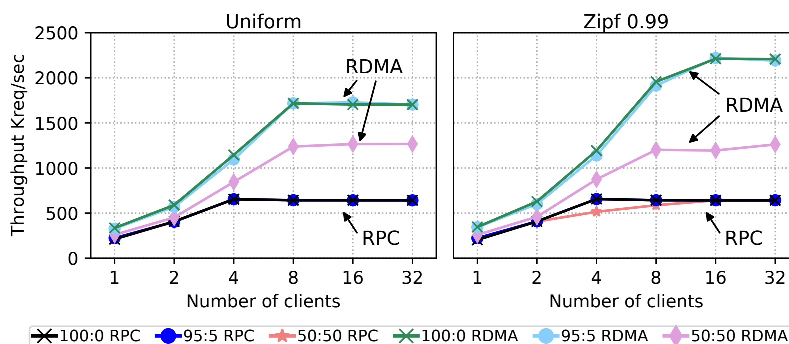


Figure 3.12: Aggregate throughput of CoRM under YCSB and uniform workloads for different read:write access ratios.

large objects require more cachelines to be checked. CoRM's cache version approach for consistency checks was a deliberate choice to mimic FaRM. An alternative approach is to store a single checksum in the header of a record or after the record [105], which could potentially be a better strategy for large records.

For local accesses, we compare the throughput of FaRM and CoRM to the one achieved by a *memcpy* call. FaRM is not more than 1.01x faster than CoRM for all sizes. FaRM and CoRM are both 1.33 times slower than *memcpy* because of the additional software layer. For larger object sizes, the performance is approximately the same for all three approaches as object accesses are memory bound.

YCSB workloads. To understand the read throughput that CoRM achieves under realistic workloads, we benchmark it under different YCSB [37] workloads. We load CoRM with 8,000,000 objects of 32 bytes and measure the throughput achieved by RPC and DirectReads. In Figure 3.12, we compare the performance of CoRM under Zipf ($\theta = 0.99$) and uniform distributions, while varying the number of clients. We report the total throughput of CoRM averaged over one minute period after a steady state under different read:write access ratios and numbers of clients. The lines tagged with RDMA use DirectRead to read objects, whereas RPC tagged lines use RPC reads. Writes are always performed using RPC.

RPC reads achieve lower throughput than RDMA reads, and the difference becomes larger for workloads with more reads. The aggregate throughput grows as we add more clients. However, in the RPC calls, the throughput stabilizes at 700K requests/sec for more than 4 clients. DirectReads allows clients to achieve 1,250K requests/sec for the 50:50 ratio, which is a 2x improvement over RPC reads. For read-dominant workloads, the throughput

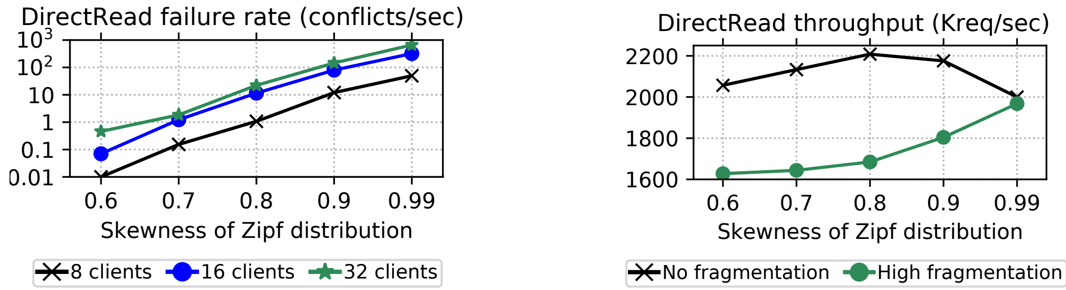


Figure 3.13: Aggregated failure rate for YCSB workloads with 50:50 access ratio.

Figure 3.14: Aggregated read throughput for YCSB workloads with 100:0 access ratio.

goes even higher up to 1,750K requests/sec and 2,200K requests/sec for uniform and Zipf distributions respectively, which shows the superiority of RDMA-based accesses over RPC. The Zipf workload shows a higher throughput because it has a better memory locality compared to the uniform distribution. This is beneficial for the destination RNIC: in fact, RNICs have limited cache for address translation entries, and once the cache is full the MTT will swap and incur in more misses [47].

DirectReads under contention. To evaluate the failure rate of lock-free one-sided reads, we measure the number of failed DirectReads over a period of one minute for the YCSB workload with 50:50 read:write ratio, while varying the skewness of Zipf distribution and the number of clients. The experiment was performed in the same setting as the previous experiment (Section 3.4.2).

Figure 3.13 shows that the number of conflicts increases with the number of clients and the skewness of the distribution. Nonetheless, even for the highly skewed workload ($\theta = 0.99$) and 32 clients, clients observed only 659 failed DirectReads per second, which is less than 0.1% of the total request rate.

Performance under fragmentation. To understand the impact of fragmentation on CoRM's throughput we benchmark its performance under YCSB workloads for two settings: *no fragmentation* (as in the previous experiment) and *high fragmentation*. To create the high fragmentation setting, we load CoRM with 16,000,000 objects of 32 bytes and then randomly deallocate the 50% of them. In Figure 3.14, we compare the performance of CoRM under the load of 8 clients, while varying the skewness factor of Zipf distribution.

Figure 3.14 shows that the unfragmented memory provides a 1.25x increase in throughput of DirectReads for moderately skewed access patterns over the fragmented one. For the highly skewed pattern ($\theta = 0.99$), CoRM's performance is approximately the same for

both settings, as the clients mostly access the same small set of keys.

3.4.3 Compaction performance

To characterize the performance the CoRM compaction process we first study its latency (i.e., the time needed to perform the memory compaction), then analyze how the throughput of remote clients is affected when the server is busy in performing the compaction.

Compaction latency. The compaction process is composed of two phases: block collection and block compaction. During the block collection phase, the thread performing the compaction gathers blocks that are candidates for compaction (Section 3.3.1): i.e., non-full and belonging to the same size class. Figure 3.15 shows the time for these two phases. Since block compaction involves page remapping, which depends on the specific RNIC and remapping strategy (Section 3.3.4), we compare the compaction latencies measured on machines equipped with ConnectX-3 and ConnectX-5 with *ibv_rereg_mr*, and ConnectX-5 with ODP and prefetching. The block collection phase involves inter-thread communications, hence we compare two different CPUs: Intel Xeon (Section 3.4) and AMD EPYC 7742 @ 2.25GHz. Each experiment consists of allocating a single object of 32 bytes on each thread and then triggering the compaction process in order to measure its latencies.

Block collection. The collection time depends on the number of threads as all threads must reply to the collection request before the compaction can happen. In this experiment, each thread replies with its only allocated block to the compaction request. The results are shown in Figure 3.15 (left). On the Intel cluster, the collection takes 10 μs for 2 threads and 31 μs for 16 threads We notice that the collection takes on 2 μs on the AMD cluster, when 2 threads are used, which is 5x times faster than Intel. The two clusters show similar latencies when increasing the number of threads.

Block compaction. The compaction time depends on the block size and the number of blocks that can be compacted. In particular, it involves the checks of compactability requirements, the data copy, and the virtual address remapping on both the OS and the RNIC. The compaction time of a single block corresponds to the unavailability period of that block as the data in this block is not accessible because of compaction. In this experiment, blocks under compaction have only one object and are always compactable (i.e., no conflicts). Thus, the number of deallocated blocks after compaction is equal to the number of threads minus one.

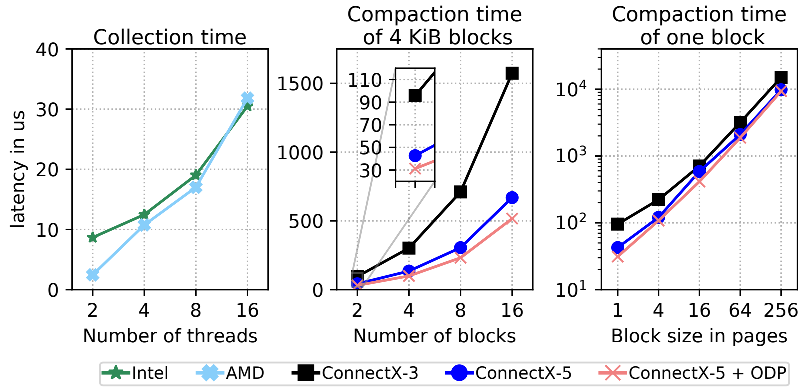


Figure 3.15: Latencies for collection and compaction for different numbers of threads and block sizes.

Figure 3.15 (center) shows how the block compaction latency varies with the number of blocks to compact. The block size is fixed to 4 KiB (i.e., one page). The compaction time grows linearly with the number of blocks (note that the x-axis is exponential). With ConnectX-3, the block compaction takes about 100 μs, with most of the time (70 μs) spent into the *ibv_rereg_mr* call. The same call on ConnectX-5 takes 7 μs: since this call is executed for each remapping, the difference in performance increases with the number of blocks to compact. The ODP strategy provides the best results as it does not require explicit memory re-registration.

Figure 3.15 (right) shows the compaction latency of a single block for different block sizes. The bigger the block size, the more physical pages need to be remapped. For 1-page blocks, the block compaction takes 100 μs for ConnectX-3, and the time grows linearly with the number of pages. For 1 MiB blocks, consisting of 256 pages, the compaction takes 12 ms for ConnectX-3. ConnectX-5 can remap pages faster, but the cost of the re-registration has the same trend as ConnectX-3. In case large blocks are used, the remapping time can be significantly reduced by using huge pages. For example, a 2 MiB page has the same remapping and re-registration latency as a 4 KiB page. Therefore, all reported data is applicable to huge pages.

Throughput during compaction. We evaluate the performance degradation of CoRM during compaction. In this experiment, we populate CoRM with 8,000,000 objects of 32 bytes and randomly deallocate the 75% of them. Then we start the throughput workload that repeatedly and sequentially reads all objects. After the warm-up, we trigger the compaction algorithm. In all experiments, the compaction is invoked after two seconds.

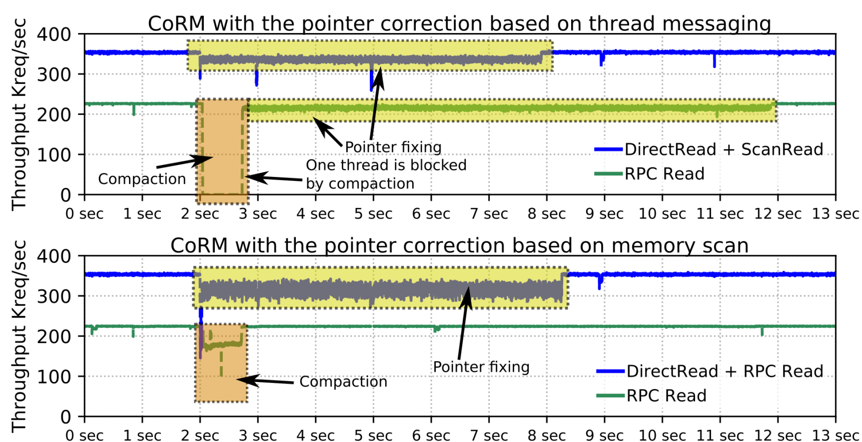


Figure 3.16: Read throughput of RPC and RDMA clients before, during, and after compaction. During the compaction, CoRM compacts 5,794 blocks. *(top)* Pointers are corrected with thread messaging; *(bottom)* pointers are corrected with the block scan strategy.

We measure the throughput of a single client *before*, *during*, and *after* compaction.

Figure 3.16 shows the read throughput observed by clients accessing data with different types of read calls (i.e., RPC and RDMA) and different pointer-fixing strategies. We also study the performance of the two different approaches for pointer-correction, which is used when an object is accessed with an indirect pointer (see Section 3.3.1).

Thread messaging. We measure the throughput of two types of clients: one using RPC reads, and the other using DirectReads (RDMA). In the first experiment (top sub-figure), CoRM is configured to use thread messaging to find objects which are requested using RPC calls. The RDMA client needs to recover by itself in case the read fails (i.e., because the object has been moved). Here we show the case where the client issues a ScanRead to back up a failed DirectRead. The RPC client observes 700 *ms* of unavailability, as a requested object has been moved to a different offset and CoRM could not find it by using the passed pointer. The reason for that is the thread that owns the block with the moved object was busy with compaction and could not reply to correction requests from other threads. It happens because all blocks under compaction belong to the same thread that performs the compaction. We intentionally configured CoRM to perform long compaction without breaks to observe that scenario. During this long compaction, the thread managed to deallocate 5794 blocks. The unavailability period could be shorter if the compaction was configured with an upper bound on the number of compacted blocks. The RDMA client, on the other hand, does not observe the unavailability as it corrects

the pointers using ScanRead, which reads the whole block with a requested object using RDMA and then scans it. After 8 seconds the RDMA client managed to correct 77,000 indirect pointers. The RPC client spends almost 9 seconds for all corrections. Overall, both clients observe 5% performance drop during pointer correction.

Memory scan. In the second experiment (bottom sub-figure), CoRM's worker threads opt for scanning the whole block to find the object requested using RPC. The RDMA client in this experiment uses RPC to correct pointers, instead of ScanRead. Compared to the previous experiment, the RCP client does not experience large unavailability periods, showing a 22% slowdown in throughput due to the fact that blocks under compaction are not readable during object migration and remapping. Once the compaction finishes, the client does not observe any significant performance drops even using indirect pointers, since a small percentage of pointers were indirect. In a similar experiment where all pointers are indirect (not reported here), the client observed only 5% slowdown. The RDMA client experiences performance degradation as it needs to issue an RPC to correct pointers to moved objects. This approach is slower than the one with ScanRead, which complies with our latency experiments. Overall, DirectReads provides 1.6x faster performance than RPC reads even in the presence of indirect pointers and when CoRM performs compaction.

Conclusion. The current experiments show that CoRM's pointer correction introduces only a *temporary* slowdown of 5% on read performance, whereas the use of indirection tables and RPCs has a *constant* 40% slowdown since it prevents the use of one-sided RDMA reads for fetching the data.

3.4.4 Compaction overheads and benefits

To enable its compaction strategy and reuse virtual addresses, CoRM stores an object identifier with each object. The object identifiers serve two scopes: (1) they enable CoRM to find relocated objects; (2) they allow detecting reads using pointers to relocated objects (i.e., same offset, different object ID). The size of the object identifiers determines the space-overhead per block and the capacity of CoRM of compacting blocks with a large number of objects. We define CoRM-n as the instance of CoRM using n bits for the object identifiers. In this experiment, we study the effects of the CoRM compaction strategy for synthetic and real-world workloads using different object identifier sizes and comparing the results with the compaction strategy proposed by Mesh.

Mesh	CoRM-0	CoRM-8	CoRM-12	CoRM-16
0 bits	28 bits	28+8 bits	28+12 bits	28+16 bits

Table 3.3: Memory overheads for compaction algorithms per object for 1 MiB blocks.

Object identifiers overhead. Table 3.3 summarizes the space overheads for different CoRM configurations for 1 MiB blocks, which is the size of blocks in FaRM. In CoRM-0, object IDs are disabled and the compaction strategy is based on object offsets, as in Mesh. However, while the overhead of the object IDs is zero, CoRM still stores the virtual block address in the header of each object to be able to reuse virtual addresses (Section 3.3.2). Assuming a system with 48-bit memory pointers and 20-bits aligned blocks, the virtual block address that CoRM needs to store is 28 bits.

The CoRM's compaction algorithm cannot compact blocks storing more objects than the ones that can be addressed with a given identifier size. For example, CoRM-8, with 8 bits for object identifiers, can address up to 256 objects in a block, hence it cannot be used for 1 MiB blocks storing 2 KiB objects, which can hold up to 512 objects. To handle these cases, CoRM can be configured to use a hybrid compaction scheme where class sizes that cannot be compacted by CoRM-n are compacted with CoRM-0.

Synthetic traces. Figure 3.17 shows the amount of memory currently allocated (i.e., active memory) after a sequence of memory operations. We generate synthetic traces that first allocate 8 M objects of a given size (sub-figures) and then randomly deallocate a fixed portion (x-axis) of them. We also plot the active memory in case an ideal memory compactor, which always frees the non-utilized memory, and in the case in which no compaction is performed. We study the compaction capability of CoRM for 8, 12, and 16-bit object IDs and include the memory overhead introduced by CoRM to store object IDs in the reported data.

As the object sizes increase, both Mesh and CoRM are able to compact more memory. Mesh works effectively for large objects with high fragmentation but incurs in many conflicts for smaller objects and low deallocation ratios. CoRM-8 and CoRM-12 perform always better than Mesh for the object sizes where they can be applied (e.g., $\geq$ 4 KiB objects for CoRM-8). CoRM-16 matches the ideal compactor already for 2 KiB objects and low deallocation rates (i.e., 0.5): this is because larger object IDs reduce the probability of conflict, hence increase the probability of compaction (object IDs are randomly chosen). For 256-byte objects, CoRM-16 requires more active memory than the non-compacting

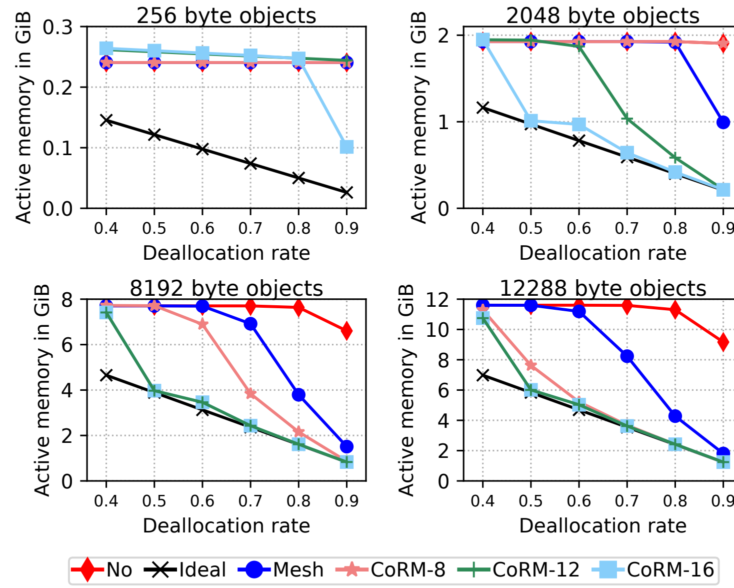


Figure 3.17: Active memory under synthetic workloads. CoRM is configured with 1 MiB blocks.

case. This overhead comes from the fact that blocks with 256 byte objects have a very high probability of collision in object identifiers. Therefore, CoRM's overheads overwhelm its compaction capabilities.

Redis traces. Redis [135] is a popular in-memory data structure server. We extract memory traces from the *memefficency* unit test of the Redis test suite (v5.0.7):

- *redis-mem-t1*: Default Redis configuration. It allocates 10'000 keys of 8 bytes each with values of sizes ranging from 1 to 16 KiB.
- *redis-mem-t2*: Redis is configured as an LRU cache with a capacity of 100 MiB. The trace first allocates 700,000 8-byte keys with values of 150 bytes each; then allocates 170,000 8-byte keys with values of size 300 bytes each.
- *redis-mem-t3*: Default Redis configuration. The trace allocates 5 keys containing data structures of 160 KiB each; then it allocates 50,000 keys with values of 150 bytes. It then removes 25,000 keys from the last batch of allocated keys.

For each trace, we report memory usage for a time point when the system had the highest fragmentation. To show the effects of the number of threads on the active memory, we run the traces multiple times, varying the number of threads used by the memory allocator: i.e., these are the treads serving RPC requests in CoRM. For each allocation request, the thread is selected randomly.

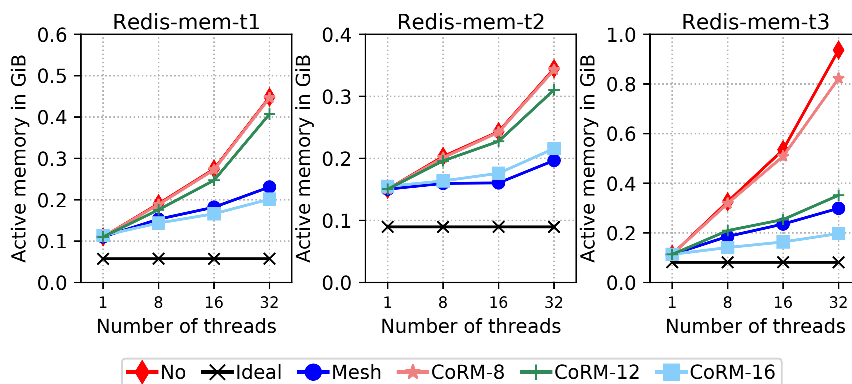


Figure 3.18: Active memory under Redis workloads. CoRM is configured with 1 MiB blocks.

Vanilla CoRM. Figure 3.18 shows the memory usage of the Redis traces under different compaction strategies. In this experiment, CoRM disables compaction for blocks that can store more objects than the ones that can be addressed with the given identifier size. For reference, we include the active memory kept by an ideal compaction strategy and in case no compaction is performed. CoRM introduces up to 4 MiB memory overhead (i.e., for CoRM-16) to store object IDs. The reported data includes this overhead.

Redis workloads do not experience allocation spikes as in the synthetic workload. Therefore, none of the algorithms can significantly reduce the active memory for a single-threaded allocator. However, the traces exhibit high fragmentation due to the low usage of some size classes. Depending on the trace, the difference in active memory between 1-thread and 32-thread allocators ranges from 3x to 12x. This increase of fragmentation is explained by the fact that with more threads there is a higher possibility of conflicts (either on the block IDs or on the offsets). The cases where Mesh performs better than CoRM are the ones where CoRM cannot compact blocks because of their large object count: e.g., CoRM-12 can compact only blocks with objects larger than 256 bytes. With 16-bit per object ID, CoRM-16 provides better memory efficiency for *redis-mem-t1* and *redis-mem-t3*. For *redis-mem-2*, Mesh manages to compact more memory than CoRM-16. Also in this case, this happens because the trace allocates objects that CoRM-16 cannot compact (i.e., 8 bytes, while CoRM-16 can compact blocks with at least 16 bytes objects). CoRM-20, which is not plotted here, manages to compact more memory than Mesh since it supports the aforementioned size class. This suggests that the object IDs should be tuned for particular workloads in order to maximize memory efficiency.

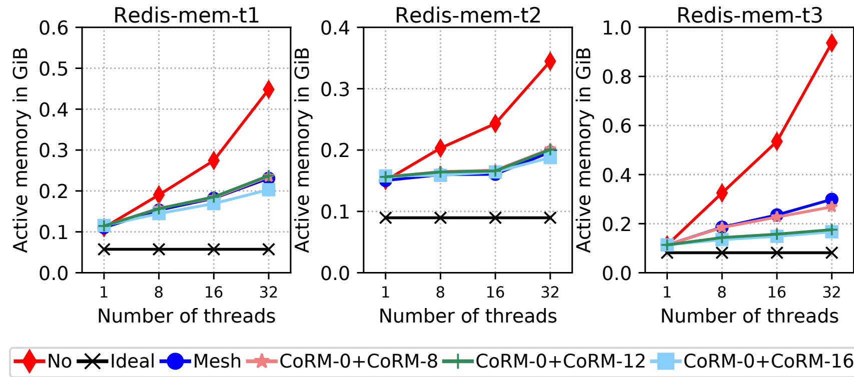


Figure 3.19: Active memory under Redis workloads. CoRM is configured with 1 MiB blocks and in hybrid mode.

g

Hybrid CoRM. To remove the negative effects of blocks that cannot be compacted with a given object identifier size, we adopt the hybrid compaction strategy described in Section 3.4.4. Figure 3.19 shows the active memory kept by the different compaction strategies with this setting, using the same Redis traces of Figure 3.18. The data shows that hybrid configuration has better compaction performance compared to pure CoRM and pure Mesh algorithms.

In all experiments, hybrid CoRM is at least as good as Mesh in terms of active memory. For *redis-mem-t1* and *redis-mem-t2*, CoRM-16 provides an improvement of 12% and 5% over Mesh, respectively. The key difference is that CoRM, other than being at least as good as Mesh for memory compaction, it enables RDMA-accelerated DSMs while being able to release and reuse virtual addresses.

Discussion. To take full advantage of CoRM's compaction capabilities, users can tune object ID sizes for different size-classes, according to the specific workloads. Ideally, applications would label class sizes with an indication of how frequently they are used. Highly-used classes would likely not benefit for compaction since their frequent allocations and deallocations would already avoid fragmentation. Instead, class types that are not frequently allocated, can be managed by CoRM which will be able to compact them while introducing a space overhead given by the object IDs. We consider an auto-labeling strategy of class sizes as future work.

3.5 Related Work

RDMA-accelerated DSM systems: FaRM [47] is a distributed memory computing platform that exploits RDMA to read remote objects. FaRM does not support memory compaction and addresses the problem of unpopular size-classes by pinning them to specific thread allocators. While this solution mitigates memory fragmentation due to unpopular size classes, it does not help to limit fragmentation due to allocation/deallocation spikes.

GAM [28] is a shared memory system that exploits RDMA to accelerate its cache coherence protocol. Unlike FaRM, GAM does not allow objects to be read using RDMA but it used RDMA for updating its shared buffer state. Like FaRM, GAM does not support memory compaction and can benefit from CoRM's compaction algorithm without compromising its RDMA functionalities.

RDMA-enabled systems with compaction: RamCloud [134] and MICA [92] are key-value stores employing log-structured memory allocators to limit fragmentation. The main drawback of this approach is that deleted objects occupy memory until they are garbage-collected. To free a memory block, the garbage collector copies alive objects from it to the tail block of the log-structured memory. As objects frequently move in memory, RamCloud and MICA use indirection tables. Therefore, MICA's RDMA-accelerated extension, HERD [70, 71], and RamCloud cannot directly access objects using one-sided RDMA and focus on accelerating RPC calls.

3.6 Summary and Discussion

We introduce CoRM, a prototype of a remote memory system that employs RDMA to accelerate remote reads and, at the same time, supports memory compaction to provide high memory efficiency. CoRM's memory accesses are strongly consistent even in the presence of concurrent compaction. In case of fragmented memory, CoRM is at least as good as Mesh in compacting memory, while saving up to 2.8x more memory w.r.t. Mesh in cases where allocation/deallocation spikes occur for large objects. The novel compaction algorithm of CoRM, based on object IDs instead of offsets, does not use indirection tables and completely relies on OS virtual address translation. All in all, CoRM fills a gap in RDMA-accelerated shared memory systems by avoiding the need for compromising memory efficiency for performance.

4

Zero-copy Data Access for Apache Kafka over RDMA

Apache Kafka [8] is an open-source distributed publish-subscribe system, which is widely used in data centers for messaging between applications, log aggregation, and stream processing. The existing Kafka implementation uses TCP/IP for communication, which has various inefficiencies such as a high message dispatch cost due to OS involvement and excessive memory copies. Recently, the availability of cost-effective RDMA-capable network controllers within data centers and cloud infrastructures have encouraged many modern applications to adopt RDMA networking, which offers the potential to outperform classical TCP/IP. The lack of richness of RDMA operations poses challenges to its efficient use, forcing many applications to use intermediate buffers or multiple round trips in their protocols.

The work in this chapter explores the following research questions:

- What are the main overheads in the existing data-intensive datapaths of Apache Kafka?

- What is the most efficient way of using existing RDMA features to accelerate Apache Kafka?
- What contribution to performance each RDMA-enabled datapath brings?

To address these questions we introduce KafkaDirect, an extension to Apache Kafka, that uses RDMA to accelerate the three most network intensive datapaths: record production, record replication, and record consumption. The RDMA modules of KafkaDirect can be enabled at need, allowing us to evaluate the contribution of each proposed optimization to the overall performance.

In this chapter, we also explore the design choices of KafkaDirect including which RDMA operations to use to take full advantage of offloaded communication. Our RDMA design aims to attain true zero-copy communication completely avoiding the need for using intermediate buffers in Kafka servers, thereby ensuring low latency and high throughput communication.

This work was done during an internship at Oracle Labs in 2019. The work in this chapter was done in collaboration with Steve Byan and Virendra Marathe from Oracle Labs and my advisor Torsten Hoefler.

4.1 Motivation

Decreasing prices on RDMA-capable network controllers (RNICs) have made them widely available in data centers and cloud infrastructures. The availability of RNICs has encouraged many modern applications (e.g., deep learning and data analytics frameworks) to adopt RDMA networking to gain higher throughput and lower latency compared to the traditional TCP/IP stack. RDMA offers these gains by offloading most of the networking functionality to the RNIC, effectively bypassing the OS kernel.

The naive use of RDMA, nonetheless, is unable to achieve maximum performance. The high-level frameworks such as RPCs [21] that hide direct memory communication primitives from the user may struggle to achieve even 20% of link bandwidth and have much higher latency and CPU usage than promised by the RNIC specification [169]. Therefore, modern RDMA-accelerated applications implement specialized protocols to take full advantage of RDMA networking. The lack of richness of RDMA operations poses challenges to its efficient use, as one-sided RDMA operations can only read and write a remote memory location. RDMA does not support more sophisticated operations such as conditional

and compound operations, though such RDMA primitives have been proposed [57, 16, 89]. Therefore, many applications are forced to use intermediate buffers or multiple round trips in their protocols. Although these protocols have been extensively studied for general key-value stores [70, 89, 105], not many RDMA solutions have been proposed for log-structured storage systems [111, 71] such as Apache Kafka.

In this chapter, we explore the most efficient way of using existing RDMA features to accelerate Apache Kafka [8], a publish-subscribe system, which performance is currently constrained by overheads in the existing TCP datapaths in the form of RPC infrastructure, CPU wakeup latency, and superfluous buffering of data. All these issues get into the way of having a tightly streamlined datapath between the various components of Kafka.

The design of our system, KafkaDirect, is inspired by the fact that general-purpose request processing is expensive due to excessive data copies. Since zero-copy request processing is crucial for CPU-intensive systems, such as Kafka, we propose to remove data copies introduced by the TCP/IP stack and general-purpose request processing by offloading CPU-intensive operations to RNICs.

Challenges. The effective use of offloaded RDMA networking for Kafka raises numerous challenges: 1) how to achieve true zero-copy communication that avoids intermediate buffering, 2) how to empower Kafka consumers to read records without the involvement of the CPU of Kafka brokers, 3) coexistence of RDMA and TCP datapaths in Kafka without obstructing its usability and performance. Our work effectively solves the aforementioned technical challenges and provides an extensive investigation of the space of possible design decisions, that can be extended to other log-structured and publish-subscribe systems (Section 4.5).

Design. KafkaDirect empowers clients to write records directly to storage using RDMA. KafkaDirect can ensure consistent writes to the same topic from multiple producers by employing RDMA atomic operations. Unlike the original Kafka, KafkaDirect follows a push approach for data replication to write records directly to the memory of replica servers. Consumers in KafkaDirect exploit RDMA Reads to directly read records from subscribed topics, completely bypassing the CPU of Kafka brokers and thereby significantly reducing their CPU usage. KafkaDirect delivers low latency and high throughput without modifications of existing data-formats, preserving backward compatibility. The RDMA modules of KafkaDirect can be enabled at need, allowing us to study the performance of each RDMA-accelerated module.

Contribution. KafkaDirect outperforms the existing Kafka systems in terms of both bandwidth and latency for all datapaths. The latency to the RDMA producer client can be as low as 80 μs, which is a 4x improvement compared to unmodified Kafka deployed over IPoIB networks. The RDMA producer client can achieve 4.5 GiB/sec for producing records to a single topic, which is a 9x improvement over today's Kafka producer bandwidth. KafkaDirect's replication module offers high-bandwidth replication which provides a 13x improvement in replication performance. In addition, the RDMA Kafka consumer offers a 50x reduction in latency and a 10x increase in throughput. Finally, the RDMA Kafka consumer offloads processing of Kafka fetch requests to the network controller, allowing the system to serve thousands of clients with no CPU cost.

4.2 Background on Publish-Subscribe systems

Publish-Subscribe messaging systems provide simple but powerful abstractions enabling asynchronous data transfer between applications. Applications that communicate through the messaging system are divided into publishers and subscribers. A publisher application appends records to a message queue, and subscribers can subscribe to the message queue to receive all published records.

Publish-Subscribe systems are a popular building block for many modern data center applications [53, 162, 163], as they shift the burden of reliable messaging from communicating applications. Publish-Subscribe applications are available as open-source systems (e.g., Apache Kafka [8, 82], Corfu [14], Scalog [44], Fuzzylog [94]) and as a service by various cloud providers [138, 101, 63, 110]. Despite the rich diversity in applications implementing the abstraction, we find that they share similar functionality and data organization. Their records are stored as a sequence of ordered records in append-only data logs, that are replicated to ensure fault-tolerance against machine failures. As the systems have similar storage designs, we only focus on Apache Kafka, but the RDMA design could be borrowed by other systems (Section 4.5).

Apache Kafka. Kafka [8, 82] is a fault-tolerant distributed publish-subscribe messaging system. A Kafka's publisher is called a *producer* that pushes records to containers called Kafka *topics*. A Kafka's subscriber, called a *consumer*, subscribes to Kafka topics to fetch the produced records.

Kafka Topics. All records in Kafka are categorized into topics that are partitioned into

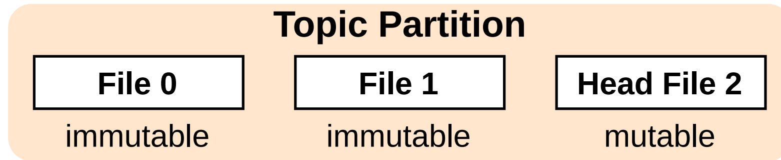


Figure 4.1: A topic partition may consist of multiple files. New records are appended to the head file. All preceding files of the topic partition are sealed and cannot be modified.

multiple partitions called *topic partitions* (TPs). Each TP is an ordered, immutable sequence of records that constitutes a log. The records in the TP are labeled with sequential ID numbers called the *Kafka offset* that uniquely identifies each record within the TP. The offset is a sequential value that Kafka linearly and uniquely assigns to each record as it is appended to a TP. Logically, a TP can be viewed as a contiguous append log. However, physically it is comprised of segments that are distinct files stored on disk (see Figure 4.1). New records are always appended to the head segment of the log, which is limited in size (1 GiB by default). When the head segment becomes full, Kafka seals the file and creates a new head file to store new records.

Each TP can be replicated across a configurable number of servers for fault tolerance. In this case, one server acts as the replication *leader* and one or more servers act as replication *followers*. The leader handles all read and write requests for the partition while the followers passively replicate the leader. A record is not considered committed until it is fully replicated to all in-sync replicas.

Kafka Broker. A broker is a storage server of the Kafka cluster. The broker receives records from producers, assigns offsets to them, and commits the records to storage on disk. It also services consumers, responding to fetch requests for its TPs and responding with the records that have been fully replicated. Each Kafka broker can process multiple TPs and act as a replication leader for some of its TPs and a follower for others to balance the load within the cluster.

4.3 RDMA design for Kafka

KafkaDirect extends Kafka with efficient RDMA networking without compromising its original API and data formats. Our RDMA modules are carefully integrated into Kafka and provide acceleration of the three most intensive datapaths of Kafka: record production,

replication, and consumption. The main design principle of KafkaDirect is to offload request processing of those datapaths to RNICs.

Unlike a naive use of RDMA that replaces TCP/IP sockets with two-sided RDMA networking, we aim to use one-sided RDMA accesses to directly access data stored on Kafka Brokers. Our design is inspired by the main observation that *general-purpose RPCs are expensive* for data-intensive requests. Even though many variations of efficient implementations of RPCs over RDMA [72, 144, 33, 146, 47] exist, their performance can still suffer from the RPC abstraction [169] that induces additional memory copies: an RPC initiator needs to copy RPC arguments to network send buffers, and an RPC executor needs to unpack received arguments from network receiver buffers. The problem becomes especially severe for storage applications that communicate large data volumes that should be stored in or read from remote storage. The requirement to copy data from the network receive buffers to storage structures can further aggravate the already well-known CPU-bottleneck problems encountered in many distributed applications [112, 158]. As a result, the CPU-intensive systems suffering from excessive data copies can be accelerated only by fully offloading request processing from the CPU to network controllers.

The only known to us RDMA-enabled implementation of Kafka, OSU Kafka [107], uses two-sided RDMA Sends to replace the TCP/IP network module of Kafka and does not use one-sided RDMA requests to directly access records. Thus, its performance is still obstructed by the need to copy messages from and to network buffers of the multipurpose request processing module. In Section 4.4 we show that KafkaDirect significantly outperforms OSU Kafka, showing the importance of our zero-copy design.

Overview. A graphical overview of KafkaDirect's broker architecture that makes the best use of RDMA networking is presented in Figure 4.2. KafkaDirect has a dedicated RDMA network module (Section 4.3.1) to serve RC QP connections from clients and brokers. We extend Kafka with our RDMA Produce module (Section 4.3.2) allowing producers to exploit RDMA WriteWithImm to write data directly to TP files and notify the broker about the incoming produce requests. KafkaDirect provides high-performance exclusive RDMA produce requests (Section 4.3.2) that do not require coordination between producers. For shared access, KafkaDirect can ensure consistent writes to the same topic from multiple producers by employing RDMA atomic operations. KafkaDirect enables low-latency data replication by adding an RDMA push replication module (Section 4.3.3) that uses one-sided RDMA writes to replicate data directly from replication leaders to replication followers. Finally, consumers in KafkaDirect exploit RDMA Reads to directly read records

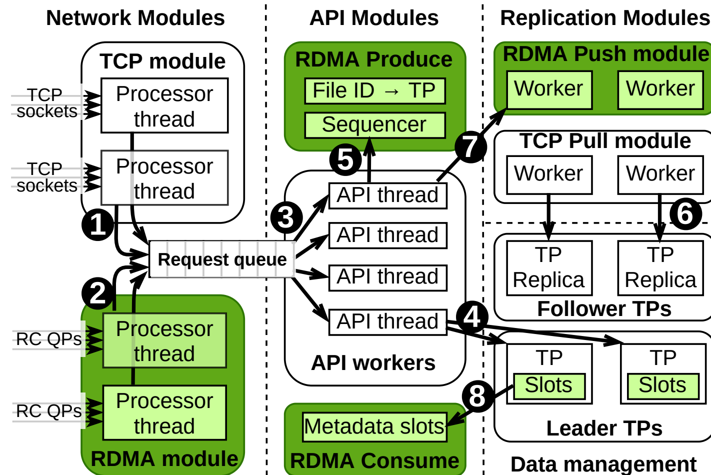


Figure 4.2: Kafka's broker architecture and our RDMA extensions (in color).

from subscribed topics (Section 4.3.4), completely bypassing the CPU of Kafka brokers and thereby significantly reducing their CPU usage. KafkaDirect's consumer module employs RDMA-readable metadata slots, that contain information about TP files, allowing consumers to get informed about new records without the broker's CPU involvement.

4.3.1 Network Layer

The original Kafka uses TCP connections to send and receive requests to and from clients and brokers. When a broker receives a request via TCP, one of its network processor threads will enqueue the request ❶ to the shared request queue (see Figure 4.2). The request will be later fetched ❸ and executed by one of the API worker threads.

KafkaDirect completely reuses Kafka's TCP module for processing all the original Kafka requests, thereby ensuring backward compatibility. KafkaDirect has an additional RDMA network module that serves RC QP connections for only processing RDMA accelerated datapaths. Its thread workers poll shared RDMA completion queues of established QPs to get RDMA completion events. Once a thread fetches a completion event, it will enqueue ❷ the corresponding request to the shared request queue.

KafkaDirect uses reliable (RC) instead of unreliable (UD) RDMA transport for two reasons. First, unlike other transports, RC supports one-sided RDMA Reads and Writes, which are exploited in our produce (Section 4.3.2) and consume datapaths (Section 4.3.4)

for zero-copy data accesses. Second, our replication and produce datapaths rely on the delivery guarantees of reliable transport to pipeline multiple produce requests (Section 4.3.2).

4.3.2 Produce datapaths

TCP produce datapath

An original Kafka producer sends records to brokers using a *produce* request that contains records and how many times the records must be replicated before receiving an acknowledgment. The broker verifies the received records and then appends ❹ them to the corresponding TPs.

The shortcoming of the existing TCP produce datapath is that *the broker performs two redundant memory copies to persist new records.* The first data copy is performed by the TCP network stack. The driver copies all received messages from its receive buffers to Kafka's receive buffers. The second copy is made by the broker when it copies data from the network receive buffer to the file buffer.

RDMA produce datapath

The high-level idea of the KafkaDirect produce datapath is to use RDMA to write records directly to remote TP files. This approach eliminates the need for performing the two aforementioned data copies. By omitting the memory copies, we aim to improve the performance of the producers.

Getting RDMA access. To get RDMA access to the head file of a TP, an RDMA producer sends a request via TCP that enables RDMA access to the head file by mapping it to the main memory (using *mmap*) and registering it with the RNIC (using *ibv_reg_mr*). Since RNICs are not able to append data to files and only can write data to an already preallocated memory region, we enable the file preallocation in Kafka's configuration. The response from the broker contains the RDMA connection string and the virtual address and the full length of the preallocated head file. Having the length allows the producer to prevent writing beyond the allocated area and to timely request allocation of a new head file.

Approaches to RDMA produce

We propose two produce algorithms that provide different access permissions: *exclusive RDMA access, and shared RDMA/TCP access*. Both approaches exploit RDMA Write-WithImm work requests to notify the broker of incoming Write requests. Note that Write-WithImm allows the sender to piggyback a 32-bit value, called *immediate data*, that is included in a corresponding completion event at the destination.

The main shortcoming of WriteWithImm is that the destination address of an incoming buffer is unknown to the receiver and is fully chosen by the sender. The completion event at the destination only contains the number of written bytes and the 32-bit immediate data. We address this problem by encoding where the data has been written into the immediate data. In KafkaDirect, when a producer requests RDMA access to a file, the broker accesses ❺ the RDMA produce module to generate a unique 16-bit ID for the requested file and sends it to the producer. The producer includes the ID in the immediate data of WriteWithImm to inform the leader to which TP the data has been written (see Figure 4.4). Once the completion event is received and enqueued ❷ to the shared request queue, one of the API worker threads fetches ❸ the request and maps the file ID to the requested TP by accessing ❺ the RDMA produce module. After that, the API worker can request ❹ the file from the data management module and perform verification of the written data. If the written data complies with the integrity checks, the broker commits the records by advancing the Kafka offset of the TP.

Exclusive RDMA access. In this mode, only one RDMA producer can publish records to a TP. For that, the producer contiguously writes records to the head file using Write-WithImm using the file ID as immediate data. Importantly, WriteWithImm to the same file must be processed sequentially and in the same order as they have been written to the file. Otherwise, a race condition could occur if two writes were processed in the opposite order by thread workers ❸. KafkaDirect solves this problem by processing RDMA produce requests in the same order as the corresponding completion events are generated ❺. We rely on InfiniBand's in-order delivery guarantees that ensure the correct order of completion events.

The datapath is consistent as long as the TP is written by a single remote producer, which is enforced by the broker. The broker never grants exclusive access to the same file to two producers. If the RDMA producer fails, its exclusive RDMA access will be revoked. Client failure can be detected from QP disconnection events. To avoid the situation where

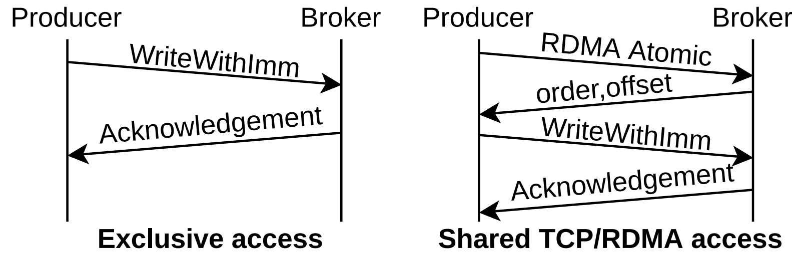


Figure 4.3: Methods for producing records via RDMA. In the exclusive mode, the single producer tracks the offset. In the shared mode, producers acquire the offset before they write.

a faulty client still accesses the memory of a TP file, the broker can disable RDMA access to the file.

Shared RDMA/TCP access. In this mode, multiple producers can publish the records to a single TP, and publishers can use either TCP or RDMA operations for that. The main challenge of shared RDMA/TCP mode is to ensure concurrent and consistent writes to the same TP file from multiple producers. KafkaDirect solves this problem by employing RDMA atomic operations such as RDMA compare-and-swap (CAS) and fetch-and-add (FAA) to achieve agreement between writers to the same file. The broker associates with each TP an 8-byte value that stores the current producer order (first 2 bytes) and the current offset in the file (remaining 6 bytes) as depicted in Figure 4.5.

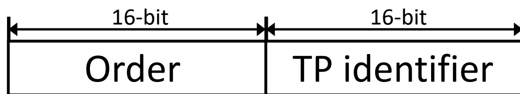


Figure 4.4: The 32-bit immediate value used to inform broker where the records has been written with WriteWithImm.

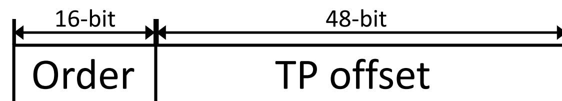


Figure 4.5: The 64-bit atomic value used to enforce ordering across producers. They must atomically fetch and increment the current order and offset before writing records.

Before writing data to a TP using WriteWithImm, a producer should reserve a memory region within the file where it can write the records. For that, the producer atomically fetches the 8-byte value associated with the file and increments its order field by one and its offset field by the size of the record it intends to write. In response, the producer retrieves the start of the region it can write to and its order.

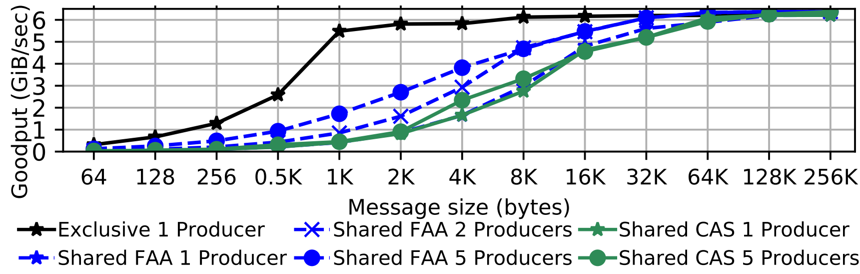


Figure 4.6: Aggregated Write bandwidth of different RDMA produce approaches.

The fetched order field is used to enforce order in the produce requests from different producers, and it must be included in the immediate data of the subsequent WriteWithImm request (Figure 4.4). Even though the maximum size of the Kafka file is 4 GiB, the current file offset field is 6 bytes allowing the producers to detect overflow of the field when RDMA FAA is used. RDMA FAA always succeeds and, therefore, producers can exceed the actual file size, which can be detected by the producers by checking these extra 2 bytes. When a broker receives a produce request via TCP to an RDMA-accessible file, it also needs to reserve a memory region by issuing an RDMA atomic to itself to ensure a consistent view between the broker and remote clients.

Shared RDMA/TCP access can be potentially damaged by client failures since holes can appear in the TP file when a client wins a segment in the file and then fails to fill it through RDMA due to crashes or slowdowns. KafkaDirect prohibits holes in the TP file by detecting failed RDMA produce requests using the order encoded in the immediate data. The RDMA produce module ➎, which is responsible for ensuring the correct processing order of RDMA requests, prevents processing a produce request i, if the request $i-1$ is not processed. The RDMA produce module sets a timeout to each incoming RDMA produce request which should wait for the arrival of preceding produce requests. If a produce request is timed out it gets aborted and RDMA access to the file is revoked causing abortion of all pending produce requests to the file. Clients can then re-enable the RDMA datapath by requesting RDMA access again.

Performance comparison. Figure 4.6 shows goodput for produce requests for different record sizes. For shared accesses, we measure the aggregate goodput for two and five producers. The microbenchmark is implemented in C/C++ and is not a part of Kafka. The goal of this experiment is to show the performance upper-bound achieved by RDMA networking. The experiment is performed on two machines connected by a 56 Gbit/sec InfiniBand network.

The highest performance is reached by the exclusive WriteWithImm as no synchronization is required and the request is performed in one round-trip. The produce requests with RDMA atomics, however, can achieve the same performance only for records larger than 32 KiB. The main reason is that the throughput of RDMA atomics is limited and cannot exceed 2.68M requests/sec for a single counter on our hardware. RDMA FAA performs better than RDMA CAS as it always succeeds to update the atomic value. Based on the results, in KafkaDirect we use RDMA FAA for shared produce accesses.

The choice of notification method. KafkaDirect relies on the immediate data capability to notify the broker about newly written records. The limitation of this approach is that the producer must be able to encode all metadata related to the request into 32 bits. Another approach is to notify the broker using an RDMA Send request. In this case, the data is written to a TP file without notification using an RDMA Write, and then the metadata is sent separately in a Send request. We will refer to this approach as *Write+Send*.

The main disadvantage of the Write+Send approach is that the producer needs to issue two requests to perform a single RDMA produce: an RDMA Write to a TP file and an RDMA Send that contains metadata. Since Infiniband guarantees in-order packet processing, the Send request will be processed after the preceding Write request preventing the broker to observe partial records.

We evaluate the latency and bandwidth of the Write+Send approach against the Write-WithImm approach using a microbenchmark written in C/C++, which unveils the best performance achievable by the notification approaches. We evaluate Send sizes starting from 4 bytes and up to 512 bytes.

Figure 4.7 shows that the latencies of the studied notification approaches are approximately the same for Writes larger than 1 KiB. For small messages, however, the latency for the WriteWithImm approach can be as little as 1.5 μs, whereas Write+Send approaches are by 1 μs slower on average.

In the bandwidth experiment, we measure the goodput of Write requests only. All approaches reach the goodput of approximately 2.4 GiB/sec for small messages of up to 512 bytes. For 1 KiB messages, WriteWithImm outperforms all Write+Send approaches by at least 0.8 GiB/sec. This difference in bandwidth gradually decreases with the increase in data size and becomes insignificant for 32 KiB records.

Overall, we believe that Kafka could exploit the Write+Send approach to transmit more

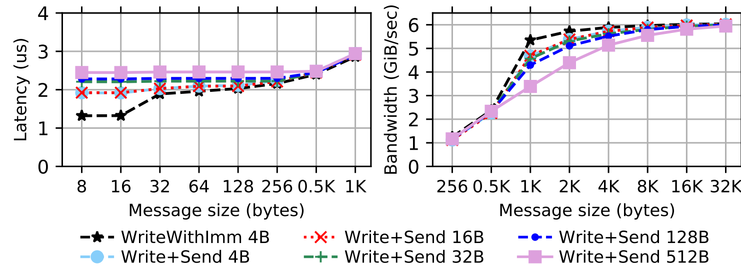


Figure 4.7: Latency and write bandwidth of different approaches for notifying the broker.

metadata to brokers, as it still ensures low latency communication and outperforms classical TCP/IP networking in the terms of bandwidth and latency. In our KafkaDirect we only implemented the WriteWithImm approach for notification as it is the lowest-latency approach.

4.3.3 Replication datapaths

TCP pull replication

Each Kafka broker has a replication module with dedicated worker threads that are responsible for keeping local TP copies in-sync with the leader. The workers periodically send *fetch* requests to the TP leader at their own pace. In response to the request, the leader sends the records which the follower replica does not have or an empty response if the follower is in-sync. The replication module on the follower receives the reply and if it contains new records it appends ❻ them to the corresponding replica TP. Such an approach is commonly called a *pull* approach.

RDMA push replication

We implement a *push* replication module that uses RDMA to replicate the records. The high-level idea is that the leader uses WriteWithImm, similar to an RDMA producer, to write new records to the corresponding TPs of all its followers without incurring extra data copies. When an API worker completes the processing of a produce request, it submits ❼ a replication request to the push replication module that immediately starts writing records to followers, thereby reducing replication latency. KafkaDirect uses the exclusive RDMA WriteWithImm approach as the leader has exclusive access to the followers.

Figure 4.8: Latency and bandwidth of batching 64-byte RDMA Writes. Note log scale of Y axis for the latency plot.

Batching of RDMA Writes. The shortcoming of RDMA push replication is that replication gets triggered for each new record. As a result, a flood of small records could potentially exhaust CPU and RNIC resources if no batching is enabled. To address this problem, KafkaDirect tries opportunistically to batch contiguous RDMA Writes into a single Write. It helps to increase the bandwidth of replication when producers append small records to the same TP.

We ran a microbenchmark to find an optimal batch size for replication requests. The microbenchmark has been implemented in C/C++ to find the best batch size for replication in the case of an overloaded leader. The test emulates the case when the leader receives small entries at a higher rate than it can replicate them. Particularly, the leader writes 64-byte individual records to a local file at 6 GiB/sec and tries to replicate the entries at the same rate, however, the entries have not been batched by a producer.

Figure 4.8 shows the effect of batching on the latency and goodput of replication with an increasing batch size in bytes. The latency of replication with no batching is approximately 2.4 μs, which is the lowest latency value as the data is sent immediately. However, replication of small objects achieves only 0.5 GiB/sec, which is only 8% of the maximum bandwidth. As the batch size increases, the goodput gradually grows until it reaches the link bandwidth of 6 GiB/sec. In contrast, the latency stays approximately the same for smaller batch sizes and then sharply increases for batches larger than 1 KiB. This is because the packet size in our network is 2 KiB, and the write requests become bottlenecked by the bandwidth of the link. As a result, current write requests get delayed by the preceding write requests.

Based on the result, in further experiments, we configure our system with batching enabled and a maximum batch size of 1 KiB, as its goodput is by an order of magnitude higher than the baseline with no batching, and that batch size does not significantly compromise the latency.

4.3.4 Consume datapaths

TCP consume datapath

A Kafka consumer periodically sends a *fetch* request to the brokers to poll new records, similar to followers. A fetch request contains the list of TPs and corresponding Kafka offsets from which to fetch records. The broker sends requested records to the client or it replies with *an empty reply* if no new records are available. Fetch requests incur a significant CPU overhead when processing requests, as brokers can serve thousands of consumers and each consumer periodically sends fetch requests regardless of whether the broker has new records.

RDMA consume datapath

The main goal of our RDMA consume design is to offload the processing of fetch requests to RNICs by exploiting one-sided RDMA Reads. However, this seemingly easy step entails subtle technical challenges. The first challenge is to prevent clients from reading not fully replicated records that would violate Kafka's consistency model. The second one is to allow clients to learn about new records without the involvement of the CPU of the broker. The last challenge is to avoid reading partial records even in the presence of variable-length records. We further describe techniques that are employed by KafkaDirect to address the aforementioned challenges.

Getting RDMA access. To start using RDMA Reads to fetch records, a consumer sends via TCP a request containing a target TP and starting offset from which it wants to read records. The broker registers the requested TP file for RDMA access and replies with information about the file including its virtual address, its *last readable byte*, and whether it is *mutable*. An RDMA consumer never reads beyond the last readable byte, which indicates the position after the last fully replicated record of the requested file, thereby preventing reading uncommitted records.

In Kafka, a file is immutable if data cannot be appended to it. Therefore, the head file of a TP is mutable and other files are immutable (see Figure 4.1). When the consumer receives the information about an immutable file, it periodically initiates RDMA Reads until it reaches the end of the file. After the file is fully read, the consumer gets access to the next file of the requested TP. Therefore, the RDMA consumer only needs to request

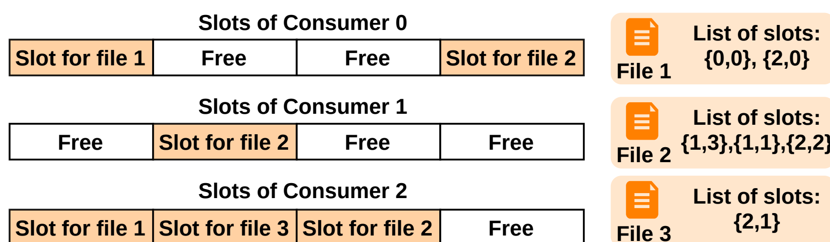


Figure 4.9: RDMA readable metadata slots for mutable files. KafkaDirect creates a contiguous region of slots for each consumer. Each registered file has a list of slots associated with it, so the slots may be updated as the file grows.

RDMA access after a whole immutable file is read. Note that RDMA registration involves mapping the file to the main memory of the broker, so an RDMA consumer also notifies the broker about the files that can be unregistered from RDMA access to reduce memory usage.

RDMA metadata slots. If the requested file is mutable, its last readable byte can be incremented over time. Therefore, the consumer needs to periodically update that information. We propose to exploit RDMA Reads to update that information. When a mutable file is registered for RDMA Reads, the broker creates ❽ an RDMA-readable *metadata memory slot* associated with the file, which contains the last readable byte of the file and whether the file is still mutable. The mutable bit value allows consumers to recognize that the file has become immutable and that they need to request access to the new head file.

An RDMA consumer can read a metadata slot for each subscribed TP to get informed whether new records have been appended. Since a consumer can be subscribed to several TPs, a naive reading of a single metadata slot for each TP could waste CPU and RNIC resources. Thus, for each RDMA consumer, KafkaDirect brokers allocate a contiguous RDMA-accessible region that is used for storing metadata slots of all mutable files requested by the consumer (see Figure 4.9). As the metadata region is contiguous, a consumer only needs a single RDMA Read to update the metadata for all files from which it is actively reading.

The consumer uses a single RDMA Read request to fetch metadata even if some of the slots are unassigned (i.e., free). The consumer always reads the smallest contiguous region containing all active slots (i.e., all not free slots). For example, the consumer 0 from Figure 4.9 needs to read all four slots (including two free slots) to update its metadata,

whereas consumers 1 and 2 can read only their active slots. The broker also tries to keep assigned slots in close proximity to each other to reduce the size of this contiguous region.

When an active file is read by multiple consumers, its metadata will be present in multiple metadata slots as depicted in Figure 4.9. Each RDMA-readable file has a list of metadata slots assigned to it. When the mutability or the last readable byte of the file is changed, the broker updates all the metadata slots associated with it.

Fetch size for RDMA reads An RDMA consumer only knows how many bytes it can fetch from a current TP file, but it is not aware of how many records it contains and what their sizes are. Thus, the fetched bytes may not exactly start and end at record borders and can include bytes of the succeeding record. To address this issue, the RDMA consumer API only returns fully read records, and the partially read records are kept until all their bytes are fetched with RDMA Reads.

The fetch size is a configurable parameter of a KafkaDirect consumer. The default fetch size is 2 KiB as it provides a good trade-off between latency (less than 3 μs) and bandwidth (more than 5 GiB/sec) for RDMA Reads. Even though the current implementation fetches a constant number of bytes using RDMA, it is possible to tune this parameter dynamically during execution. One approach is to estimate the expected size of a record and tune the fetch size accordingly. Alternatively, if the header of a partial record is fully fetched, it is possible to read the size of the record and tune the fetch size accordingly. This alternative approach is helpful when large data entries are stored in the TP.

4.4 Evaluation

We evaluate the performance of KafkaDirect using a series of benchmarks to thoroughly assess the effect of our zero-copy design. For that, we extended the standard Kafka [36], OpenMessaging [125], and the event processing [159] benchmarks to support our RDMA API and to make measurements with microsecond precision. To evaluate the overall impact of our RDMA datapaths, we evaluate the performance of each KafkaDirect's RDMA module in isolation using Kafka and OpenMessaging benchmarks [36, 125]. First, we configure KafkaDirect to enable RDMA only in the produce datapath and study the performance of exclusive and shared RDMA produce protocols (Section 4.4.1). Second, we deploy KafkaDirect in distributed mode and measure the latency and bandwidth of the RDMA replication module (Section 4.4.2). Then, we study the performance of the RDMA

consume module for fetching new data records and checking the availability of new records (Section 4.4.3). Lastly, we show results for event processing benchmark [159] to show how KafkaDirect improves the performance of data processing frameworks such as Spark [171]. Note that Kafka is not meant to handle large messages and the record size is limited to 1 MiB.

Implementation. Our implementation of KafkaDirect is based on Kafka 2.2.1. We use DiSNI [143], a low-latency RDMA library allowing applications to access RDMA networking hardware directly from within the Java Virtual Machine. The DiSNI API is a wrapper over the RDMA C-language user libraries [1] accessed through Java Native Interface calls. For this work, we also extended the DiSNI library to support RDMA atomic requests.

We compare the performance of KafkaDirect with the original Apache Kafka 2.2.1, and an RDMA-enabled Kafka [107] proposed by the Ohio State University. We refer to them as **Kafka** and **OSU Kafka** in our experiments. OSU Kafka does not use one-sided RDMA requests to access records and only uses RDMA Sends that entail copying requests from and to network buffers, resulting in loss of performance. In the experiments, **Kafka** represents the performance of the unmodified Kafka over high-bandwidth RDMA-capable networks, and **OSU Kafka** represents the Kafka that uses two-sided RDMA networking only for request messaging, and **KafkaDirect** represents our design with full offload of data accesses using one-sided RDMA networking.

Experimental setup. The experiments are conducted on a 12-node InfiniBand cluster, where each machine is equipped with a 56 Gbit/sec Mellanox ConnectX-4 network card. Each machine has two 8-core Intel Xeon CPU E5-2630 v3 CPUs and 256 GiB of DDR4 DRAM.

In all experiments, the unmodified Kafka was deployed over the same network to have a fair comparison with RDMA-enabled systems. All Kafka-based systems are deployed with 1 GiB log files and enabled file preallocation, i.e., Kafka immediately allocates storage for each created 1 GiB file. Unless otherwise specified, Kafka-based systems were deployed with default parameters that include eight API worker threads and three network workers.

To be completely oblivious of the performance of storage device used for storing log and TP files, Kafka's files are created in *tmpfs* [140], which is backed by DRAM. Otherwise, the performance of Kafka would be bottlenecked by the speed of the storage device. By making this change we do not compromise the reliability guarantees of Kafka, as Kafka's failure

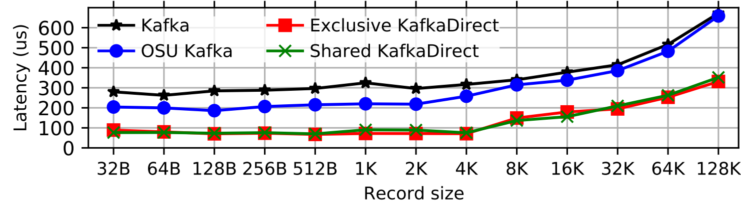


Figure 4.10: Latency of produce request when replication is disabled. Producers do not batch requests.

tolerance only relies on replication and is independent of the availability of persistent storage. The bottleneck of the persistent storage media can be alternatively removed by several techniques including the use of faster NVMe devices (e.g., AORUS Gen4 AIC that achieves 110 Gbit/sec read and 110 Gbit/sec write bandwidth [155]) or the use of multiple storage media at each broker. We leave this exploration for future work.

4.4.1 The effect of RDMA on produce datapath

Latency. We measure the median latency of produce requests. The latency is a round-trip time measured by a produce client: it sends a single produce request and waits for an acknowledgment from the broker. The topic was created with a single partition and with no replication enabled.

Figure 4.10 shows that OSU Kafka reduces the latency of the original Kafka by about 90 μs for small sizes, however, for 128 KiB records, OSU Kafka has the same latency as the original Kafka. The lowest latency is observed for KafkaDirect clients: 90 μs for small messages and approximately 345 μs for large 128 KiB messages. Overall, KafkaDirect provides 3.3x and 2x improvement over Kafka and OSU Kafka, respectively.

The latency of an exclusive KafkaDirect producer is 2.5 μs lower than a producer with the shared TCP/RDMA approach. The difference comes from the requirement of the shared approach to issue an RDMA FAA operation. Interestingly, the latency of an RDMA produce request is not as low as the latency of an RDMA Write request, which is approximately 2.5 μs. The overhead of 88 μs comes from two Kafka's design decisions: the producer API makes a copy of user data to prevent mutation of it during transmission; and Kafka has different threads for API and network workers, incurring inter-thread communication (forwarding a request takes 11 us). A processing of a small record takes on average 14 us for an API thread, including CRC32C checksum calculation. The rest of the

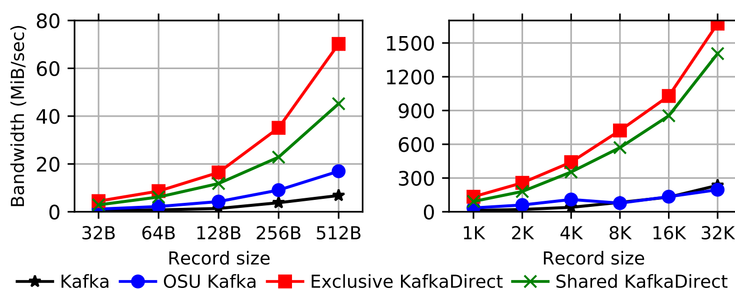


Figure 4.11: Bandwidth of produce request to one partition. Replication is disabled. Producers do not batch requests.

overhead comes from the thread invocations due to blocking polling of the RNIC events, the network, and producer's API.

The experiment shows that the zero-copy produce datapath of KafkaDirect significantly outperforms the Send/Recv approach used by Kafka and OSU Kafka in terms of latency.

Bandwidth. In this experiment, we measure the goodput of produce requests. The producer dispatches as many requests as possible to a single TP. The TP is not replicated to show the performance of the produce datapath only.

Figure 4.11 shows that KafkaDirect achieves the highest performance, whereas the lowest performance is observed for the original Kafka. The low performance comes from extra data copies induced by the TCP/IP stack and copies from network buffers to TP file buffers. OSU Kafka removes some of these copies and, on the experiment with 512-byte records, achieves a 2x improvement. In the same experiment KafkaDirect shows a 10x speedup for the exclusive produce datapath and a 5x improvement for the shared produce datapath. On average in all experiments, an exclusive RDMA producer achieved a 7x speedup compared to Kafka and 3.8x compared to OSU Kafka. In the experiment, the RDMA producer could achieve 1.65 GiB/sec with 32 KiB records, whereas the original Kafka achieved only 280 MiB/sec. A shared producer also has a significantly improved bandwidth with large records in the produce datapath and shows a 5x improvement compared to Kafka.

Figure 4.12 shows the effect of partitioning on the bandwidth of producers. The bandwidth of all systems increases with the number of partitions as each TP file can be accessed by at most one API worker at a time due to locking. Thus, four partitions can be concurrently written by four workers, thereby improving overall bandwidth. The performance saturates at 8 partitions which is the number of API workers in Kafka. KafkaDirect achieves 4.5

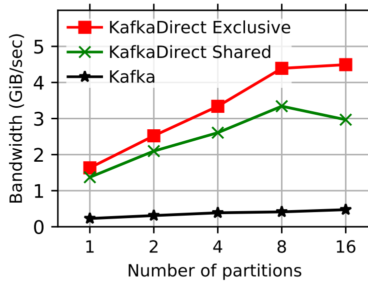


Figure 4.12: Bandwidth of produce requests for 32 KiB records.

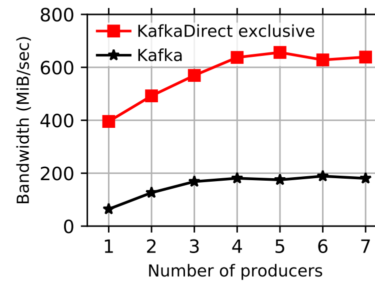


Figure 4.13: Total bandwidth of producers for 4 KiB records. Broker is deployed with one worker.

GiB/sec for the exclusive RDMA datapath and 3 GiB/sec for the shared RDMA datapath, which is a 9x and 4.5x improvement over Kafka, respectively.

The experiments show that the produce datapaths of Kafka and OSU Kafka are bottlenecked by extra data copies, and that our RDMA extension removes this bottleneck.

Bandwidth of a single API worker. To evaluate the maximum bandwidth that can be achieved by a single API worker, we deployed systems with one API worker. To plot the bandwidth curve, we vary the worker's load by increasing the number of producers. Each producer writes 4 KiB records to its private TP, to eliminate contention between producers (so the number of producers is equal to the number of topics). The main goal of this configuration is to remove contention between threads at polling the request queue (see Figure 4.2).

Figure 4.13 reveals that the performance of the KafkaDirect broker plateaus at 630 MiB/sec when the system processes the records from more than four clients. For the original Kafka, the top performance is only 190 MiB/sec. Thus, to achieve the line rate of 6 GiB/sec KafkaDirect should be deployed with at least 10 API workers, whereas for the original Kafka more than 33 workers are required. *We conclude that KafkaDirect provides a 3.3x reduction in CPU load.*

4.4.2 The effect of RDMA on replication

Latency. We measure the latency of produce requests when the system is deployed with replication enabled. The latency is a round-trip time measured by a producer that waits for an acknowledgment. The acknowledgment is received when the data is fully replicated to all replicas. We measure latency when 1) RDMA is enabled only for produce datapath,

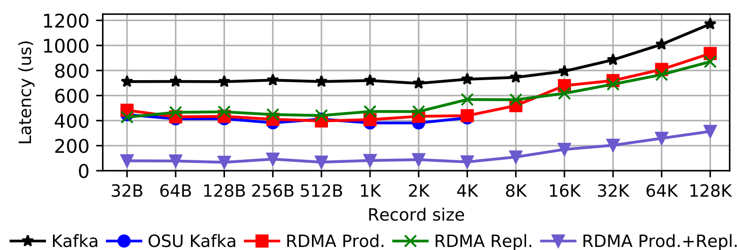


Figure 4.14: Latency of producer for 3-way replication.

2) RDMA is enabled only for replication datapath, and 3) RDMA is enabled for both datapaths.

According to Figure 4.14, the latency of Kafka with three-way replication is approximately 700 μs for small records, which is twice that of a produce request with no replication. Enabling either RDMA modules of KafkaDirect reduces the latency by 300 μs. Interestingly, when both modules are enabled the latency decreases to about 100 μs. OSU Kafka only reduces the latency by 300 μs, similar to KafkaDirect when either one of the two RDMA modules is in use. Overall, KafkaDirect provides a 7x improvement over Kafka and a 4x improvement over OSU Kafka for three-way replication.

Our RDMA replication module has the lowest latency since the leader broker starts replication immediately, rather than waiting for replicas to pull the data.

Bandwidth. We measure the average goodput of produce requests when the topic is three-way replicated (the leader replicates data to two other machines). We were not able to measure all data points for OSU Kafka as it was crashing for experiments with large records.

Figure 4.15 shows that the highest performance is observed for KafkaDirect, which achieves a 14x speedup for 32 KiB records compared to Kafka. Interestingly, just enabling RDMA replication does not contribute much to the total bandwidth since the performance is bottlenecked by the slow TCP producer. The RDMA producer can achieve more than 500 MiB/sec, which is 420 MiB/sec faster than the original Kafka.

The data unveils that the performance of the RDMA producer is limited by the speed of the pull replication. Our RDMA replication module manages to mitigate the bottleneck and to double the performance. Overall, the speedup of KafkaDirect is from 9x to 14x depending on the size of the records.

To understand the role of the replication factor on performance, we measure the bandwidth

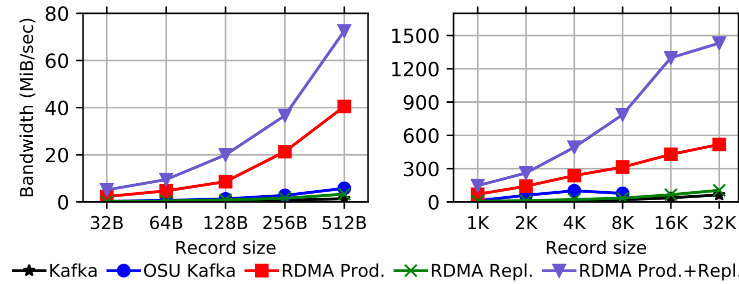


Figure 4.15: Bandwidth of producer for three-way replication.

of a producer with increasing replication factor. Figure 4.16 shows the bandwidth for 32 KiB records when data is replicated from one to four times. Note that a replication factor of one means that data is stored only on the leader. An RDMA producer achieves 1.5 GiB/sec when the replication is disabled. However, when records are replicated using TCP, the performance drops to 0.5 GiB/sec. In contrast, our RDMA replication module replicates data at the required rate and avoids this one-third slowdown. For two replicas, KafkaDirect provides a 14x speedup compared to the original Kafka.

The main observation is that the increase in the number of replicas does not significantly reduce the overall performance of all tested systems. This due to the original Kafka optimization [113] that enables transferring content of mapped files over the TCP/IP without incurring extra copies. At the receiver side, however, each follower still performs two memory copies: from the driver's receive buffer to one of Kafka's receive buffers, and from the receive buffer to the file buffer. These two copies are avoided by our KafkaDirect design, thereby reducing CPU utilization on the replica brokers.

Batching of replication requests. KafkaDirect supports batching of consecutive contiguous writes into a single RDMA operation during replication (Section 4.3.3). The goal of batching is to increase the bandwidth of replication when producers dispatch many small produce requests. To evaluate the effect of batching on replication performance we measure the bandwidth of produce requests when KafkaDirect is deployed with RDMA replication enabled, and the RDMA producer injects 32-byte records that are not batched.

Figure 4.17 shows the average bandwidth of produce requests for two- and three-way replication with increasing maximum batch size of the RDMA replication module. No batching achieves a bandwidth of 3.8 MiB/sec for both two- and three-way replication. The bandwidth increases with increasing batch size and plateaus at 5.2 MiB/sec. In general, the speed of RDMA Write can be as high as 200 MiB/sec for such small writes,

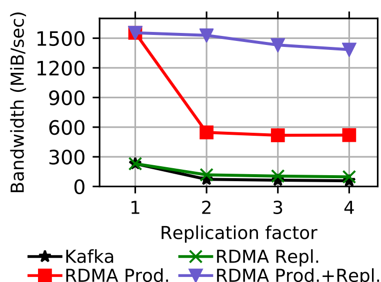


Figure 4.16: Bandwidth of produce request for 32 KiB records.

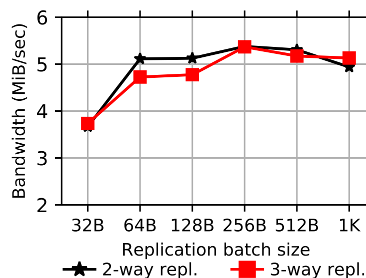


Figure 4.17: Bandwidth of 32-byte produce requests with increasing batch size.

however, the performance was bottlenecked by the speed of the API worker which commits new records to TP files: the workers need to calculate checksum over the new records and acquire an exclusive write lock. Therefore, the replication module was replicating records at 5.2 MiB/sec, thereby underutilizing the network. To improve network utilization one can delay the replication requests by a constant timeout to batch more requests, which would, however, incur higher replication latency.

We did not observe the performance numbers achieved in our C/C++ benchmark (Section 4.3.3). Nonetheless, we believe that our batching mechanism can be still beneficial for other publish-subscribe systems, especially, for ones without integrity checks, since our batching is opportunistic meaning that the replication worker does not wait for requests to accumulate, and can dispatch a batch of a smaller size than the maximum batch size.

4.4.3 The effect of RDMA on consume datapath

Latency. We measure the round-trip time measured by a consumer. We load Kafka-based systems with 10,000 records to a single partition and the consumer fetches them one by one. Note that the client latency is independent of produce and replication latency in this experiment, as all records are preloaded. All systems are deployed with a default file size of 1 GiB. We could not measure the latency of fetch requests for OSU Kafka, as we did not have access to the source code to instrument the consumer API. Instead, we measure its end-to-end latency in the next experiment.

Figure 4.18 shows that the latency of the original Kafka is at least 200 μs for all tested record sizes. The latency is composed of a TCP/IP round trip and the processing of the fetch request. Our RDMA consumer fetches a record within 4.2 μs, which is a 50x improvement over the original Kafka. The latency is so low because the data was preloaded

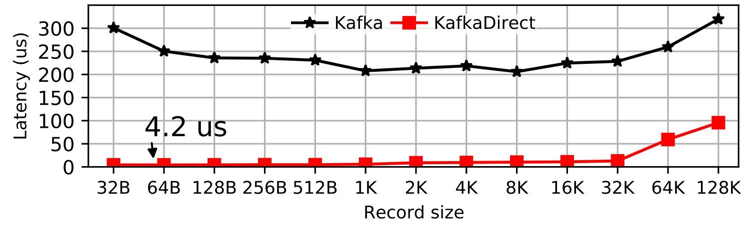


Figure 4.18: Latency of consumer with increasing record size.

to the broker and the RDMA consumer read all the remote records with RDMA without requesting access to TP files. In general, if an RDMA consumer reads entries from a TP consisting of many files, it needs to request RDMA access after reading each file.

The latency of an RDMA fetch request is 4.2 μs, which is 2 μs greater than the latency of a pure RDMA Read request. The overhead comes from Kafka's consumer API design which requires returning a native Java buffer (i.e., allocated in Java's heap) to the user. The DISNi RDMA Java library only works with off-heap buffers, therefore our implementation always needs to copy the fetched records to a native Java buffer. One possible solution to this problem is to extend the Kafka API to allow users to provide an off-heap buffer where the records can be fetched without extra copies.

Latency of empty fetch requests. We evaluate the cost of checking the availability of new records in a TP. For Kafka, it is the latency of a fetch request in the case when the broker does not have new records. For KafkaDirect, it is the latency of reading a remote metadata slot using RDMA Read (Section 4.3.4). The experiment reveals that the latency of an empty TCP fetch request is at least 200 μs, whereas the latency of reading a remote metadata slot is only 2.5 μs. What is more, the RDMA fetch metadata request does not involve the broker's CPU and is completely offloaded to the RNIC. *As a result, our KafkaDirect can serve thousands of RDMA fetch requests without any CPU involvement.*

End-to-end Latency. The previous experiment measures the latency of consumers when they fetch data from immutable files. Therefore, each RDMA consumer does not need to frequently update the metadata of TP files to discover new records. In this experiment, we measure an end-to-end latency where a single client application plays the role of producer and consumer. The client sends a single record to Kafka and then fetches it with the consumer API. Thus, the measured round-trip latency consists of both produce and fetch requests. Since KafkaDirect supports enabling only particular RDMA modules we measure latency for when 1) RDMA is enabled only for the produce datapath, 2) RDMA is enabled only for the consume datapath, and 3) RDMA is enabled for both datapaths.

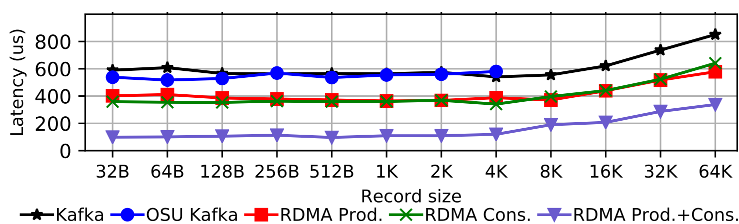


Figure 4.19: End-to-end latency with increasing record sizes.

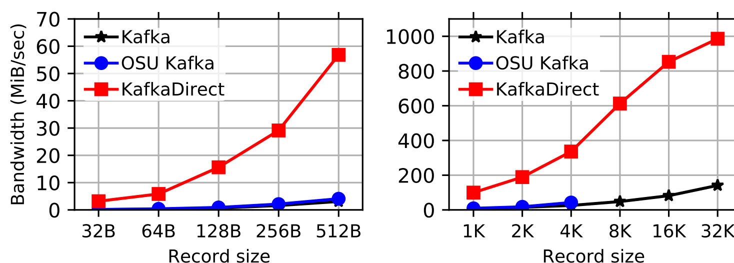


Figure 4.20: Consume bandwidth with increasing record size.

Figure 4.19 shows the median end-to-end latency of Kafka is about 600 μs for small records. OSU Kafka gives approximately the same latency as the original Kafka, but at some data points, we observe a 50 μs reduction in latency. The use of RDMA for either the produce or consume datapath reduces the latency by at least 200 μs. When both RDMA modules are enabled the latency is as low as 100 μs. Interestingly, since the latency of RDMA produce is about 93 μs, the actual latency of RDMA fetch was about 7 μs which consists of data fetching (4.2 μs) and metadata update (2.8 μs). *We conclude that KafkaDirect can offer a 5.8x reduction in end-to-end latency, and that our RDMA consumer can efficiently work with frequently updated TPs.*

Bandwidth. We measure the average goodput of a consumer for the systems that were loaded by a producer to one partition. In addition, to avoid the effect of batching, the broker was configured to reply with one record for each fetch request. We were not able to measure all data points for OSU Kafka as it was crashing for some experiments.

Figure 4.20 shows that the highest throughput was observed for our RDMA consumer. OSU Kafka and the original Kafka have approximately the same performance, which is less than 150 MiB/sec even for large records. The RDMA consumer, on the other hand, shows a 9x improvement over the original Kafka and managed to achieve 1 GiB/sec bandwidth.

It is worth noting that the performance of our RDMA consumer is bottlenecked by the consumer's implementation, whereas in the original Kafka it is limited by the broker. It

comes from the fact that the RDMA fetch request is completely offloaded to the RNIC and does not require the involvement of brokers' CPU. As a result, the number of RDMA consumers is only limited by the capabilities of the RNIC, allowing KafkaDirect brokers to serve thousands of consumer clients without incurring any CPU overhead.

The maximum RDMA Read bandwidth achievable by our RNIC is about 6 GiB/sec, however, KafkaDirect only achieved 5.2 GiB/sec even for large records (the experiment is not plotted here). The reduction in bandwidth comes from the fact that the RDMA consumer must check the integrity of the fetched data and copy the data from the internal off-heap buffers used for RDMA into Java native buffers, that are returned to the caller.

Throughput of empty fetch requests. The main shortcoming of Kafka's consume datapath is that consumers periodically send fetch requests regardless of the availability of new records. As a result, brokers spend a lot of CPU cycles on processing fetch requests and sending empty replies to clients. We call such requests as *empty fetch requests*. We deployed Kafka with default parameters and measured how many empty fetch requests it can process. The experiment showed that a broker could not process more than 53K empty fetch requests per second and the performance was bottlenecked by the TCP network module. A broker of KafkaDirect managed to process 8,300K empty fetch requests per second providing a 156x improvement over the original Kafka. The speedup comes from the fact that RDMA consumers use one-sided RDMA Reads to find out about newly available records by reading remote metadata slots (Section 4.3.4). Note that the processing of empty fetch requests in KafkaDirect does not involve the CPU of brokers and is bottlenecked by the RNIC speed.

4.4.4 Improving Data Processing Applications.

We integrated KafkaDirect into Apache Spark 2.4.4 [171] and measured its performance for the streaming benchmark [159]. The benchmark emulates events generated by an IoT traffic sensor that measures the number of cars and their average speed for road lanes. The IoT device publishes these events in JSON format into two separate topics, that are polled by event processing engines. To be oblivious from the processing speed of streaming engines we report the delay between the timestamp when the sensor generated an event and the time when the processing engine read the event.

Figure 4.21 reports the measured delays for two workloads: constant-rate and periodic-burst. The first workload has a constant publishing rate, whereas, in the periodic burst

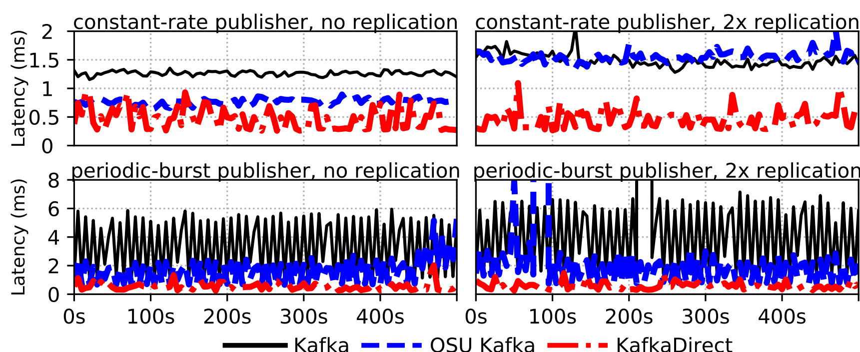


Figure 4.21: Event delays under constant-rate and periodic-burst workloads for no and 2x replication settings.

one, every ten seconds an enlarged batch is published. The plot shows that the lowest delays were achieved by KafkaDirect for all workloads, especially for the setting with replicated topics. For the constant-rate workload, KafkaDirect has higher variance than competitors as the fetching process was affected by *commit offset requests*. The commit offset request helps consumers acknowledge the reception of records to avoid processing of the same records twice in the case of node failures. Since KafkaDirect does not use RDMA for that request, its performance was decreased by the use of the TCP/IP stack. KafkaDirect could implement an accelerated *commit offset requests* with the use of RDMA FAA, which is an interesting direction for future research.

Despite that limitation of KafkaDirect, it had much lower variations in latency for the periodic-burst workload. What is more, Kafka and OSU Kafka experienced a short period of unavailability for the replicated settings. *The experiment shows that KafkaDirect performs well in the case of bursty data, and provides a 3.3x latency reduction on average.*

4.5 Related work

Publish-subscribe systems. Corfu [14] and Scalog [44] are shared log systems that maintain total order across records stored on different servers. Unlike Kafka, Corfu and Scalog have a single logical TP that is partitioned across servers. To publish records, a Corfu's client determines the available position in the shared log (similar to Kafka offsets) using a dedicated sequencer node, and then writes data to that position. Corfu clients are also responsible for data replication. Unlike Corfu, a client of Scalog appends records to any server, which then will replicate the records. Periodically, each storage server of

Scalog talks to the sequencer node that assigns a unique position to all fully replicated records. Scalog's sequencer algorithm ensures global ordering across all stored records. Fuzzylog [94] is a partially ordered log that tracks order between records stored in geo-replicated shards using Skeen's algorithm [55]. We believe that the mentioned systems could reuse our RDMA datapaths with slight modifications since they store *immutable* records with a *log-structured* design as Kafka. In particular, sequencer nodes besides logical positions of records could also return their virtual addresses to enable RDMA accesses.

RDMA-enabled log-structured systems. HERD [70], FlatStore [34], and RamCloud [111] are log-structured key-value stores that use index structure as a level of indirection between keys and storage location. Since the traversal of the index may result in multiple RDMA operations, they only use RDMA-based RPCs for request processing and do not expose direct object access to clients. HERD also optimizes RPCs to deliver requests to buffers in the proximity of the expected storage location. HyperLoop [80] is a framework that offloads chain replication to RNICs with cross-channel communications support [152]. HyperLoop improves the replication performance of write-ahead transactional logs and can be employed by log-structured systems. DaRE [122] is a replicated state-machine that implements a replication protocol using one-sided RDMA Writes to write data directly to remote logs. Additionally, DaRE employs RDMA Reads to verify the execution progress of replication followers.

4.6 Summary and Discussion

This chapter explores challenges and solutions for the efficient acceleration of Apache Kafka with zero-copy RDMA networking. Our implementation, KafkaDirect, employs RDMA Writes and the immediate data capability to write data directly to storage and, at the same time, to notify the broker, thereby avoiding the need for extra messages. KafkaDirect also makes use of RDMA atomics to enable shared write access to a single file. KafkaDirect empowers consumers to use RDMA Reads to fetch records directly from Kafka with no CPU involvement on the broker. We demonstrate the effectiveness of these techniques in multiple settings. Our evaluation shows that RDMA can significantly improve the performance of publish-subscribe systems and enable scaling to a larger number of clients.

5

Serialization-free RDMA networking in Java

Managed languages such as Java and Scala do not allow developers to directly access heap objects. As a result, to send on-heap data over the network, it has to be explicitly converted to byte streams before sending and converted back to objects after receiving, thereby preventing zero-copy RDMA networking for on-heap objects. The technique, also known as object serialization/deserialization, is an expensive procedure limiting the performance of Java-based distributed systems as it induces additional memory copies and requires data transformation resulting in high CPU and memory bandwidth consumption. With the widespread use of Java in large scale data processing and the increased availability of RDMA, it is time to rethink current object serialization/deserialization techniques to take full advantage of offloaded networking.

The work in this chapter explores the following research questions:

- What are the main overheads in the existing serialization techniques for object transfers?

- How can we send objects across heaps without superfluous memory copies and data transformation induced by serialization?
- What applications can benefit from the serialization-free networking?

To address these questions, we developed Naos, a library that allows objects to be sent serialization-free from a local Java virtual machine (JVM) to a remote one with minimal CPU involvement and over RDMA networks. As Naos eliminates the need to copy and transform objects, it offers significant speedups compared to state-of-the-art serialization libraries. Naos exposes a simple high level API hiding the complexity of the RDMA protocol that transparently allows Java-based systems to take advantage of offloaded RDMA networking.

The content of this chapter has been published at the USENIX Annual Technical Conference (USENIX ATC) in 2021 [149]. The work in this chapter was done in collaboration with Rodrigo Bruno and our advisors Torsten Hoefler and Gustavo Alonso.

5.1 Motivation

Managed programming languages, such as Java and Scala, are a common vehicle for developing distributed platforms such as Spark [171], Flink [29], or Zookeeper [60]. However, the high level abstractions available in managed languages often cause significant performance overheads. In particular, to exchange data over the network, Java applications are currently forced to transform structured data via serialization, causing a high CPU overhead and requiring copying the data multiple times. While less of an issue in single-node applications, the overhead is substantial in distributed settings, especially in big data applications. Serialization already accounts for 6% of total CPU cycles at Google datacenters [73].

Data transfer with object serialization/deserialization (OSD) is a complex process involving five steps: **graph traversal** to identify all objects that should be serialized; **data transformation** to convert the objects into a byte stream (network-friendly format); **transmission** to send the serialized data over the network; **data traversal** at the receiver to decode the received data; and **object construction** that involves allocating memory and object re-initialization.

To illustrate the CPU overhead caused by OSD, we benchmarked the Kryo [141] serializer and measured its CPU utilization while sending objects over different networks. Figure 5.1

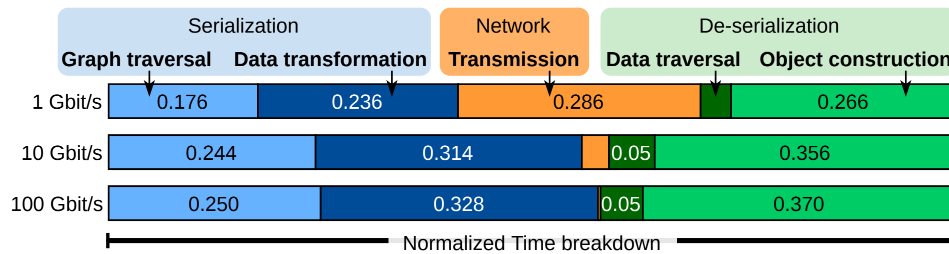


Figure 5.1: Impact of network bandwidth on OSD time.

shows the fraction of time spent on each OSD step for the transfer of an array of 1.28M objects, all of the same exact type. Each object has two fields, each encapsulating a primitive type. Results show that the time spent in OSD increases as networks get faster. For a 10 Gbit/s network, it takes less than 3% of the time to send data over the network, but it takes more than 31%/35% of the time in data transformation/object construction. This discrepancy is even more evident in 100 Gbit/s networks in which the network time drops to less than 0.01% and the time spent on the CPU performing OSD accounts for almost 100% of the transfer time. While networks are getting faster, the pressure is moving away from the network and into the CPU (and memory bandwidth), further aggravating the already well-known CPU-bottleneck problems encountered in distributed applications [112, 158]. Furthermore, existing distributed platforms that heavily rely on OSD are not able to take advantage of faster networks such as RDMA.

With the widespread use of Java in large scale data processing and the increased availability of RDMA, it is time to rethink current OSD techniques so that part of the load of object shifts from the CPU back to the network. In this study, we aim to develop native runtime support for *serialization-free networking* that avoids superfluous memory copies and data transformation by sending objects directly from the source heap into the remote heap. Sending and receiving data without data marshalling enables the use of zero-copy RDMA networking, bypassing not only serialization but the need to copy data. Such a design significantly reduces the pressure on the CPU at the cost of higher data volumes to be transferred, since objects are sent in their uncompressed memory format.

To test and evaluate these ideas, we have developed Naos (Naos stands for Not Another Object Serializer), a library and runtime plugin for OpenJDK 11 HotSpot JVM that allows objects in the source heap to be directly written into a remote heap, avoiding data transformations and excessive data copies. Naos is designed to accelerate object transfers in distributed applications by taking advantage of RDMA communication (although it

also supports conventional TCP sockets).

Naos allows applications to directly send objects without employing serialization libraries. Its API requires no type registration nor serialization snippets, guaranteeing developers a close to zero effort when building systems using Naos. Finally, Naos is the first (to the best of our knowledge) library integrating RDMA into JVM allowing the user to communicate on-heap objects transparently, thereby easing the adoption of RDMA networking by JVM-based distributed applications. Our evaluation shows that Naos provides a 2x throughput speedup over serialization approaches for transferring contiguous objects and for moderately sparse object graphs.

Contributions. Naos is the first *serialization-free* communication library for JVM that allows applications to send objects directly through RDMA or TCP connections. Naos unlocks efficient asynchronous RDMA networking to JVM users hiding all the burden of low-level RDMA programming from the users, thereby facilitating the adoption of RDMA. For that, Naos solves several complex design issues such as sending unmodified memory segments across Java heaps without employing intermediate buffers, and interacting with concurrent garbage collection without compromising JVM's memory safety. For the first issue, Naos proposes a novel algorithm that writes objects directly to the remote heap and makes them valid on the receiver's address space (Section 5.3.3). For the second one, Naos proposes techniques preventing a concurrent JVM garbage collector from moving unsent objects that may be accessed by RNIC and from accessing unrecovered received objects (Section 5.3.2). Finally, Naos enables pipelining communication and serialization, which was previously impossible with the OSD approach (Section 5.3.4).

5.2 Background on Object Serialization

Overview. Many third-party libraries [85, 62, 141] have been developed to perform OSD in Java. Some of them provide Java bindings for popular cross-language OSD approaches (e.g., Protobuf [62]), allowing serializing arbitrary data structures into well-defined messages that can then be exchanged using any network protocol. While remaining independent of programming languages or operating systems, such libraries suffer from low performance [108]. Therefore, JVM-based big-data applications (e.g., Spark, Flink) rely on specialized libraries such as Kryo [141], designed specifically for JVMs.

Figure 5.2 and Figure 5.3 present a serialization example of a Java object and its data

Sender with Kryo

```
1: buffer = ByteBuffer.allocate(512);
2: person = new Person(18, "Mike");
3: kryo = new Kryo();
4: kryo.register(Person.class);
5: out = new Output(buffer);
6: kryo.writeObject(out, person);
7: connection.write(buffer)
```

Receiver with Kryo

```
1: buffer = ByteBuffer.allocate(512);
2: connection.read(buffer);
3: kryo = new Kryo();
4: kryo.register(Person.class);
5: in = new Input(buffer);
6: obj = kryo.readObject(in,Person.class);
7: person = (Person)obj;
```

Sender with Naos

```
1: person = new Person(18,"Mike");
2: connection.writeObject(person);
```

Receiver with Naos

```
1: object = connection.readObject();
2: person = (Person)object;
```

Figure 5.2: Serialization, Deserialization for Kryo and Naos

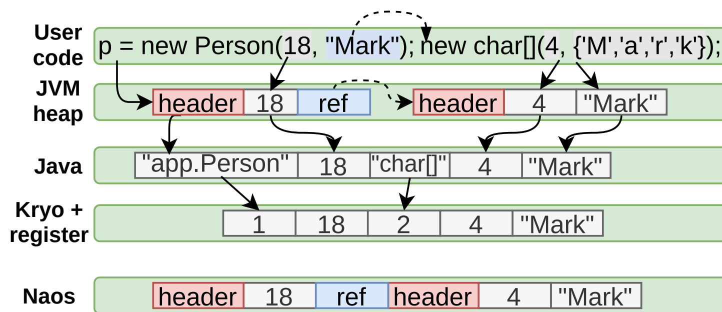


Figure 5.3: Data format for Kryo and Naos

formats: memory layout (JVM heap), and serialization formats (Java, Kryo). All Java objects start with a JVM-specific header (red) followed by a number of primitive (gray) or reference fields (blue). The object of type `Person` has one primitive `int` field followed by a reference field to a character array (`char[]`). The character array starts with the `length` of the array followed by all characters.

Serializing an object involves traversing the object graph starting from that object and, upon visiting each reachable object, copying all primitive fields into the pre-allocated byte buffer. During native Java serialization, headers are replaced by class descriptors in textual format (`app.Person`) and field references are replaced by the contents of the pointed object. Deserialization follows a similar logic; upon visiting a serialized object, a new object must be allocated, and all primitive fields are copied out of the buffer into the allocated object.

Kryo. Kryo [141], one of the most widely used OSD libraries, addresses some limitations of native Java serialization by requiring manual registration of classes to achieve a more compact representation of the serialized data. Figure 5.3 shows a serialized data format

in Kryo with class registration (Kryo+register). Kryo can represent all primitive types and classes using integer identifiers, thereby reducing the amount of space needed for storing type names. Although the class registration is trivial in this example, this task is cumbersome for applications with hundreds of data types. Compared to Kryo, Naos provides a cleaner interface (see Figure 5.2) with no need for developer involvement. To send a Java object, one can directly write it (`writeObject`) to the network. The receiver can directly read the object with `readObject`.

Accelerated OSD. To address the overhead of having to transform the data, Cereal [65] and Optimus Prime [123] resort to dedicated hardware accelerators for OSD. These accelerators are co-designed with the serialization format to parallelize the OSD process. Even though their data formats are not portable across different JVMs, their simulation results promise 15x speedup in serialization throughput on average over Kryo at the expense of requiring specialized hardware.

Zero-transformation OSD. The trade-off portability vs. performance is also exploited by the serialization library Skyway [108]. By dropping portability, Skyway manages to partially avoid data transformations and object construction by serializing Java objects in their JVM formats, i.e., the objects are written to communication buffers in the same binary format they are stored in the heap. Like Skyway, Naos sends objects in the JVM heap format, assuming that communicating parties run on the same JVM software. Unlike Naos, however, Skyway is a *serialization library* requiring to copy objects to and from communication buffers. Naos, on the other hand, completely removes the need to explicitly serialize and deserialize objects to send objects between Java heaps even with RDMA. What is more, Skyway's memory management prevents the use of RDMA networking (Section 5.4).

Naos integration and applicability. *Naos is not a serialization library.* Naos only covers end-to-end transfers (see Table 5.1) and cannot replace OSD in systems that do not use it for communication (e.g., for writing objects to disks). Naos has been primarily designed for future systems that want to take advantage of serialization-free zero-copy RDMA networking.

In several existing Java frameworks the main obstacle to using Naos is that some of these systems do not consider the possibility to send objects without serialization. For example, Spark and Hadoop completely decouple serialization from communication: their serialization modules are designed to serialize objects only to files, and their shuffle modules are designed to communicate only files. Such file-centric design simplifies inter-node com-

munication, as processes can share file descriptors instead of sending data, and helps to reduce memory usage by dumping data to disks. However, it makes integrating Naos very difficult. For such use-cases, conventional OSD libraries are a better fit than Naos if a redesign for true zero-copy is infeasible.

Table 5.1: APIs of Naos RDMA.

API	Description
`void writeObject(Object)`	Blocking send of a single object
`Object readObject()`	Blocking read of an object from heap
`boolean isReadable()`	Check whether an object can be read
`long writeObjectAsync(Object)`	Nonblocking send of a single object
`int waitHandle(long)`	Wait for a send request to complete
`int testHandle(long)`	Tests completion of a send request

5.3 System Overview

Naos allows Java applications to send/receive objects directly through RDMA or TCP connections. Naos uses a collection of algorithms and data structures to efficiently transmit large complex data structures. Figure 5.4 presents a graphical overview of Naos' workflow, including the main algorithms and data structures. An object transfer starts with a `writeObject` ❶ triggering a DFS graph traversal ❷ (Section 5.3.1). During the traversal, pointers to already visited objects are detected using an interval tree. After the traversal, both the objects ❸ and metadata ❹ are sent over the network using RDMA ❺ (Section 5.3.2). Naos uses a circular message buffer to send metadata ❽ and writes objects directly to the remote heap ❻. Upon reception of the data and metadata, the receiver starts recovering (Section 5.3.3) the object graph by fixing class pointers ⓫ and field pointers ⓭. Once pointers are fixed, the head of the object graph is returned ⓮ to the caller of `readObject` ❾.

The `writeObject` call in Naos is blocking, that is, the call returns once the object transmission is completed. It ensures that the object is received by the destination. In contrast to the classical TCP/IP semantics, all RDMA operations are executed asynchronously by design, allowing overlapping computation with communication. Naos also provides a nonblocking `writeObjectAsync` call enabling asynchronous communication for RDMA connections (Section 5.3.2). The nonblocking call initiates the send operation but does

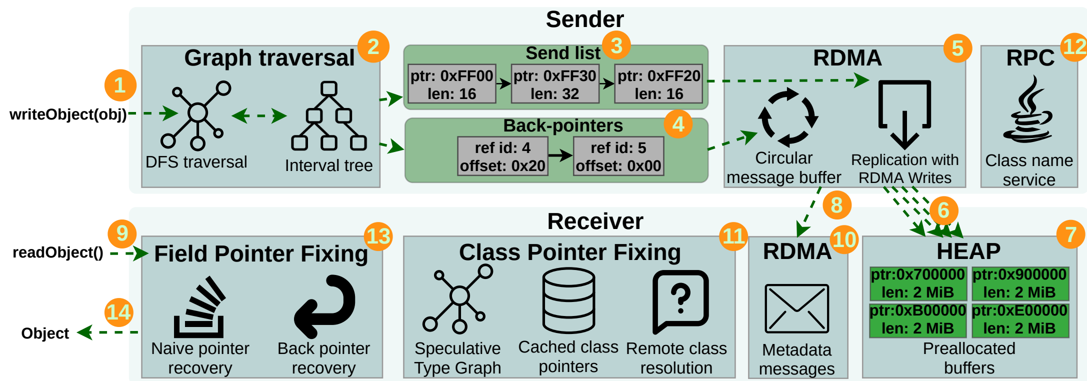


Figure 5.4: Naos' workflow for sending and receiving a Java object.

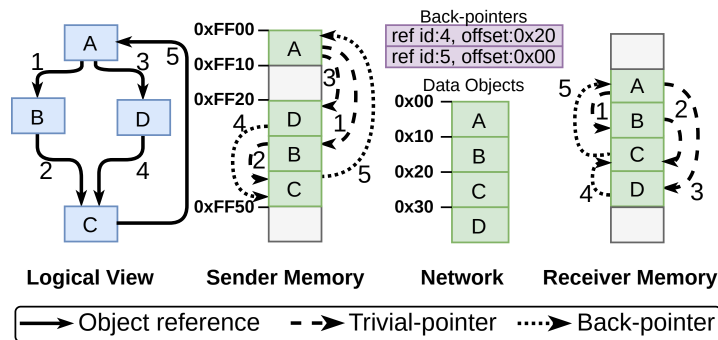


Figure 5.5: Object views of Naos' graph traversal and pointer recovery.

not fully complete it. Instead, it returns a request handle, that is used by a user to wait for the completion using `waitHandle` call or to verify whether the request is completed using `testHandle` call.

5.3.1 Object Graph Traversal

Java objects can contain reference fields pointing to other Java objects and therefore, when an object is passed as an argument to `writeObject` ❶, all objects reachable from it need to be sent. To find all objects reachable from a particular object, Naos traverses the object graph ❷ in Depth-First-Search (DFS) order. Figure 5.5 illustrates a simple example of an object graph's (*Logical View*), sender memory layout, format sent over the network, and receiver memory layout. The *sender memory* starts at address `0xFF00` and all objects occupy 16 bytes. Edges are numbered according to DFS order.

When an object is visited for the first time, it is included in the *Send list* ❸: a list of

memory blocks that will be sent over the network. Each memory block has two elements: the starting virtual address, and the length. The send list contains objects ordered according to DFS order, and the objects are sent in this order over the network. Naos also merges the memory blocks that are adjacent in the send list to reduce its length. For that, during traversal Naos checks whether a new visited memory block is a continuation of the last block of the send list: if yes, then Naos increases the length of the last block, otherwise, Naos adds a new block to the list. The resulting send list is presented in ❸, which contains three elements: for object A, for objects B and C as they are adjacent in memory and in DFS order, and for object D.

	The length of the send list				**Traversal time (us)**			
Structure	(1-0-0)	(1-1-0)	(1-2-0)	(1-1-1)	(1-0-0)	(1-1-0)	(1-2-0)	(1-1-1)
BFS	1	2048	3072	3072	42	194	315	271
DFS	1	2	2	1	42	57	74	76

Table 5.2: Graph traversal of the object *array*.

DFS vs BFS traversal. Even though Skyway [108] uses BFS traversal for serialization, Naos exploits DFS due to the fact that Java objects are constructed in DFS order (i.e., a JVM first allocates memory for an object and then recursively for all its fields). Thus, DFS traversal has better memory locality that can be illustrated by traversing an object *array* from the following code snippet. Let us consider a class *Person* that has different graph structures denoted as (L0-L1-L2), where Li is the number of objects on the level i of the object graph (e.g., the object in Figure 5.5 has structure (1-2-1)).

```
1: Person[] array = new Person[1024];
2: for(int i=0; i<1024; i++)
3:   array[i] = new Person();
```

Table 5.2 reports the length of the send list after BFS and DFS traversals and corresponding traversal time for several object graphs. The data shows that for complex graph structures DFS provides much shorter send lists and faster traversal time.

Back-pointers. Naos sends objects directly from one heap to another. As a result, objects are sent containing pointers that are valid only in the sender address space, but not in the receiver's. Naos addresses this problem by sending extra metadata along with data objects, which is used by the receiver to efficiently recover the pointers (Section 5.3.3).

Algorithm 1 Was object o already visited?

```
 1: if o.addr = curr.addr + curr.len ∧ o.addr ≠ next.addr then
 2:   | curr.len ← curr.len + o.size                              ▷ hot-path
 3:   | return false
 4: if o.addr > curr.addr ∧ o.addr < next.addr then
 5:   | curr ← tree.insert_before(next, o)                        ▷ warm-path
 6:   | return false
 7: node, success ← tree.insert(o)                                ▷ cold-path
 8: if success then
 9:   | curr ← node
10:   | next ← curr.next()
11:   | return false
12: return true                                                   ▷ is a back-pointer
```

Naos is designed to send as little metadata as possible. The metadata contains a 24-byte header with object and metadata sizes and, if present, pointers to already visited objects ❹. These pointers are redundant edges after building a spanning tree over the object graph using DFS. We call them *back-pointers* since they always point to already visited objects in the send list (see Figure 5.5). For each back-pointer, a reference identifier representing the order by which the reference was visited in DFS order, and an offset within the send list where this reference should point to are sent to the receiver as metadata. In our example in Figure 5.5, only references 4 ($D \rightarrow C$) and 5 ($C \rightarrow A$) are sent.

All edges of the spanning tree (we call them *trivial-pointers* for simplicity) can be automatically inferred during a DFS traversal in the receiver (Section 5.3.3). This allows Naos to send no information about *trivial-pointers* resulting in a massive reduction of metadata sent over the network. Note the graphs without cycles do not contain back-pointers, which covers the vast majority of the most popular Java data structures.

Back-pointer/Cycle detection. To detect pointers to already visited objects (i.e., back-pointers), Naos uses a *memory interval tree* that keeps track of all visited memory intervals during DFS traversal. The interval tree is implemented using a red-black tree, which is selected over a hashtable (as Java and Kryo do) for two reasons. First, for large data structures, the hashtable grows (one entry per visited object) to large sizes and will lead to expensive lookups due to hash collision. Second, references to already visited objects are very rare and references pointing to objects in nearby memory positions are common in most Java popular data structures. Therefore, an interval tree, in most cases, contains a few large memory intervals, thereby ensuring fast lookups. We further optimize our

interval tree by providing different fast paths.

Algorithm 1 presents how Naos decides whether a particular object o has been already visited. Two helper variables are used: *curr* points to the last node inserted into the tree; *next* points to the tree node that follows *curr* in the tree. All tree nodes keep an initial address *addr* and its length *length*. If the object's address is adjacent to the last memory interval inserted into the tree, the insertion is performed in *O(1)* time (*hot-path*). If the memory pointer is higher than the current tree node and lower than the *next* tree node, then insertion is performed in *O(1)* (*warm-path*), unless the tree needs to be re-balanced, taking *O(log(n))* time . Otherwise, the memory pointer is inserted in the tree in *O(log(n))* time (*cold-path*).

As a comparison, Skyway does not use complex structures for cycle detection and simply extends the JVM header of Java objects by 8 bytes. Even though it ensures that the newness of an object can be be checked in *O(1)*, it results in a 15.4% increase in memory usage [108].

5.3.2 Network exchange of on-Heap Objects

Naos adds native RDMA communication to JVM without compromising JVM's memory safety. Naos' interface does not expose explicit RDMA access to the remote or local heap memory. Instead, its API allows only sending and receiving Java objects, hiding all the burden of low-level RDMA programming from the user. Internally, though, Naos fully relies on efficient one-sided RDMA communication to completely avoid redundant data copies. Naos also supports TCP for sending objects directly from its heap, but the use of RDMA requires overcoming peculiarities of managed languages such as concurrent garbage collection.

Blocking RDMA protocol. This section describes the *blocking* RDMA protocol for a single connection. All connections are handled independently and do not share resources. The core idea of Naos RDMA is that the receiver pre-registers buffers of fixed size in its heap and registers them for RDMA Write access. The sender uses RDMA Writes ❻ to write the objects from its local heap directly to the known reserved buffers in the remote heap ❼. The metadata is sent separately using a circular buffer ❽ for RDMA messaging [47, 122].

The protocol allows the sender to start writing memory to the remote heap even if the receiver did not call `readObject`, as illustrated in Figure 5.6(a). The sender can continue

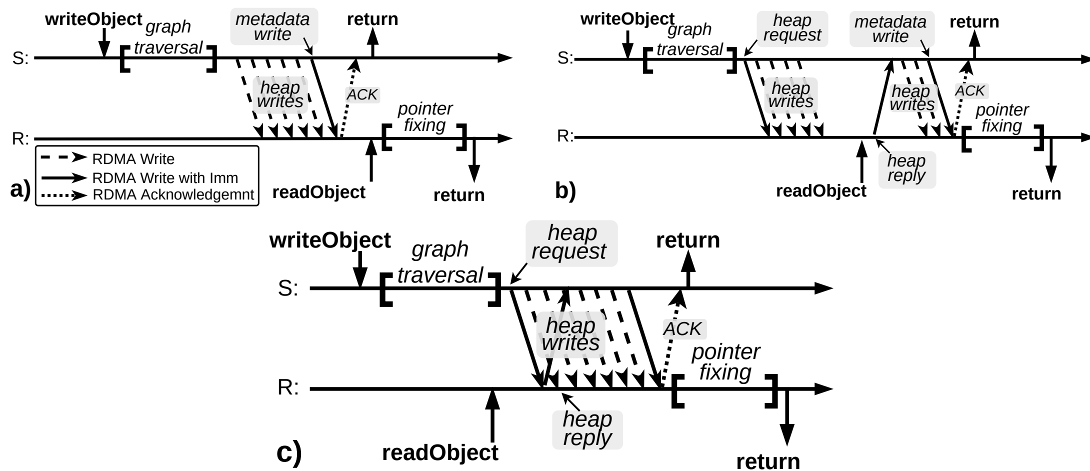


Figure 5.6: Blocking communication mechanism for three scenarios: **(a)** the sender can fit data to the pre-allocated receiver heap; **(b,c)** the sender needs to request extra heap memory. The receiver was not ready to receive data in **(b)**, and it was ready in **(c)**.

writing the data while it has enough free remote memory. Once the sender completes writing all objects to the remote heap using RDMA, it sends a separate completion message with metadata via the circular message buffer 8. The remote circular buffer is filled using *RDMA Write with immediate data*, which generates a completion event on the receiver after the write completes. The sender can unblock from sending once it receives an acknowledgment from the network indicating that all data has been written to the receiver. The acknowledgment is generated by the network and does not require the receiver's interaction. The receiver fetches the received object when it calls `readObject`, after all pointers are recovered (Section 5.3.3).

Sender's heap management. Naos utilizes *object pinning* to prevent a JVM garbage collector (GC) from moving objects until they are fully transmitted by the RNIC. Object pinning is already offered by some garbage collectors, such as Shenandoah [50]. Shenandoah is a high-performance GC that is supported by upstream OpenJDK. Besides object pinning, Naos also utilizes Shenandoah's memory allocator, that maintains the heap as the collection of fixed size Shenandoah regions. To pin and unpin objects efficiently, Naos pins whole Shenandoah regions containing the affected objects instead of pinning individual objects. During a send request, Naos pins and remembers all affected Shenandoah memory regions. Once the request completes, Naos unpins the regions associated with the request. Shenandoah allows pinning a region multiple times, and each region needs to be unpinned as many times as it has been pinned, thereby successfully preventing Naos from

accidentally unlocking the GC for unsent objects.

RNICs cannot simply send data from any buffer and communication buffers must be registered at the RNIC[1]. Thus, the sender must register the memory addresses of all objects it needs to send. However, RDMA memory registration is an expensive process that may take hundreds of microseconds for a single buffer [71, 104, 150]. Therefore, naive registration of all objects from the send list may completely cancel all performance advantages of RDMA. Naos addresses this issue by registering large fixed-size memory regions (i.e., Shenandoah regions) where the objects are allocated. It enables reusing a single memory registration for all objects stored in it, exploiting spatial locality. Naos also caches memory registrations to reuse them later for future sends, exploiting temporal locality.

Receiver's heap management. When the sender runs out of the remote buffers for writing, it sends a request to the receiver to register more on-heap memory, as illustrated in Figure 5.6(b,c). Thus, the sender can block until the receiver replies with new heap buffers, as in Figure 5.6(b). However, when the receiver is ready to receive data it can immediately reply to the heap request and do not obstruct the sender as in Figure 5.6(c). The receiver can reply to heap requests when it calls `readObject` or `isReadable`. During these calls, Naos checks for received requests by polling completion events from the RNIC. The process of handling requests is invisible to the caller, which hides the complexity of the underlying protocol from the user.

Upon receiving a heap request, the receiver allocates a new Java byte array buffer of fixed size inside the Java heap and *registers its payload* for RDMA Write access and replies with the RDMA address of the registered buffer. To prevent the GC from moving the reserved on-heap buffers, Naos utilizes *object pinning* offered by Shenandoah [50]. Importantly, the sender writes data to the *payload* of the pre-allocated byte array as it prevents the GC from reading invalid data. The main reason for that is that the unrecovered received objects have invalid class and object pointers (Section 5.3.3). Thanks to this enclosure, the GC observes only the array and skips reading objects stored in the payload.

The sender fills the remote buffers in the order it received them from the receiver, constituting a queue of remote heap buffers. Since pre-registered RDMA heap buffers are of fixed size, the sender is not always capable of fully utilizing them. To address this issue,

[1]Modern RNICs support implicit on-demand paging (ODP) [88] that removes the need to register buffers. In our preliminary experiments, however, ODP performed worse than conventional explicit memory registration.

the sender informs the receiver about how many bytes were unused in each *finalized* heap buffer by sending *heap truncate request*. A buffer becomes finalized when the sender jumps to the next buffer in the queue. After receiving the data, the receiver revokes RDMA access to finalized buffers and then unpins them to enable the GC for received objects. It also deallocates unused memory of the finalized buffer and removes the array header to make all received objects visible to the GC.

Nonblocking object sending. The main difference between the blocking `writeObject` and the nonblocking `writeObjectAsync` is that the later returns right after the dispatching *metadata write* request to the device. The nonblocking call submits all communication requests to the RNIC but does not wait for a network acknowledgment. Instead, Naos returns a request handle that can be used by an application to confirm the delivery of the object using `testHandle` call. Compared to the blocking call, Naos prevents the GC from moving affected objects even after the call returns. Naos *pins* the affected objects before exiting the JVM, and unpins them later once the corresponding acknowledgment is received.

Naos TCP. Naos supports sending objects directly from the heap using TCP as well. Unlike RDMA connections, a traditional TCP socket connection has a single datapath. Thus, to send the objects to the remote heap, the TCP sender first writes the metadata to the socket and then all elements of the send list. The receiver first reads metadata to a temporary buffer from its socket, then, to avoid redundant data copies, it directly reads the data from the socket to the heap. For that, it allocates a byte array buffer of the required size inside the Java heap, and then reads the data from the socket to the payload of the allocated buffer.

Network buffering. Naos is designed to send data directly from the heap without intermediate buffering. However, the size of a JVM object can be as small as 24 bytes. Thus, a highly sparse object graph can result in a lot of small writes to the network, which can significantly reduce the network performance. To address this issue, Naos may buffer small objects before sending them to the remote heap. Large objects are still sent directly from the heap. Naos sends buffered objects once it batches enough bytes to utilize the network, or when a large object needs to be flushed to preserve DFS object order (Section 5.3.1).

An alternative approach is to use scatter-gather capability of RNICs [96] for RDMA networking and scatter-gather I/O for TCP sockets. The scatter-gather networking enables building a network message from multiple buffers without intermediate buffering. The

current version of Naos does not implement it, but it is an interesting direction for future research.

Memory safety of Naos. Naos uses reliable transport to ensure the delivery of transmitted data. Naos materializes only fully received objects, which prevents returning partially received objects from a faulty sender. Faulty sends can be detected during graph recovery from the network errors provided by the reliable transport. If an error is detected, the receiver revokes RDMA access to pre-allocated buffers and deallocates the unused memory.

Naos' implementation follows all security advice related to RDMA networking [133], therefore, we believe that Naos does not open security breaches. In particular, the pre-allocated heap buffers are not shared between connections preventing remote JVMs to access buffers of each other. In addition, each sender registers its heap only for local read access preventing other remote JVMs to access it. Finally, remote read access is disabled, and Naos only temporarily enables write access to pre-allocated in-heap buffers, which are private for each sender. Once the in-heap buffer is full, the write access is revoked.

For compatibility between communicating applications, Naos requires that communicating JVMs have the same memory layout of in-heap objects. This can be achieved by running the same JVM with the same settings including GC.

5.3.3 Object Graph Recovery

Naos sends unmodified memory segments from one heap to another. As a result, objects are sent containing pointers that are valid only on the sender address space, but not on the receiver's. Naos' graph recovery algorithm overwrites these pointers making them valid on the receiver's address space. Java objects have two types of pointers: **class pointers** and **object pointers**. Class pointers point to JVM-internal data structures that describe Java types. Object pointers are reference fields that point to other on-heap Java objects.

Naos uses a recovery approach different from the one used in Skyway [108]. Since Skyway copies objects to communication buffers, it can afford modifying data before sending. Thus, Skyway simply replaces class pointers with integers (as Kryo does) and object pointers with their relative offsets within the communication buffer. Such design allows the receiver to simply replace class integers with corresponding class pointers and relative object offsets with corresponding absolute addresses. Unlike Skyway, Naos sends objects directly from the heap using RDMA requiring more sophisticated algorithm for pointer fixing in return for not requiring data copying.

Algorithm 2 Object Graph Recovery

```
1: buffer                                          ▷ the buffer with received objects
2: refid ← 0                                       ▷ the number of traversed references
3: offset ← 0                                      ▷ current offset in the receive buffer
4: stack.push(new field(), new hint())             ▷ push dummy field and hint
5: while stack.is_not_empty() do
6:   field, hint ← stack.pop()
7:   FIX_FIELD_POINTER(field, hint)
8:   refid ← refid + 1
```

Phase 1 – Fix Field Reference

```
9: procedure FIX_FIELD_POINTER(field, hint)
10:   if refid = cur_back_pointer.id then          ▷ a back-pointer
11:     field.ptr ← buffer + cur_back_pointer.offset
12:     cur_back_pointer ← get_next_back_pointer()
13:   else                                         ▷ a trivial-pointer
14:     obj ← (obj)(buffer + offset)
15:     field.ptr ← obj
16:     FIX_CLASS_POINTER(obj, hint)
17:     ITERATE_FIELDS(obj, hint)
18:     offset ← offset + obj.size
```

Phase 2 – Fix Class

```
19: procedure FIX_CLASS_POINTER(obj, hint)
20:   if hint.rem_class = obj.class then
21:     // hint is correct, do nothing             ▷ hot-path
22:   else
23:     if class_cache.contains(obj.class) then
24:       new_hint ← class_cache.get(obj.class)    ▷ warm-path
25:       hint.update(new_hint)
26:     else
27:       new_hint ← class_service(obj.class)      ▷ cold-path
28:       class_cache.put(obj.class, new_hint)
29:       hint.update(new_hint)
30:   obj.class ← hint.loc_class
```

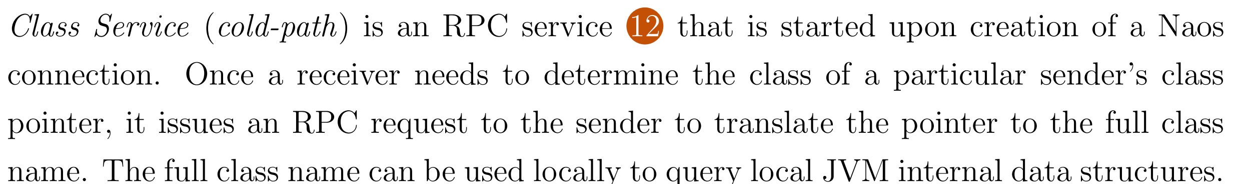
Phase 3 – Iterate Fields

```
31: procedure ITERATE_FIELDS(obj, hint)
32:   for field, field_hint in hint.fields do
33:     stack.push({obj + field.offset, field_hint})
```

Algorithm 2 describes the Naos' graph recovery approach that starts with a DFS traversal of the object fields (lines 5-8). The traversal is initialized by pushing a *dummy* field pointing to the first received object. The graph recovery terminates when the DFS *stack* is empty. At that point, all pointers are valid in the receiver's heap and the first object can be safely returned to the user.

Fixing Field References. For every object field found in the stack, the algorithm applies *FIX_FIELD_POINTER* procedure, which investigates whether the tested reference is a *back-pointer* or a *trivial-pointer* by checking whether the received metadata contains the current reference ID (line 10). For *back-pointers*, the *offset* associated with the current pointer is used to fix the reference. If the reference is a *trivial-pointer*, the new memory address can be determined by just using the current *offset* in the *receive* buffer (line 14). For a trivial-pointer, the next step is to fix the class field of the pointed unvisited object (line 16). Note that Naos sends no metadata for trivial pointers, since the sender and the receiver traverse the graph in the same DFS order, providing a significant reduction in metadata size.

Fixing Class References. Updating class pointers is a particularly expensive operation if not designed carefully, since the class pointer needs to be fixed for every object. To achieve high performance, Naos proposes a 3-way approach:



Class Service (*cold-path*) is an RPC service 12 that is started upon creation of a Naos connection. Once a receiver needs to determine the class of a particular sender's class pointer, it issues an RPC request to the sender to translate the pointer to the full class name. The full class name can be used locally to query local JVM internal data structures.

Class Map (*warm-path*) is a per-connection table that caches all class translations. However, accessing a table for every object reference still produces a large overhead, especially in large graphs. To overcome this limitation, Naos proposes the use of Speculative Type Graphs (STG), a type of polymorphic cache inspired by [58].

STG (*hot-path*) is a data structure that dynamically captures type relations in the object graph, providing a translation *hint* for each class pointer. Each STG hint caches: i) a translation between a local and remote class pointer; ii) class description including object fields; iii) pointers to other hints for each field allowing to build hints recursively (lines 32-33). Using STG, Naos can speculate on the type of a particular object using a hint. If the hint is correct the class translation and retrieval of a class descriptor takes *O(1)* time (line 20). Speculation might fail due to type polymorphism in Java (line 22) and, in that

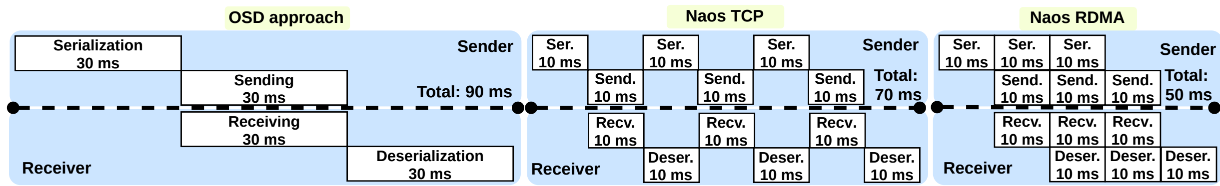


Figure 5.7: The communication benefits of Naos' pipelining compared to the conventional OSD approach.

case, the cache is used for resolving the class pointer and the STG is updated (line 25) with the new translation hint. In practice, however, most data structures have very regular type graphs allowing the STG to guess correctly most of the times.

After the class pointer of an object is fixed, Naos iterates all its reference fields (line 17). Naos utilizes object's class pointer translation hint to create translation hints for its reference fields (lines 32-33), which are then pushed into the *stack* together with the corresponding object reference.

5.3.4 Overlapping network and graph traversal

An important disadvantage of conventional serialization approach is that it does not support overlapping serialization and communication: an object must be fully serialized before sending it over a network. Similarly, the receiver cannot start deserialization unless it receives all the data (see Figure 5.7). As a result, applications can suffer from high end-to-end latency for large object graphs.

Naos supports pipelining graph traversal with communication on the sender and pointer fixing with object receiving on the receiver. Both Naos TCP and Naos RDMA benefit from pipelining as it allows the receiver to start pointer fixing of *partially received object graphs*, thereby reducing end-to-end latency. Using offloaded RDMA communication, Naos RDMA can continue traversing the graph after submitting write requests to the RNIC, thereby overlapping communication and graph traversal on both the sender and the receiver.

Pipelining in Naos is implemented by pausing the object traversal and sending partial graphs to the remote heap. A partial graph contains only objects and back-pointers found at a given traversal stage. The receiver can read the partial graph and start pointer fixing. Once all received objects are traversed, the receiver reads the next fragment of the graph.

Figure 5.7 illustrates how Naos with pipelining improves communication latency of large object graphs compared to the OSD approach. The OSD approach cannot break serialization of a single graph, which results in 9 ms latency. Naos TCP can send partially traversed graphs reducing the latency by 2 ms, but cannot overlap computation with communication. Naos RDMA enables overlapping communication and graph traversal, which reduces the latency by another 2 ms.

5.4 Evaluation

We evaluate the performance of Naos and compare it with Java, Kryo, and Skyway[2] serialization engines using four different classes of workloads. First, the performance of Naos is studied by transferring data structures that are commonly used in distributed applications. The goal is to measure the performance benefits of the different techniques proposed in Naos and the trade-offs involved depending on the shape of object graph. In addition, it also shows the impact of using RDMA instead of TCP. Second, we study the role of data streaming and pipelining in OSD performance. Then, we show results for integrating the Naos library into Apache Dubbo [7], a high-performance RPC framework developed in Java, to show the impact of Naos on RPC workloads. Lastly, we use a map-reduce implementation of PageRank to measure the performance of Naos for data processing workloads.

Experimental setup. All experiments were performed on a cluster of 4 nodes interconnected by 100 Gbit/s Mellanox ConnectX-5 NICs. Each node is equipped with an Intel(R) Xeon(R) CPU 6154 @ 3.00 GHz and 384 GB of RAM.

Implementation details. Naos is implemented and tested for OpenJDK HotSpot 11.0.6 [6], a widely-used production JVM. Naos does not require changes to the internals of the JVM and is implemented as a JNI plugin and a Java-level library that allows users to write objects directly to TCP and RDMA connections. Naos TCP provides constructors to create a Naos connection from TCP connections of various network libraries (e.g., `java.net.Socket`). Naos RDMA does not rely on existing JVM RDMA libraries and fully implements a specialized RDMA network library including an API to create and

[2]We could not compare with the original Skyway as it is not open-source. Therefore, we re-implemented Skyway following the instruction provided in the paper [108]. Note that we did not extend object headers by 8 bytes for cycle detection and simply evaluated Skyway without cycle detection.

connect RDMA endpoints. Our plugin is implemented in Java and C++ and depends on: *libibverbs*, an implementation of the RDMA verbs, and *librdmacm*, an implementation of the RDMA connection manager.

RDMA communicators for Java and Kryo serializers have been implemented using Disni [143] RDMA library, a high-performance Java RDMA library that encapsulates native C RDMA verbs API. The Disni library is used by Java applications such as Spark [97], Crail [145], and DaRPC [144]. Note that Skyway cannot be used with existing RDMA libraries, including Disni, as these libraries can only work with specialized off-heap memory residing outside of the Java heap memory, whereas Skyway requires the memory buffers reside inside the heap memory to deserialize objects. These limitation stems from the fact that garbage collection can move on-heap buffers while they are being accessed by the RNIC.

In all experiments, the JVM was configured with default parameters and enabled Shenandoah garbage collector as it is the only collector that is currently supported by Naos. Shenandoah was configured with 32 MiB memory regions. Naos was configured with 20 MiB receive buffers. If not stated differently, Naos and all serialization algorithms were deployed without graph cycle detection and with no pipelining (Section 5.3.4).

5.4.1 Serializing Java Data Structures

The performance of OSD approaches is measured using three data structures that are among the most common serialized data structures in real-world workloads deployed in platforms such as Spark, Hadoop, and Flink: a) an array of `float` primitive types, which is common for machine learning workloads; b) an array of `class Point` containing only two primitive types, which represents a 2D Euclidean point; c) an array of `class Pair` containing an integer and a char array, which represents a key-value pair, in many algorithms such as Word Count. In our experiments the char array had length 5, the average word length in the English language.

Benchmarks are carefully designed to guarantee the optimal configuration of all serializers. In particular, for Java, Kryo, and Skyway, all buffers are pre-allocated with the correct size to avoid re-allocation and memory copies during the serialization process. Besides, all types were pre-registered in Kryo to guarantee maximum data format compression. Measurements are taken after a JVM warmup (of at least 100 ms) until convergence of the JIT compiler to achieve maximum performance. All experiments run in complete isolation for several seconds and the aggregated statistics are reported.

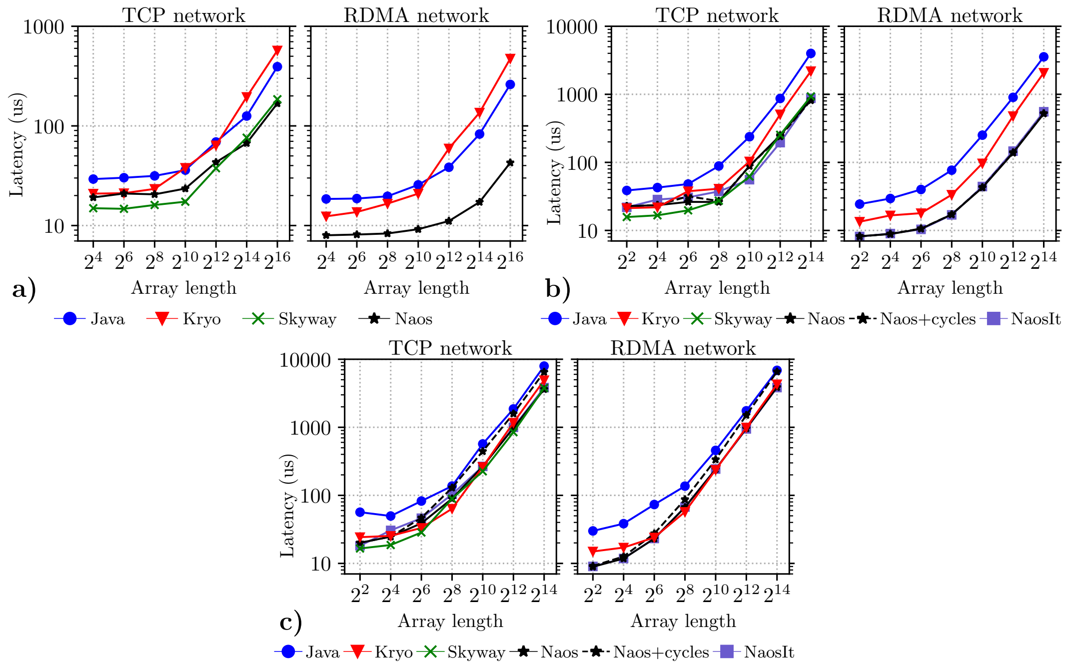


Figure 5.8: Latency in us for **a)** an array of floats **b)** an array of Points **c)** an array of Pairs. The y-axis is in log scale.

Latency. Figure 5.8 shows the average latency of transferring the aforementioned data types with increasing their size.

Naos performs excellently for contiguous data structures such as the array of float, as it can send them from the heap without making extra copies and using fewer RDMA requests. For comparison, Kryo, Java, and Skyway must first serialize objects to a dedicated send buffer. RDMA-Naos' latency can be as small as 8 μs, which is at least a 2x and a 2.4x improvements over Kryo and Java serializers, respectively, for small arrays, and at least a 4.5x for large arrays. For example, Naos RDMA needs only 42 μs to send 2^{16} floats, whereas serialization approaches need at least 190 μs.

Naos RDMA has lower latency than Skyway, however, Skyway performs better than TCP-Naos for small arrays because of two reasons. First, Naos buffers small objects (less than 256B) to better utilize the network (Section 5.3.2). Second, Naos TCP allocates on-heap memory after data arrives, whereas Skyway has all buffers preallocated in our experiments. Both reasons give an advantage to Skyway over Naos TCP for small arrays. For large arrays, Naos TCP provides a 9.1% reduction in latency over Skyway as it incurs fewer data copies.

An array of float is the simplest object graph for graph traversal as it contains a single contiguous object. An array of Point, however, is non-contiguous in memory as this array contains references to objects of class Point, which are 32 bytes each. Nonetheless, Naos provides a 2x and a 4x improvements on average over Java and Kryo for RDMA networks, even with cycle detection enabled (*+cycles*). *Naos+cycles* benefits from our hot-paths of Algorithm 1 as the JVM tends to collocate objects in memory even for the potentially sparse object graphs. The experiment shows that moderately sparse graphs with small objects are not an issue for Naos.

An array of Pair is even sparser graph than the array of Point, as the class Pair has more references than the class Point. Naos RDMA still achieves the lowest latencies for all sizes. However, with cycle detection, Naos' traversal is slower for long arrays compared to Kryo. The main problem is that Naos sends more data than conventional serializers since it needs to send a JVM header of 16 bytes for each Java object. We conclude that Naos does not always provide lower latency compared to conventional OSD approaches and that its performance depends on sparsity and the number of traversed objects.

A shortcoming of Skyway's and Naos' data format is that they do not compress arrays with references and are forced to send long arrays with (invalid) pointers, whereas Kryo can encode this information in few bytes. To address this issue, we designed a specialized call for Naos, namely *NaosIt*, that sends only objects stored in an array. The receiver of such compressed message creates a new array and then fills it with received objects. *NaosIt* reduces the size of communicated data, but requires extra memory allocation on the receiver. Overall, NaosIt provides a small improvement over Naos, as the experiments are performed on 100 Gbit/s network. Such compression would be more beneficial for slower networks.

CPU and network costs. To show the key differences between Naos networking and the traditional OSD approaches, Table 5.3 shows the time breakdown of transferring various data structures and their network cost. Naos as a serialization-free approach always has zero cost for serialization and deserialization. Naos' graph traversal time is included in the send time. The OSD approaches with RDMA has zero receive cost as the data delivered directly to pre-allocated receive buffers by the RNIC. Naos, on the other hand, has non-zero cost as the receive time includes the graph recovery.

Object serialization in TCP experiments takes longer than for RDMA. The difference comes from the fact that in TCP experiments the data is serialized to on-heap buffers, which can be affected by the GC, whereas RDMA requires data to be serialized to off-heap

Test	TCP				RDMA				Size (B)
	Ser.	Send	Receive	Deser.	Ser.	Send	Receive	Deser.	
Array of native float with 8192 elements									
Java	15-18	6-9	6-8	17-24	14-17	0-1	0	15-20	32795
Kryo	24-29	7-10	6-8	26-31	24-27	0-1	0	25-86	32772
Skyway	2-3	7-9	7-8	0-1	NA	NA	NA	NA	32792
Naos	0	7-9	16-52	0	0	10-11	0-1	0	32792
Array of class Point with 1024 elements									
Java	109-116	5-7	4-6	94-102	112-119	0-1	0	113-121	14469
Kryo	44-47	4-6	2-3	36-39	43-47	0-1	0	38-41	8132
Skyway	23-26	6-8	6-8	10-11	NA	NA	NA	NA	28696
Naos	0	22-26	27-76	0	0	25-26	13-15	0	28696
NaosIt	0	22-23	28-39	0	0	24-25	15-17	0	24576
Array of class Pair with 1024 elements									
Java	216-664	7-19	10-12	208-222	211-223	0-1	0	217-230	30864
Kryo	135-231	5-10	6-8	79-83	135-148	0-1	0	80-83	18436
Skyway	149-154	9-13	15-31	23-24	NA	NA	NA	NA	61464
Naos	0	161-168	84-137	0	0	199-206	34-39	0	61464
NaosIt	0	159-164	109-138	0	0	200-206	36-40	0	57344
	CPU sender		CPU receiver		CPU sender		CPU receiver		Network

Table 5.3: CPU time breakdown (in μs) and Network cost for transferring arrays. Percentiles 5 and 95 are reported.

buffers, that are invisible to the GC.

Java and Kryo for RDMA have the same send cost which is the cost of submitting offloaded RDMA request to RNIC. Blocking Naos RDMA has higher cost to send as it needs to wait for a network acknowledgment to finish sending.

For all data types, Naos RDMA shows at least a 2x reduction in CPU time for receiver over Kryo and Java. The main reason is that conventional serialization libraries need to allocate and initialize memory for each received object. Naos does not construct objects and only fixes pointers in the received data. For senders, however, Naos is better at reducing CPU cost for simple graphs such as arrays of floats and points. Note that Naos TCP has a longer receive time than Skyway as it needs to allocate receive memory, whereas Skyway worked with pre-allocated buffers in our experiments.

The network cost of Naos and Skyway increases with the number of transmitted Java objects. For an array of floats, therefore, the size of the transmitted data is approximately

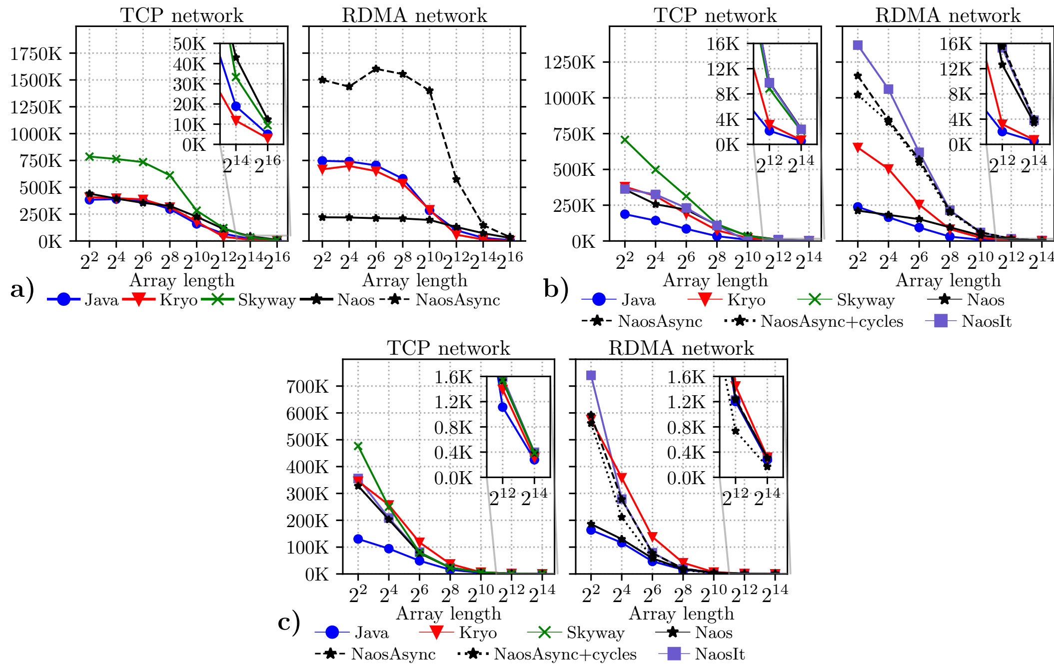


Figure 5.9: Throughput in objects/sec for **a)** an array of floats **b)** an array of Points **c)** an array of Pairs.

the same for all approaches. On the other hand, for an array of Points or Pairs, the network cost of Naos is about 2x higher in comparison with Java and about 3.5x over Kryo. Kryo has the lowest network costs as it replaces the class descriptors with integer identifiers significantly compressing object graphs. Naos and Skyway have the same network cost as they have the same data format, but our NaosIt provides a reduction in the network size for array containers.

Throughput. In this experiment, senders continuously send objects to the receiver. For RDMA approaches with serializers, we provide at the sender and the receiver a large number of send and receive buffers to enable asynchronous communication so that the sender can start serializing and sending the next object without the need to wait for the completion of the previous requests.

Figure 5.9**(a)** shows that Naos TCP was not able to significantly outperform Skyway for small arrays, as the throughput of Naos was mostly limited by the receive buffer allocation, whereas Skyway, with pre-allocated memory, achieved 750K requests/sec. For arrays larger than 2^{12} elements, however, Naos TCP outperforms Skyway as the cost of data copies at the sender overwhelms the cost of memory allocation at the receiver, showing the advantage of our zero-copy design.

The performance of blocking Naos RDMA is bound by the network latency, which prevented the application to send requests at a higher rate. The NaosAsync RDMA, which avoids waiting for an acknowledgment, achieves the highest performance showing the importance of asynchronous communication. For the array of 512 floats, Naos achieves 1600K requests/sec, which is a 2x speedup over existing serialization approaches.

Figures 5.9**(b,c)** show that the throughput of Naos RDMA was limited by the network bandwidth since NaosIt, that communicates less data, outperforms NaosAsync RDMA. This observation indicates the benefit of our data compression.

The cycle detection decreases the throughput of Naos by less than 3% for moderately sparse graphs. For sparser graphs such as an array of Pairs the slowdown increases to 19%, which is explained by the growth of the Naos' interval tree for cycle detection. Therefore, Naos has lower performance than Kryo, but still outperforms the Java serializer. We think that, in real systems, Naos can be used together with traditional OSD libraries depending on the sparsity of the object graph.

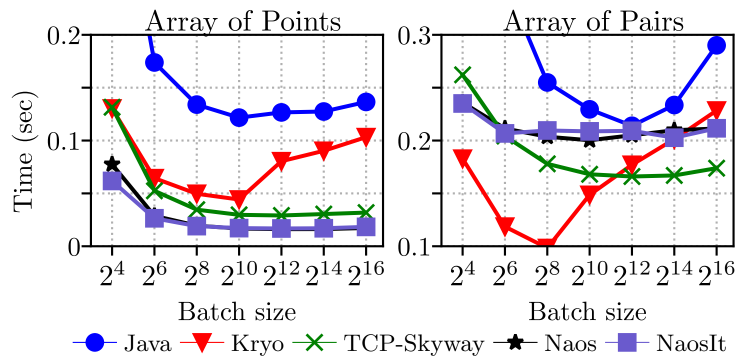


Figure 5.10: Streaming an array of 2^{20} elements.

Streaming data transfers. Data processing frameworks such as Spark and Flink rely on data streaming to enable processing of continuous streams of data. The continuous data stream is generated by sending small chunks of data to the processing nodes. To represent this use-case we implemented the streaming of long data arrays over RDMA networks. Figure 5.10 shows the streaming time of an array of Points and Pairs with increasing the chunk size.

For the array of Points, Naos RDMA outperforms all serializers for all chunk sizes, and decreases the streaming time of Kryo by 2.1x. For the array of Pairs, Kryo has the highest performance, by sending 256 objects at a time, due to its ability to compress the objects

efficiently. For larger chunks, Naos and Skyway take less time than Kryo, since Kryo starts suffering from longer object construction for larger chunks, whereas Skyway and Naos do not need to construct objects.

Even though Skyway and Naos have the same data format, Skyway streamed the array of Pairs faster than Naos. The difference comes from the complexity of Naos' communication algorithm, leading to the higher CPU cost at the sender (see Table 5.3). Skyway's serialization code only copies traversed objects to send buffers, whereas Naos as a communication library needs to take more factors into account: building send lists, RDMA memory registration, and triggering multiple RDMA requests. Naos could employ various modern RDMA techniques for optimized memory accesses [96], which are interesting directions for future research.

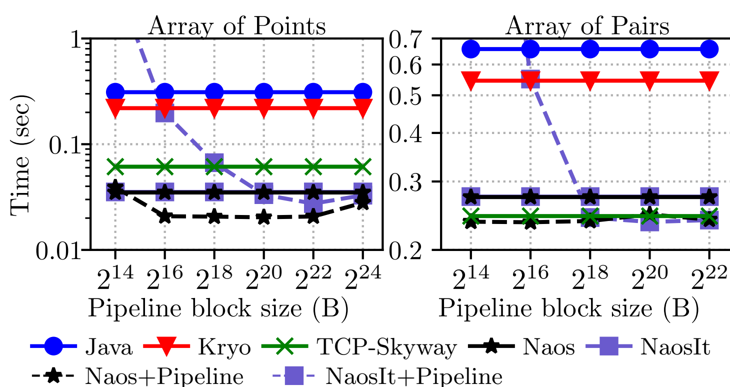


Figure 5.11: Pipelining an array of 2^{20} elements.

Pipelining data transfers. Naos supports pipelining graph traversal with communication on the sender, and pointer fixing with communication on the receiver. Unlike Naos, conventional OSD approaches require an object to be fully serialized before sending it over the network. In this experiment, we show the effect of pipelining for large object graphs by measuring the time of transferring arrays with 2^{20} elements.

The latencies of Java, Kryo, Skyway, and Naos with no pipelining are depicted as straight lines in Figure 5.11 as they are independent of the pipeline size. Naos with pipelining provides a 20% reduction in latency in comparison with a non-pipelined variant, since the receiver can start pointer recovery earlier. Note that in the previous experiments with streaming large sparse object graphs, Kryo outperformed Naos as it could split the graph into chunks. For inseparable large graphs, however, Naos takes less time even for highly sparse graphs.

5.4.2 Accelerating applications with Naos

Naos provides a simple programming interface (see Table 5.1) hiding all the burden of low-level RDMA communication. In particular, RDMA benchmarks from the previous experiments take only 10 lines for Naos and over 300 lines for the Disni RDMA library. Thus, we believe that it is simple to build systems using Naos. As proof, we have extended Apache Dubbo with Naos communicator, and implemented a Naos-enabled map-reduce framework.

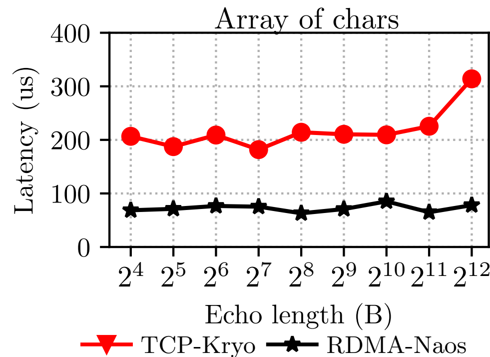


Figure 5.12: Dubbo RPC latency.

Zero-copy RPC messages with Dubbo. To show that Naos is easy to use, we extended an RPC library Apache Dubbo with the Naos communicator. For that we added a new Naos-enabled communication module that has no serialization module.

In the first experiment we measure the latency of an RPC function that echoes back a Java String. Naos' performance was compared with the default TCP network library, Mina [9], with Kryo serializer. Naos was deployed with cycle detection. Note that Dubbo besides an RPC arguments also sends an RPC metadata resulting in sending several Java objects. Figure 5.12 shows that employing Naos RDMA decreases the latency by at least 55% for all tested sizes.

Workload	(50:50)	(95:5)	(100:0)
TCP-Kryo	14K-24K	16K-24K	23K-25K
RDMA-Naos	194K-266K	196K-266K	219K-281K

Table 5.4: Throughput in requests/sec under YCSB workloads with various (read:write) ratios. Percentiles 5 and 95 are reported.

	LiveJournal [13] (2 nodes)				Orkut [170] (3 nodes)			
	TCP		RDMA		TCP		RDMA	
Test	Total	Stage	Total	Stage	Total	Stage	Total	Stage
Java	413.46	3.67-4.04	406.87	3.53-3.99	364.53	3.20-3.38	365.29	3.10-3.44
Kryo	410.94	3.62-4.06	411.33	3.63-3.99	357.58	2.98-3.47	354.06	2.91-3.42
Skyway	394.32	3.52-3.77	NA	NA	350.48	3.07-3.24	NA	NA
Naos	393.05	3.54-3.76	395.05	3.59-3.70	342.06	2.85-3.32	343.00	2.84-3.35
NaosIt	394.49	3.56-3.77	386.01	3.48-3.69	345.94	2.82-3.45	333.16	2.72-3.24
Skyway†	386.31	3.54-3.78	NA	NA	340.82	2.95-3.30	NA	NA
Naos†	373.04	3.27-3.72	369.10	3.16-3.63	331.31	2.86-3.22	335.50	2.85-3.22

† PageRank with sparsity-aware implementation.

Table 5.5: Total and per stage processing times in seconds for 100 iterations of PageRank algorithm. Percentiles 5 and 95 are reported for PageRank iterations.

To understand the performance of Naos under a realistic throughput workload, we built a key-value store (KVS) using a Java concurrent hashtable and Dubbo library for communication. We populated the KVS with one million entries of 1 KiB each. We benchmark it under different YCSB [37] workloads. Table 5.4 shows that Naos RDMA achieves an average speedup of 11x over TCP-Kryo. The speedup comes from the fact that TCP-Kryo was bottlenecked by the CPU, whereas Naos consumes less CPU time to send a KVS request. The experiment shows that Naos can be utilized for KVS workloads as KVS requests and responses have low sparsity.

In comparison with microbenchmarks (Section 5.4.1), the performance difference between Naos and Kryo is much higher for the current workload than for the microbenchmarks, where Kryo's performance was measured after JIT compilation that significantly improved its performance for repetitive sending of the same object. Since Naos does not depend on Java runtime optimizations, it can achieve much higher performance than Kryo for dynamic workloads.

Improving Data Processing Applications. We could not integrate Naos into Spark as its shuffle module is designed to communicate files with serialized objects. Integration of Naos would require a substantial redesign of Spark's code base. Therefore, we implemented our own map-reduce framework that takes advantage of Naos. Our framework supports all discussed serializers including Skyway and also offers RDMA networking with Disni.

It was designed to resemble Spark but perform shuffle completely in-memory.

We evaluate the OSD approaches by running PageRank on real-world graphs as input: LiveJournal [13] and Orkut [170]. The LiveJournal dataset was processed with two shuffle workers and Orkut with three shuffle workers. Naos was deployed with 256 KiB pipelining and without cycle detection. We report total runtime including data loading and 5 and 95 percentiles for processing a single Pagerank iteration. We also provide two implementations of PageRank: the first one follows conventional design where each score update is a class of 32 bytes; the second implementation was designed to communicate dense contiguous score updates, thereby reducing sparsity of communicated shuffle blocks.

Table 5.5 shows the lowest runtime was achieved by Naos and Skyway for the first implementation. A side effect of Naos and Skyway is that, after receiving, objects are always contiguous in memory, thereby improving data locality. As a result, an application can process such contiguous objects faster as fewer memory pages need to be fetched. Overall, Naos TCP performs approximately as Skyway, but NaosIt RDMA provides 2.1% and 4.8% improvement over Skyway for LiveJournal and Orkut, respectively. The experiment shows that zero-transformation approaches for OSD can reduce processing time for data-processing workloads.

The sparsity-aware implementation provides an additional 4% reduction in runtimes, showing that applications need to take Naos' limitations into consideration to achieve the highest performance. Thus, Naos could be used in combination with works on data sparsity reduction for JVMs [167, 166, 25].

5.5 Summary and Discussion

We have presented Naos, a JVM communication library that enables transferring objects directly from one heap to another over the network with minimal CPU involvement and zero-copy . We demonstrated that existing OSD techniques are bound to CPU and that, as networks get faster, they will become the bottleneck of distributed systems. Naos completely avoids the need to serialize and deserialize objects for data transfers, with the corresponding performance advantages. Naos provides a simple API that simplifies the use of RDMA from JVM-based applications. Our evaluation shows that Naos outperforms all existing OSD approaches for moderately sparse object graphs.

6

Efficient NIC-based Authentication and Encryption for RDMA

State-of-the-art remote direct memory access (RDMA) technologies have shown to be vulnerable against attacks by in-network adversaries [133], as they provide only a weak form of protection by including access tokens in each message. A network eavesdropper can easily obtain sensitive information and modify bypassing packets, affecting not only secrecy but also integrity. Tampering with packets can have drastic consequences. For example, when memory pages with code are changed remotely, altering packet contents enables remote code injection.

In this chapter we propose sRDMA, a protocol that provides authentication and encryption for RDMA to prevent information leakage and message tampering. sRDMA is designed to introduce minimal changes to the existing InfiniBand protocol with minor performance overheads. For that, sRDMA extends the existing InfiniBand transport and the pro-

gramming interface with low-overhead symmetric cryptography. Additionally, we improve our design by introducing PD-level keys to reduce the memory overhead on the network controllers, and augment the InfiniBand architecture with extended memory protection.

The materials used in this chapter have been published at the USENIX Annual Technical Conference (USENIX ATC) in 2020 [151]. The paper was done in collaboration with Benjamin Rothenberger and our advisors Torsten Hoefler and Adrian Perrig.

6.1 Motivation

Despite numerous state-of-the-art systems [47, 28, 111] leveraging remote direct memory access (RDMA) primitives to achieve high performance guarantees and resource utilization, current RDMA technologies lack any form of cryptographic authentication or encryption. Instead RDMA mechanisms provide a weak form of protection by including access tokens in each message. Given that RDMA networks are mainly used in data-center environments and at large-scale deployments, detecting bugged wires is seemingly impossible. But not only in-network adversaries are an issue, also malicious end hosts can affect the security of an RDMA network. If an adversary is able to obtain control over a machine in an RDMA network (e.g., by escaping its virtual machine or hypervisor confinement in a cloud service [161]), it can fabricate and inject arbitrary packets. If the adversary can guess or obtain the memory protection tokens (which are transmitted in plaintext), it can read and write memory locations that have been exposed using RDMA on *any machine in the network*, leading to a powerful attack vector for lateral movement in a data center network.

Given these threats, the security of current RDMA data center networks highly depends on isolation. However, even isolation cannot defend against in-network attackers. Thus, RDMA networks require cryptographic authentication and encryption. Unfortunately, application-level encryption (e.g., *TLS* [128]) is not possible, since RDMA read and write can operate as purely one-sided communication routines. Furthermore, such an approach requires employing a temporary buffer for incoming encrypted messages. The message would then be decrypted by the CPU and copied to the desired location, which would cause high overhead—negating RDMA's advantages. Additionally, cryptographic protection using *IPSec* [45] does not support RDMA traffic as the protocol is unaware of the underlying RDMA headers and achieves no source authentication (see Section 6.7).

In our work, we introduce a secure RDMA (sRDMA) design using a secure reliable connection (SRC) queue pair (QP) that uses symmetric cryptography for source and data authentication and employs Network Interface Cards (NICs) to perform cryptographic operations. Symmetric cryptography reduces the computational overhead compared to asymmetric cryptography by 3–5 orders of magnitude. Thus, it is suitable for high-performance and low-latency applications based on RDMA, e.g., [70, 28]. Since symmetric cryptography introduces per-connection memory overhead and memory on NICs is constrained, we augment our proposed mechanisms using protection domain level keys and efficient dynamic key derivation, which eliminates the need for storing QP-level keys and drastically reduces the memory overhead on RDMA-capable NICs (RNICs).

Contributions. We design a SRC QP that effectively prevents attacks in an RDMA network, with minimal changes to the current InfiniBand architecture (IBA) standard (Section 6.4.2). We improve our design by introducing PD-level keys to reduce the memory overhead on the RNIC (Section 6.4.5), and augment IBA with extended memory protection that permits memory accesses only to trusted entities. We provide an implementation of our design using modern programmable network adapters equipped with ARM multi-core processors [24, 153] (Section 6.5). We extensively evaluate our design using artificial and real-world traces. Additionally, we modified the RDMA-based key value store, HERD [70], to make use of sRDMA (Section 6.6).

6.2 Background on InfiniBand Transport

Several network architectures support RDMA: InfiniBand (IB) [10] , RDMA over Converged Ethernet (RoCE) [11], and internet Wide Area RDMA Protocol (iWARP) [126]. InfiniBand is a network architecture fully designed to enable reliable RDMA with its own hardware and protocol specification. RoCE is an extension to Ethernet to enable RDMA over an Ethernet network. Finally, iWARP is a protocol that allows using RDMA over TCP/IP. In this work, we focus on the InfiniBand and RoCE as they are the most widely used interconnect for RDMA, but the proposed ideas can be easily extended to other RDMA architectures.

Several transport types are supported by the IBA to communicate between endpoints: reliable connection (RC), unreliable connection, unreliable datagram, extended reliable connection, and raw packet. In this paper, we only consider the RC transport type, since

it is the only type that supports both RDMA read and write requests.

The RC transport type establishes a *queue pair (QP)* between the two communicating parties. QPs are bi-directional message transport engines used to send and receive data in InfiniBand. Endpoints of a single RC QP can only communicate with each other but not with any other QP in the same or any other target adapter. Each QP endpoint has a queue pair number (QPN) assigned by the RNIC which uniquely identifies the QP within the RNIC.

The RC transport uses several techniques to ensure reliability. The target must respond to each request packet with a positive acknowledge packet or a negative acknowledge packet. The acknowledgement-based protocol permits the requester to verify that all packets are delivered to the target. To ensure the integrity of a packet, each packet contains two checksums that are verified by the target node. Finally, the RNIC counts received and sent packets using a packet sequence number (PSN), which is included in each packet. Thus, endpoints of a QP must know the PSN of each other to enforce in-order delivery and detect duplicate and lost packets.

6.2.1 IBA Memory Protection

The IBA protection mechanisms provide protection from unauthorized access to the local memory by network controllers. The local memory can also be protected against prohibited memory accesses. Three mechanisms exist to enforce memory access restrictions: Memory Regions, Protection Domains (PD), and Memory Windows.

Memory Regions. To get access to host memory, the RNIC must first allocate a corresponding memory region. This process involves copying page table entries of the corresponding memory to the memory management unit of the RNIC. When a memory region is created, the RNIC generates keys for local and remote accesses, namely *l_key* and *r_key*. The memory region can be accessed by any local QP which has the *l_key* as long as they are in the same PD, and by their remote QP endpoints which have the *r_key*. The endpoints must prove the possession of this key by including it in every RDMA request, such as RDMA Write and Read. *r_key* is not used in any form of cryptographic computation, but rather is used as access tokens that are transmitted in plaintext.

Protection Domain. PDs provide protection from unauthorized or inadvertent use of memory regions. PDs group IB resources such as QP connections and memory regions

that can work together: QP connections created in one PD cannot access memory regions allocated in another PD. In other words, a memory region can be accessed by any QP from its PD. All QPs and memory regions must have a PD and can be a member of one PD only.

Memory Windows. Memory windows extend protection of memory regions by allowing remote QPs to have different access rights within a memory region and grant access to only a slice of the memory region.

6.3 Problem Definition

This section describes the adversary model we consider, outlines different types of attacks, and the security properties we strive to achieve.

6.3.1 Desired Security Properties

The current IBA protection mechanisms do not suffice to ensure secure communication between endpoints, allowing adversaries numerous attacks. Thus, the primary goal of our work is to *secure RDMA protocols against attacks* by providing source and data authentication along with data secrecy and data freshness. Source authentication denotes the verification of the source address of a host that sends a packet and is designed to determine whether a packet originated from the claimed source. Data authentication ensures that the packet content has not been modified. Data secrecy ensures that the data remains hidden from a network eavesdropper. Data freshness ensures that data has not been recorded and replayed by a network attacker. Additionally, our proposal should require *minimal changes to the protocol*, and introduce only a *minor performance overhead*. This does not only include latency and processing overhead for RDMA requests but also memory state overhead on the RNIC.

6.3.2 Adversary Model

In our adversary model we consider end hosts that are equipped with RNICs and interact with each other through RDMA, and an adversary with the following parameters.

Location. We assume that the adversary can reside at *arbitrary locations* within the network. Thus, we consider both network-based adversaries (e.g., rogue cloud provider, rogue administrator, malicious bump-in-the-wire device) and adversaries located at end hosts (e.g., compromise of an end host). This includes compromise of the machines of communicating parties. However, we assume that RNICs are *trusted* by their host. This could be achieved using remote attestation, whereby a trusted party checks the internal state of a potentially compromised network device. We further assume that the internal bus is trusted, such that the CPU can securely communicate with the RNIC.

Capabilities. A network-based adversary can passively eavesdrop on messages, but also actively tamper with the communication. Since RDMA communication is performed in plaintext, an adversary that is located on the path between communicating parties can obtain any information in all IB and Ethernet headers. Furthermore, he can also alter any of these values, as this only requires recalculation of packet checksums, whose algorithms and seeds are known and specified by the IBA.

Given these capabilities, the adversary can also fabricate packets and send them towards a destination of its choice using spoofed QP numbers, *r_key*s, and PSNs (e.g., to modify a memory region without authorization to influence the behavior of applications running on the remote host).

Cryptography. The adversary has no efficient way of breaking cryptographic primitives. For pseudorandom function families, this means that no efficient algorithm can distinguish between an output of a function chosen randomly from the pseudorandom function family (PRF) and a random value.

6.4 Secure RDMA System Design

We propose a new transport type for reliable communication based on the IBA. We introduce a secure reliable connection (SRC) QP that uses symmetric cryptography for source and data authentication, and thus provides guarantees for the origin of a packet, data authenticity and payload secrecy.

To require minimal changes to the current IBA specification, our proposed design of the SRC QP consists of two main changes: 1) we add symmetric key initialization for QPs, and 2) we propose a new packet header called secure transport header (STH) which contains a message authentication code (MAC) providing integrity of the packet content. The STH

Table 6.1: Notation used in this chapter.

$\|\|$	Bitstring concatenation
$PRF_K(\cdot)$	Pseudorandom function using key K
$MAC_K(\cdot)$	Message authentication code using key K
A, B	Endpoints uniquely identified by the combination of the adapter port address (APA) and Queue Pair Numbers (QPN)
$K_{A,B}$	Symmetric key shared between node A and B
$nonce_{A\rightarrow B}$	cryptographic nonce used for communication in the direction from node A to node B
K_{PD}, K_{MR}, K_{SR}	Symmetric key used for protection domain, memory region, or sub memory region

must be included in all requests and response packets corresponding to RDMA reads and writes.

Besides basic QP-channel protection, we also propose *PD level protection* eliminating the need for storing cryptographic keys for each QP, which drastically reduces the memory overhead on RNICs. Additionally, it enables *extended memory protection* that provides memory access control based on encryption. All QPs and memory regions created in a secure PD will be inherently secured by it.

Table 6.1 lists the security-related notation used in this paper.

6.4.1 Assumptions

Trust in RNIC. We assume that the RNIC is trusted by its host. It can not only perform authentication of outgoing packets, but is also trusted to perform en-/decryption of the packet payload. We further assume that the internal bus is trusted, such that the CPU can securely communicate with the RNIC.

QP-level Key Establishment. Our system enables the establishment of a QP-level symmetric key. To guarantee interoperability, our design is agnostic of this underlying mechanism. IBA could use for instance a (D)TLS [127] or QUIC [64] handshake as a mechanism to obtain a QP-level symmetric key.

Key Validity. As the validity period of a QP-level symmetric key is bound to the

Size (bits)	0	96	128	160	224	256	384	512
Value	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7

Table 6.2: Possible sizes of STH, depending on the 3 bit value indicated in the Base Transport Header of an IB packet.

lifetime of a QP, key rollover can be performed by closing and reopening a QP between the communicating entities. Thus, key lifetime can be managed on the application level.

6.4.2 Secure Reliable Connection Queue Pair

We propose new transport type—Secure Reliable Connection (SRC) QP—that uses symmetric cryptography for source and data authentication. The introduction of SRC requires minimal changes to the current specification. Specifically, the QP initialization requires specifying a protection algorithm and a symmetric key. This allows us to bootstrap secrecy and authentication for QP-based communication.

Secure Transport Header. Secure Transport Header (STH) consists of MAC to provide header and packet authentication. The STH must be included in all request and response packets of sRDMA. Depending on the authentication mode installed to the secure QP, the MAC either authenticates only the packet header or the entire packet. To specify the length of the STH, we use 3 (out of 7) reserved invariant bits in the Base Transport Header. Based on the 3 bit value (see Table 6.2), the size of the MAC changes: minimum 96 bits, and maximum 512 bits. If the reserved 3 bits are all zero, then the STH is not present in the packet, thereby enabling support of both classical and secure QP connections.

Reusing PSN as a Per-Packet Nonce. sRDMA prevents replay attacks by including a unique nonce in the MAC computation of each packet (Section 6.4.3). Nonces are used as initialization vectors for ciphers to ensure that every packet is unique. They must *only be used once*, but their choice can be *predictable* and they can be *transmitted in clear* [76, 132]. In case a nonce is reused, the cryptographic properties of a cipher are affected (e.g., “sudden death” property of Poly1305 [19]).

A naive solution is to transmit a nonce as cleartext with each packet (e.g., as in TLS up to version 1.2 [129]). However, this would incur an additional transmission overhead of at least 64 bits, and additional 64 bits memory overhead on RNICs memory to store the nonce.

To avoid the overhead of transmitting the nonce, our protocol takes advantage of the sequential nature of IB packets. It uses the sequence numbers as nonces as they are tracked by end points and can never be reused. The approach resembles how TLS 1.3 [128] exploits the packet number as a nonce; however, the size of the PSN in the IB packet is only 24 bits, which would cause a reuse of a nonce after 80 ms assuming that an RNIC is able to send 200 million packets per second [95].

sRDMA extends the local PSN counters for inbound and outbound packets on the RNIC to 64 bits each, and reuse them as a per-packet nonce, thereby introducing only 40 bits overhead for each nonce. However, the size of the PSN transmitted on the wire remains unchanged (24 bits) and contains the least significant bits of the 64 bit counter. sRDMA is able to infer the 64 bit nonce used to secure the packet using only the 24 bit PSN specified in the header. Under the same assumption on the packet rate, the reuse of nonce occurs after 3,000 years.

To ensure that the nonce never gets reused by both endpoints, we use the most significant bit to identify the direction of communication between the entities A and B using their endpoint identifiers: the combination of adapter port address and Queue Pair Number (QPN).

6.4.3 Header Authentication

To perform header authentication, sRDMA uses the established symmetric keys and calculates a MAC for each packet:

$$mac_{hdr} = MAC_{K_{A,B}}(nonce_{A \to B} \parallel RH \parallel BTH)$$

Here, RH denotes the routing header, which defines the adapter port address, and BTH the base transport header, which includes destination QPN. Note that these headers uniquely identify the sender and receiver RNIC, and limit the input size of the MAC computation (only the packet header instead of the entire packet with arbitrary payload length). Thus, assuming an block-cipher-based MAC is used, a fixed number invocations of the block-cipher are required to calculate a MAC.

The RNIC of the receiving node will recompute the MAC for each packet and compare it to the MAC appended in the STH. Fields that are modified during the packet's transmission are replaced with ones during the MAC computation (same as for invariant checksum).

Table 6.3: Overheads of sRDMA for N RC QP connections with AES-128 cipher in 4 different protection modes. Here, *pd-prot* and *ext-mem* stand for PD-level protection and extended memory protection, and are described in Section 6.4.5 and 6.4.6.

AES-128 protection	Key overhead	Nonce	Header
basic	16B * N	10B * N	16B
pd-prot	16B	10B * N	16B
ext-mem	16B * N + 16B	10B * N	16B
pd-prot + *ext-mem*	16B + 16B	10B * N	16B

Header authentication prevents not only source-address-spoofing attacks, but also unauthorized access to memory regions by augmenting the existing IBA memory-protection mechanisms (i.e., *r_key* and memory windows).

6.4.4 Packet Authentication and Encryption

For packet authentication and payload encryption, we assume that the RNIC is trusted. Thus, the host is allowed to offload all cryptographic operations to the RNIC. We use authenticated encryption with associated data (AEAD), to simultaneously obtain secrecy and authenticity for the payload. The authentication tag is transmitted using the MAC field in the STH.

6.4.5 PD-level Protection

Introducing QP-level keys requires storing a 16 byte key per QP (see Table 6.3). As an RNIC might have a large number of QPs simultaneously, this can lead to a significant memory overhead on the RNIC. Memory on RNICs is a constrained resource, and a large part is consumed by IB connection contexts and page-table entries for registered memory. Multiple works report significant performance degradation of RDMA operations when the amount of memory registered or the number of QPs is increased [71, 47]. This is due to the RNIC running out of memory for storing page-table entries and starting to fetch them from system memory across the PCI bus. For instance, Dragojevic et al. [47] observe 4x throughput drop in their evaluation when 4,096 memory pages are registered within

the RNIC compared to a single-page experiment. Thus, we aim to mitigate the memory overheads introduced by QP-level keys.

To reduce the memory overhead and eliminate the need of storing a symmetric key per QP, we introduce PD-level protection. In this mechanism, we assign a symmetric key K_{PD} to each protection domain PD and use this key to derive QP-level keys using efficient key derivation [61]. PD-level keys are exchanged using the same mechanism as QP-level keys (see Section 6.4.1). The derivation process works as follows:

$$K_{A,B} = PRF_{K_{PD}}(APA_A \parallel QPN_A \parallel APA_B \parallel QPN_B)$$

PRF denotes a pseudorandom function with a PD-level key K_{PD} and a pair of unique end point identifiers (i.e., adapter port address (APA) and queue pair number (QPN)) as input. When an RDMA request targets a QP that is located within a protection domain PD, the RNIC uses the corresponding symmetric key K_{PD} to derive the QP-level key on-the-fly. The QP-level key is then used to perform authentication and encryption. Thus, instead of storing a symmetric key per QP, the RNIC is only required to store a key per PD. To minimize the processing overhead, the RNIC can cache the derived QP-level keys (e.g., after the first packet of a message arrives). K_{PD} is initialized upon creation of the PD and thus the lifetime of K_{PD} is bound to the lifetime of the PD. In order to perform a key rollover, a new protection domain must be created.

6.4.6 Extended Memory Protection

Using encryption of memory regions enables an even stronger mechanism for access control, as only entities in possession of the required key are able to read the content of a memory region. For this purpose, we use PD-level memory protection and derive memory level keys from K_{PD} for memory regions that are created within the protection domain. The derivation process works as follows:

$$K_{MR} = PRF_{K_{PD}}(START_{MR} \parallel END_{MR} \parallel r_key_{MR})$$

Alternatively, the K_{MR} can be provided by the application to protect memory from unauthorized accesses.

When remote parties want to access a subregion (SR) of the region MR, they need to prove the possession of the K_{MR} by computing a key to the SR:

$$K_{SR} = PRF_{K_{MR}}(START_{SR} \parallel END_{SR}) \tag{6.1}$$

Nonce for Key Derivation. To avoid replay attacks, our system must use a separate nonce for each memory region. However, it is not possible to use a memory access counter as nonce, as multiple QPs can access the same memory region. Therefore, this would require the RNIC to include a random nonce in each packet, which must be unique among all nonces used to access the memory region. Given that multiple parties have access to the region, this property is hard to achieve. Additionally, we want to avoid transferring a separate MAC for memory access in the packet header. Thus, we suggest to reuse the MAC of the header by overwriting it as follows:

$$mac_{hdr} = MAC_{K_{A,B}}(K_{SR}||mac_{hdr})$$

Such design allows sRDMA to reuse the per-packet nonce used in computation of mac_{hdr} and ensure the possession of K_{MR} to access memory. This construction is secure since the key is unknown to an adversary.

6.5 Implementation

Towards our goal of supporting secure QP connections, this section describes how we implement the sRDMA protocol using modern programmable network adapters equipped with ARM multi-core processors [24, 153]. sRDMA core primitives are implemented in 3,500 lines of C++ code and rely on various libraries: libibverbs, an implementation of the RDMA verbs; librdmacm, an implementation of the RDMA connection manager; Openssl 1.1.1a, a general-purpose cryptography library; and libev, a high-performance event loop. Our implementation supports more than 20 different cryptographic algorithms, such as the AES cipher and SHA hash families, to enable authentication and data secrecy for secure QPs. The implementation, all tests, and benchmark scripts are available in the open-source release.

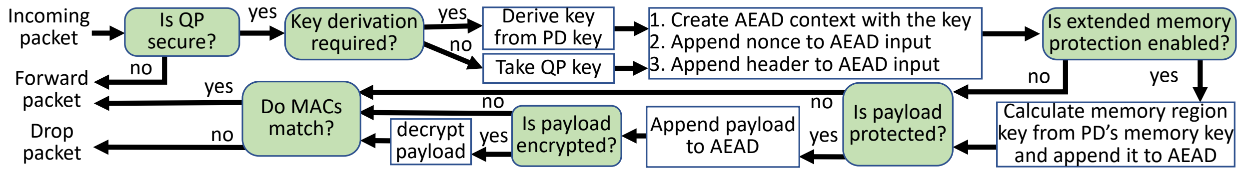


Figure 6.1: RNIC packet processing on receive.

6.5.1 Notation and Experimental Setup

In the rest of the paper, we refer to a programmable network adapter as a *SmartNIC*. Our SmartNIC is capable of running a full Linux stack, supports RDMA over RoCEv2 and has crypto acceleration enabled. When RDMA requests are initiated on the SmartNIC and target the local host we refer to them as DMA requests as they only pass across the PCIe bus.

Our implementation is bi-directional, i.e., sRDMA writes and reads can be sent in both directions passing through both SmartNICs of the initiator and the target. Therefore, we distinguish between three roles as depicted in Figure 6.2: *Initiator*, *SmartNIC*, and *Target*, and the initiator always communicates with the target via two SmartNICs. Such a design allows full offloading of cryptographic computations from the initiator and the target to their respective SmartNICs.

6.5.2 Implementation of the Secure QP

We provide a library that models a secure QP connection between an initiator and a target as three standard RC QPs: one DMA connection between the SmartNIC and the host on *each* endpoint, and one connection between the SmartNICs.

Connection Establishment. Our secure QP library encapsulates connection establishment, which is performed in three stages as for a classical RC QP. When an application wants to establish a secure QP, it first creates a local QP in the INIT state. In this state, the connection between the host and the SmartNIC is created, and all necessary symmetric keys are copied to the SmartNIC. Then the QP must be transitioned to the RTR state by passing information about the target such as the QPN, the LID, and the PSN. To perform this transition we establish an RC QP connection between the two SmartNICs and create a special connection context on each SmartNIC. Finally, to send messages we transit the secure QP to the RTS state by passing the local send PSN. The application workers

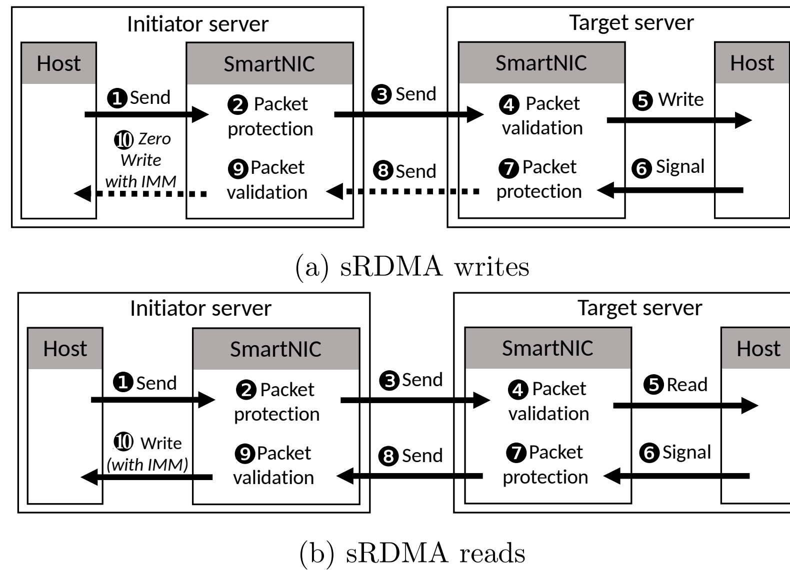
on the SmartNIC are responsible for packet counting, key derivation, and cryptographic algorithms.

Memory Registration. When a memory region should be secured with extended memory protection, the library intercepts a memory registration request and sends memory region information to a thread on the local SmartNIC.

Secure QP communication. The initiator uses IB Send to deliver packets for both sRDMA reads and writes to its SmartNIC. The SmartNIC uses IB Receive to receive incoming packets from DMA connections and from remote SmartNICs. The SmartNIC secures all incoming packets from a DMA connection according to the cryptographic mechanism agreed on with the target. To secure a packet, the SmartNIC appends the IB transport and RDMA headers along with the generated MAC to the packet header. In our implementation, we use IB scatter/gather entries to attach an additional header before the main payload provided by the initiator. Scatter/gather entries allow building up an outgoing message from multiple buffers. After that, the packets are forwarded to the SmartNIC of the target QP. The target's SmartNIC verifies the security header as depicted in Figure 6.1, and decides on initiating an RDMA Read or RDMA write depending on the type of the request towards the target's host. The replies from the target are secured by its SmartNIC and forwarded back to the initiator.

sRDMA request completion. If the initiator expects an acknowledgment for a signaled request, the SmartNIC is responsible for acknowledging the initiator about the completion of the request. We use IB requests with immediate data to generate completion events on the host. The secure QP library is able to intercept completion events to distinguish between classical IB completions and sRDMA completions. The intercepted sRDMA completions are modified to inform the initiator about the sRDMA completion instead of the classical IB completion.

Packet security. The whole process of packet verification and key generation is shown in Figure 6.1. The SmartNIC performs header authentication, packet authentication, or payload encryption depending on which security protocol has been set up for the QP and which packet is processed. The SmartNIC will derive the QP's key if the QP is initialized in a secure PD, and also verify extended memory protection if the registered memory region has extended memory protection set up. On receiving, the SmartNIC also checks whether the QP is indeed a secure QP, as our implementation also supports classical insecure RC QPs. For insecure RC QPs, packets do not carry a MAC and are always trusted by SmartNICs.



(a) sRDMA writes

(b) sRDMA reads

Figure 6.2: Implementation of RDMA operations.

6.5.3 sRDMA requests

Figure 6.2a depicts the implementation of an sRDMA write. The initiator ❶ sends a packet to the local SmartNIC containing the payload and the remote memory address. The local SmartNIC ❷ appends the IB header and the STH and ❸ sends the secured packet to the remote SmartNIC. The remote SmartNIC ❹ processes the header and ❺ initiates a signaled DMA write to the host memory specified in the header. Upon ❻ the completion of the DMA write, the SmartNIC, depending on whether the sRDMA write is signaled, ❼❽ sends an authenticated Ack packet to the initiator's SmartNIC. The SmartNIC of the initiator then ❾ verifies the packet and ❿ performs an empty RDMA write with immediate data to its host, which consumes one posted receive at the host application. Finally, the secure QP interface intercepts such completions and modifies them to notify the application about the secure request completion.

sRDMA also implements secure Send operations which are similar to sRDMA writes, but they always generate the completion on the target and do not require knowing destination buffers. Since a Send request does not contain the header with destination buffer, it does not support memory protection.

sRDMA read has a similar structure as an sRDMA write as depicted in Figure 6.2b but there are some subtle differences. The initiator ❶ sends the message containing remote and local memory addresses and their *r_key*s to the local SmartNIC. The initiator's SmartNIC

creates a special local read completion context with the initiator's memory address where the remote data must be copied to. Then the local SmartNIC ❷❸ sends the authenticated read request to the remote SmartNIC, which ❹ verifies the request and ❺ initiates a signaled DMA read from the target host memory to one of the SmartNIC's buffers. When ❻ the completion of a DMA read is generated, the SmartNIC ❼❽ sends an authenticated read response with read data to the initiator's SmartNIC. The initiator's SmartNIC ❾ verifies the MAC of response packets and decides whether to ❿ write their content to the memory address specified in the matched local read completion context using a DMA write request. The DMA write will be with immediate data if the sRDMA read is signaled.

6.6 Evaluation

We conduct a series of benchmarks to thoroughly profile our system. To evaluate the overall sRDMA performance and the impact of cryptographic operations, we first evaluate the performance of each cryptographic algorithm. Secondly, we evaluate the latency and bandwidth of sRDMA writes and reads to assess the overheads of secure QPs over insecure QPs. Subsequently, we study the impact of bulk sRDMA operations by measuring the achievable bandwidth for different read/write ratios. Later, we evaluate the performance of the HERD key-value store [70] to examine the impact of sRDMA.

Test settings. The experiments are conducted on two servers directly connected to each other using the RoCEv2 protocol. These servers run Ubuntu 18.04.1 LTS with a 4.15.0-43-generic Linux kernel. Each server is equipped with a Broadcom PS225 25 Gbit/s programmable network controller. Both network adapters have eight-core 64-bit ARM Cortex-A72 3.0 GHz processors and 8 GiB of dual-channel DDR4 DRAM.

6.6.1 Authentication performance

We first study the performance of the cryptographic engine installed in the SmartNICs. We evaluate 7 different cryptographic algorithms of the openssl 1.1.1a library for message authentication: *aes-128*, *aes-192*, *aes-256*, *chacha20-poly1305*, *sha1-160*, *sha2-256*, *sha2-512*.

Figure 6.3 depicts the achievable throughput in Gbit/s of those algorithms for different numbers of threads and block sizes. The line rate of the tested RNIC over the RoCEv2

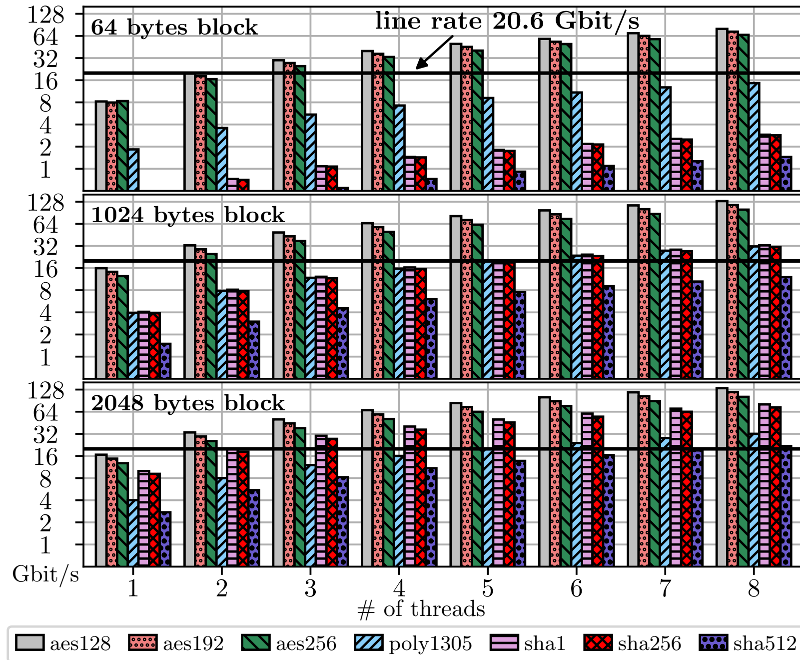


Figure 6.3: Authentication performance using openssl.

protocol is 20.6 Gbit/s, which is goodput of 25 Gbit/s link. AES algorithms are the fastest for small blocks and achieve 8 Gbit/s for 64 byte blocks using a single thread. Thus, our sRDMA library uses the AES128 algorithm as the PRF function needed for key derivation. For larger blocks hash-based methods perform almost as fast as cipher-based algorithms. We observe that chacha20-poly1305 is 4x slower on average than the AES algorithms. The data also reveals that we cannot achieve the line rate for packet authentication with SHA512.

For varying key sizes of AES algorithms, we have not noticed significant differences in performance and hereafter report results exclusively for the AES128 algorithm. As SHA1-based authentication provides similar performance as SHA256 in all tests, we omit its data in all plots. Additionally, we label chacha20-poly1305 in Figures as *poly1305*.

6.6.2 Evaluation modes

All evaluations have been performed with no security enabled (*NO security*) and in four protection modes:

No security. In *No security* mode RDMA reads and writes are performed as described in Section 6.5.3 but with skipping packet protection and validation (❷❹❼❾ in Figure 6.2).

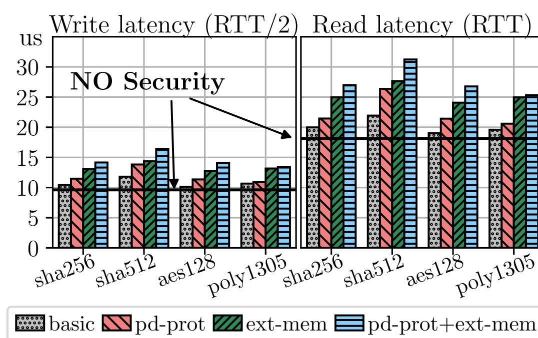


Figure 6.4: Source authentication latency of reads and writes carrying 32 Bytes payload.

Basic mode. In *basic* mode the key is attached to the secure QP connection directly, so the key is in the RNIC's cache when an incoming packet must be processed.

PD-prot mode. The secure QP is created without an individual key, but in the secure PD (*pd-prot*) with a key derivation algorithm. We consider the case when the RNIC does not cache derived keys, and therefore, every time a packet arrives, the RNIC must derive the QP key from the PD key. In these experiments we want to show the performance of the system with constant cache misses. Using the cache we expect the same performance as in basic mode without key derivation.

Ext-mem mode. In this mode, the QP is created with an individual key and with extended memory protection (*ext-mem*) enabled. Extended memory protection requires derivation of memory level keys from a PD-level key. In this case, when a packet arrives, the RNIC must generate a key to access memory specified in the RDMA header from the PD-level key and include the generated key in MAC calculation.

PD-prot + ext-mem mode. The last mode combines our two protection methods: secure PD and extended memory protection (*pd-prot + ext-mem*). Therefore, the RNIC is responsible for generating both keys when a packet should be processed.

6.6.3 Latency

To evaluate the overall sRDMA performance and the impact of cryptographic operations, we split latency tests in two categories: header authentication only and full packet security.

Header authentication. Figure 6.4 presents the median latency of sRDMA reads and writes in all four protection modes for header authentication. The figure reports the median only as for all measurements deviation from the median is less than 0.4 μs. All

measurements are done for packets carrying the payload of 32 bytes. sRDMA write latency is measured for a half round trip, whereas sRDMA reads are for a full round trip. The latency of sRDMA writes without security is 9.55 μs and of sRDMA reads is 18.2 μs which build the baseline for our experiments.

Figure 6.4 shows that all tested security algorithms in the first mode add about 0.9 μs for sRDMA writes which is approximately 9% more than the insecure version. Another interesting observation is that the QP key derivation is more expensive than memory key derivation. The difference stems from the fact that a key-derivation process involves reinitialization of cryptographic contexts and different algorithms have different reinitialization performance (e.g., AES generates round keys [41]). The same phenomenon occurs for sRDMA reads. As expected, the highest latency is achieved for sRDMA operations with both key derivation and extended memory protection.

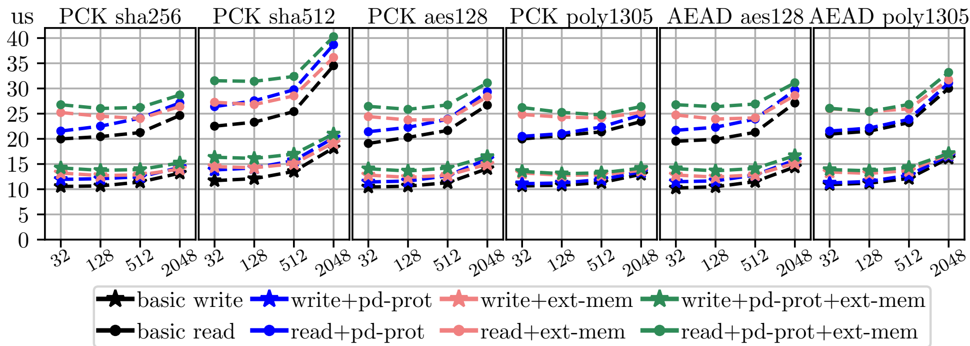


Figure 6.5: Latency of packet authentication (PCK) and encryption ($AEAD$) as a function of payload sizes.

Packet security. We evaluate the latency of packet authentication (PCK) and packet encryption ($AEAD$) for different payload sizes and in four protection modes. Figure 6.5 illustrates the median latency of sRDMA reads and writes for SHA256, SHA512, AES128, and POLY1305. In each subplot, the top four lines illustrate sRDMA read round-trip latency, and the bottom four lines half-round-trip latency of sRDMA writes.

Figure 6.5 highlights that payload authentication is more expensive than header authentication. It takes 15 μs to write and secure 2 KiB payload in the first mode in comparison to header authentication of the same packet with the median of 12 μs. The graph also illustrates that latency increases for both reads and writes with payload size as more data must be authenticated. For AEAD, latency goes up even faster with respect to payload size since more data is en-/decrypted. As anticipated, SHA512 has the highest latency as

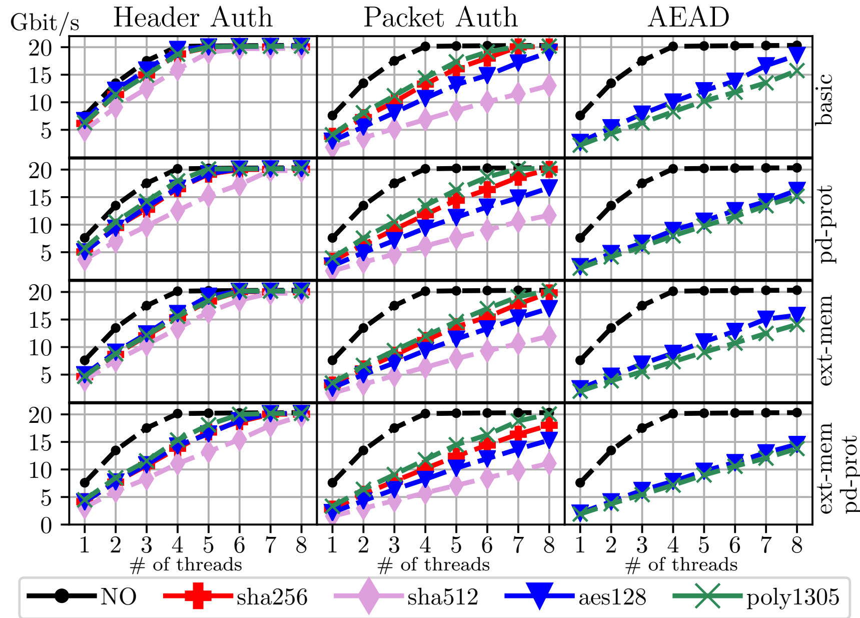


Figure 6.6: Bandwidth of sRDMA Writes in four different protection modes, and with *NO* security enabled.

the most expensive algorithm. We observe that for smaller payload sizes payload authentication and payload encryption achieves approximately the same performance in terms of latency.

6.6.4 Bandwidth

We measure performance separately for sRDMA reads and writes. As for latency benchmarks, all evaluations are performed in four protection modes. Our implementation is multi-threaded where each thread can process requests from a single secure QP. The number of threads represents the number of connections between endpoints. For n threads, each host establishes n secure connections with its SmartNIC, and SmartNICs establish n connections between each other. Thread workers on a SmartNIC do not share any resources and are pinned to distinct cores. In all evaluations the initiator issues requests continuously to the target, but with a limited number of outstanding requests (96 per connection). Once the initiator receives the signal for an sRDMA request completion it posts new requests to maintain 96 outstanding requests. The payload size is 2,048 bytes and bandwidth is measured in Gbit/s of goodput. We also assume the worst case scenario for the secure PD mode (*pd-prot*): the RNIC derives the QP key from the PD key for each

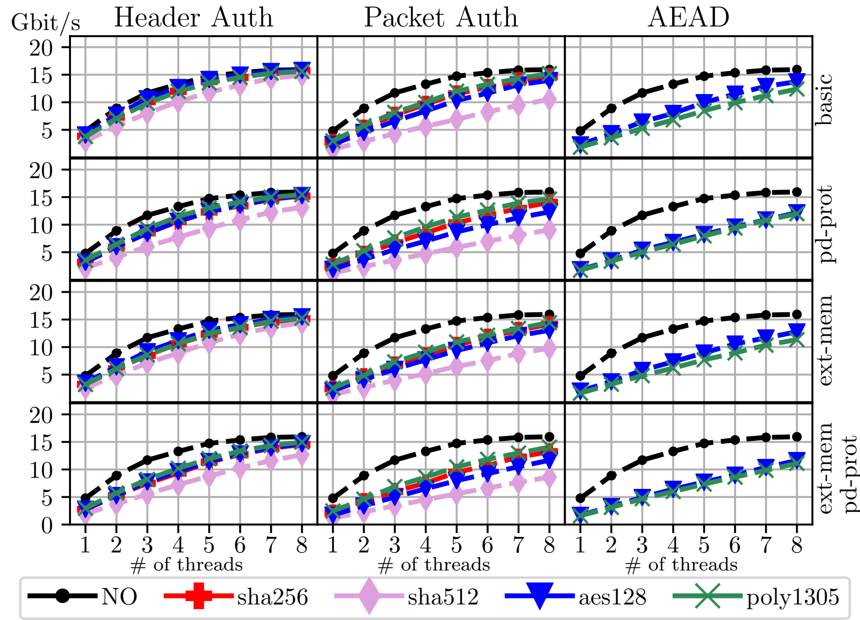


Figure 6.7: Bandwidth of sRDMA Reads in four different protection modes, and with *NO* security enabled.

packet. In other words, we consider the case when the RNIC does not cache derived keys. The main reason for that is that *pd-prot* mode with caching has the same performance as *basic* mode.

Figure 6.6 depicts communication bandwidth for sRDMA writes with different cryptographic algorithms. The black line (*NO*) in the header column stands for sRDMA writes with no security enabled. We observe that the single-threaded test with no security achieves only 8 Gbit/s while the highest RDMA goodput bandwidth achievable on our interconnect is 20.6 Gbit/s. The slowdown is caused by processing and parsing headers of messages by the general purpose ARM CPUs of the SmartNICs. Even if no security is enabled, a thread worker reads and parses headers of incoming packets and, depending on the operation code, initiates RDMA requests according to our implementation described in Section 6.5.3. In our tests we treat performance of secure operations with no security as the baseline. The highest achievable goodput bandwidth with no security is 20.5 Gbit/s which is line rate.

Figure 6.6 illustrates that sRDMA writes with header authentication can achieve line rate in all four protection modes if we use all 8 threads. The slowest header authentication is observed for SHA512 due to hashing performance. For full packet authentication SHA512 reaches only a goodput of 13 Gbit/s which is even slower than AEAD algorithms. In the

payload encryption mode, our implementation can also achieve line rate for the SHA256 and POLY1305 algorithms. AES128 based authentication achieves 19.6 Gbit/s which is 95% of the line rate. The data also demonstrates that key derivation algorithms slow down sRDMA writes by 2 Gbit/s on average. However, in header authentication mode all algorithms can achieve 20 Gbit/s without performance loss when all 8 threads are used. Another interesting observation is that POLY1305 is faster than AES128 in packet-authentication mode, but slower in packet-encryption one. In AEAD mode, the highest write bandwidth of 19 Gbit/s is observed for the AES128 algorithm.

We have performed a similar benchmark for sRDMA reads in various protection modes. Results of our evaluations are depicted in Figure 6.7. Again, the black line (*NO*) stands for no security installed and represents the baseline for sRDMA reads. sRDMA reads are more expensive than writes despite the fact that they transfer the same amount of protected bytes as signaled sRDMA writes. Both sRDMA operations require six hops for a full round trip, and they both transfer the same payload size but in different directions. For writes, data is sent from the initiator to the target, and for reads from the target to the initiator. The differences in performance stem from the fact that an sRDMA read is a more complex operation than an sRDMA write and requires to create a special read context and matching it at initiator's SmartNIC (see Section 6.5.3). In addition, receive buffers on SmartNICs for reads and writes have different lifetimes. For example, a receive buffer can be released on the target SmartNIC once the completion of the RDMA write is received (❻ in Figure 6.2a), however, for reads the buffer on the target SmartNIC can be released once the completion of ❽ is received from Figure 6.2b. According to the data, the highest achievable sRDMA read bandwidth is 16.71 GBit/s for 8 threads and about 4.7 Gbit/s for single-threaded test. Overall, our measurements indicate that reads are 16% slower than writes for all tests due to the complexity of sRDMA reads.

CPU Usage in Bandwidth Experiments. In our experiments, sRDMA introduces no overhead on the host CPU usage as packet processing is fully offloaded to the SmartNIC. The host application only needs to submit an RDMA request to the SmartNIC, which performs all cryptographic computations as described in Section 6.5.3. The SmartNIC, on the other hand, has full CPU usage in almost all experiments, which can be observed in the inability of the majority of security schemes to achieve line rate. The main reason for that is the SmartNIC needs to load the incoming packets from its DRAM to the L1 cache of its CPU cores in order to process the packets depending on the installed security level. Thus, all protection levels which require the CPU to read the whole packet have 800% CPU

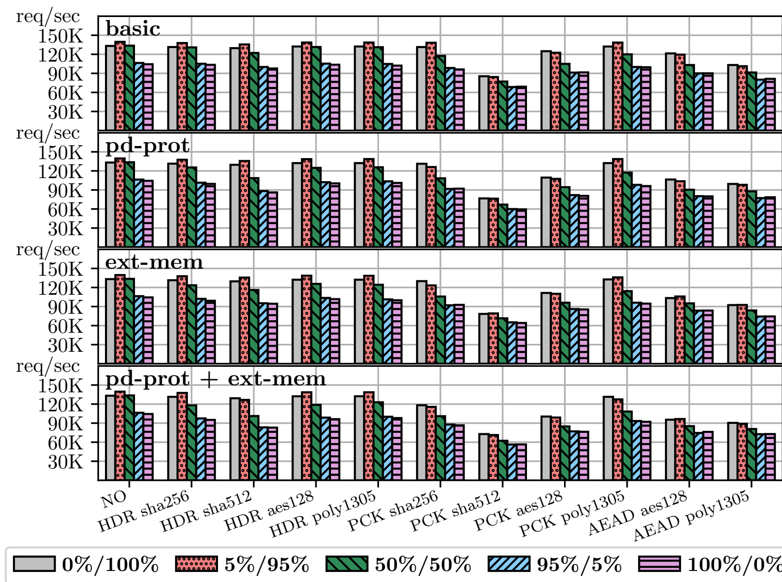


Figure 6.8: Throughput of mixed read/write benchmark.

usage for 8 worker threads, even though in authentication performance experiment (see Figure 6.3) all authentication algorithms achieves the line rate for 2 KiB blocks. It comes from the fact that the packet authentication and AEAD are memory-bound problems, and, therefore, CPU works at full capacity to copy the data to its caches.

Header authentication requires reading only the header to authenticate the packet. Thus, header authentication algorithms could achieve 100% of line-rate, although, the performance still suffers from cache misses. The lowest CPU usage is observed for AES128 authentication scheme, which is 440% CPU usage for the bandwidth experiment with sRDMA Write requests. The sRDMA reads, on other hand, consume almost 750% on the target SmartNIC.

6.6.5 Mixed write/read workload

The results of Figure 6.6 and Figure 6.7 are valid for either read-only or write-only workloads, which are uncommon for read-world applications. Therefore, we measure the throughput of sRDMA in a more realistic scenario as used in key-value stores that exploit one-sided RDMA operations. Figure 6.8 shows the throughput for workloads with different (read/write) ratios, including write only (0%/100%), write mostly (5%/95%), equal-shares (50%/50%), read-mostly (95%/5%) and read-only (100%/0%). The read-heavy workload

is representative for applications such as photo tagging. The update-heavy workload is typical for applications such as an advertisement log that records recent user activities. In this benchmark the payload size is 2,048 bytes, and sRDMA is deployed with all 8 workers. We also consider the worst case scenario for the secure PD mode (*pd-prot*), when the RNIC derives the QP key for each packet. The *pd-prot* mode with QP key caching has the same performance as *basic* mode.

Figure 6.8 illustrates that (5%/95%) workload performs better than (0%/100%) one. The reason for that is better utilization of the bi-directional connection between endpoints since sRDMA writes send data from the initiator to the target, whereas sRDMA reads from the target to the initiator. Therefore, in that case we achieve a better utilization of the connection in the direction of the initiator. In theory, a (50%/50%) ratio would lead to the highest throughput as both links would be loaded evenly; however the lower performance of sRDMA reads overwhelms benefits of the network utilization. For the same reason, the throughput decreases for higher read ratios.

6.6.6 Key-value store workload

HERD [70] is an RDMA-accelerated key-value store which uses a mix of RDMA write and IB send verbs. HERD uses MICA's [92] algorithm for both GETs and PUTs: each GET requires up to two random memory lookups, and each PUT requires one. In HERD, clients transmit their request to the server's memory using RDMA writes, and get responses via unreliable datagram QPs. To comply with our sRDMA design, we made some changes to the original HERD implementation. First of all, we replace all unreliable datagram QPs with RC QPs as they are not reliable and not point-to-point and thus incompatible with sRDMA. That is, the server replies to clients via RC QPs, but still uses IB Send verbs. For that, we also implement secure SEND operations which are similar to sRDMA writes, but they always generate the completion on the target and do not require knowing destination buffers. Since an IB Send request does not contain the header with destination buffer, it does not support extended memory protection. The second change is that clients send requests via reliable sRDMA writes instead of unreliable writes.

Key-value-store experiments use one server machine and one client machine. The server machine has only one worker process when the client machine has 8 processes. Each client process establishes an sRDMA connection to the server. The key size is 16 bytes and the value size is 32 bytes. Therefore, clients send and receive small messages of less than

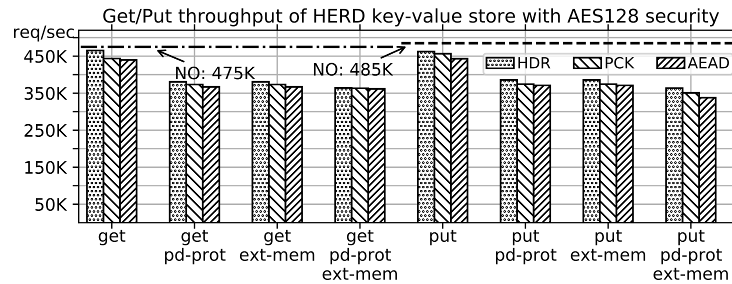


Figure 6.9: Throughput of the HERD key-value store over sRDMA.

40 bytes. The key-value store contains 8,388,608 keys and occupies 1 GiB of memory. Figure 6.9 depicts the throughput for puts and gets in different protections modes based on the AES128 cipher. We also measure HERD's throughput with NO protection which is 475K requests/sec for gets and 492K requests/sec for puts. Puts are faster than gets because they cause fewer lookups in internal memory structures.

According to the data in Figure 6.9, basic packet authentication without key-derivation algorithms achieves almost the same throughput as the unprotected version. Interestingly, even the AEAD mode decreases the throughput by 7.3%. In the setting with a secure PD when the key must be generated for each request, we observe a 21% slow down in both puts and gets. It is worth mentioning that we intentionally derive the QP keys for each request in the secure PD mode (*pd-prot*) to see the effect of constant misses in QP keys. In real settings, an RNIC would have a cache with generated keys to reduce computation. In such case, the *pd-prot* mode has the same performance as the *basic* mode.

The *ext-mem* achieves the same performance as the *pd-prot* case as they both need to derive one key. *pd-prot + ext-mem* has a slightly lower throughput as more keys have to be derived. Note that the caching of memory keys and the connection keys would alleviate the derivation penalty, making them as efficient as the *basic* mode.

6.7 Related Work on Securing IBA

RFC 5042 [118] analyzes the security issues around uses of RDMA protocols. It reviews various attacks against resources including spoofing, tampering, information disclosure, and DoS. As a countermeasure the authors suggest to employ IPsec authentication and encryption [45]. However, IPSec currently does not support RDMA traffic, because it is unaware of the encapsulated RDMA headers and thus cannot distinguish QP endpoints.

Table 6.4: Comp. of sRDMA to IPSec and TLS over RoCE.

Protocol	Sec. comm.	IBA supp.	One-sided comm.	Hdr overhead.
RDMA	✗	✓	✓	-
IPSec	✓	✗	✗	50-80 B
(d)TLS	✓	✗	✗	25-40 B
[87, 86]	✗	✗	✓	12-16 B
sRDMA	✓	✓	✓	12-64 B

A naive application of IPSec to RoCE packets would not achieve source authentication as all RoCE traffic is destined to the same UDP port (and not the QPN). Thus, the use of IPsec would incur changes in the packet format, whereas sRDMA is supported by native IBA and RoCE. Additionally, the complexity of IPsec and its high processing overheads [116] make it ill-suited for high-performance and low-latency applications and would introduce a header overhead of 50-80 bytes [78]. While the IPsec-enabled Cavium LiquidIO III [100] and Mellanox Innova [154] NICs support RoCE, they do not support IPsec-based protection of RoCE packets.

Lee et al. [87, 86] discuss security vulnerabilities in IBA and show that they could be exploited by an adversary with moderate overhead. The authors suggest to replace the Invariant CRC field with a MAC to achieve packet authentication. Unfortunately, this might lead to routers dropping packets with invalid ICRC, making the proposed solution incompatible with legacy routers. Additionally, they discuss how IBA could reduce its key exposure risk by introducing partition- and queue-level key distributions. However, modifying partition-level keys can lead to packets being dropped as they might be used by routers and switches to enforce partitioning. Furthermore, their design uses the 24 bit PSN as a nonce which cause a reuse of a PSN after 80 ms on modern RNICs [95]. Finally, the authors provide no implementation of their system, but rather simulate the performance of symmetric ciphers to show that they are suitable for high performance networking.

RDMA Side-Channel Attack. Kurth et al. [83] have shown that the Intel DDIO [38] and RDMA features facilitate a side-channel attack named NetCAT. Intel DDIO technology allows RDMA reads and writes access not only the pinned memory region but also parts of the last level cache of the CPU. NetCAT remotely measures cache activity caused

by a victim SSH connection to perform a keystroke timing analysis. An attacker can make use of the attack to recover words typed by a victim client in the SSH session from another computer.

Tsai et al. [157] implemented a set of RDMA-based remote sidechannel attacks that allow an attacker on one client machine to learn how victims on other client machines access data. They further extend their work by building side-channel attacks on Crail [145].

Using sRDMA a large attack surface could be removed by permitting only trusted entities to initiate RDMA requests.

6.8 Summary and Discussion

Using NIC-based authentication and encryption enables secure communication for systems that require high performance guarantees such as RDMA mechanisms. sRDMA provides strong authenticity and secrecy, and prevents several forms of DoS attacks. Thus, safety- and security-critical applications that rely on RDMA must use sRDMA to prevent attacks by malicious entities within the same network.

Our software implementation on the SmartNIC causes a high load due to data movement overheads. The datapath could be optimized with a different architecture using specialized programmable packet processing units [77, 57]. Furthermore, sRDMA could also be hardened into fixed logic as the area and power consumption overhead are marginal compared to regular input/output processing [51, 93, 69]. Additionally, sRDMA minimizes memory consumption on the RNIC using PD-level protection.

7

Conclusions and future work

Data management in networked systems has been a difficult system problem for decades, and the advent of new network accelerators makes efficient system design even more challenging. The modern network devices can be used to create system architectures that are aware of all capabilities of their network devices, achieving higher performance and providing more features than conventional systems. However, as we show in this dissertation, the design of such architectures often require a complete redesign of data structures and access protocols to satisfy the client's requirements and to obtain the necessary performance.

In this thesis, we explore and address complex data management problems across various networked systems to improve their efficiency, security, and performance guarantees. With this dissertation, we hope to contribute to the design of future data management systems over RDMA-capable networks.

In Chapter 2, we have shown how the dynamic management of storage schemes in key-value stores can improve the utilization of cluster resources. Existing storage schemes such as replication and erasure codes provide different trade-offs in terms of reliability, network and storage costs, and request processing. Using the trade-offs, the developers can tune operation costs while maintaining the highest performance.

In Chapter 3, we have demonstrated the benefits of the memory compaction in the RDMA-enabled remote memory systems. More concretely, we showed how systems that utilize one-sided RDMA operations can support compaction without compromising object identifiers stored by clients. We achieved it by introducing a compaction algorithm based on RDMA-aware page remapping that does not sacrifice RDMA access tokens.

In Chapter 4, we presented KafkaDirect, which illustrated the most efficient way of using existing RDMA features to accelerate accesses to log-structured message queues. We showed that the choice of RDMA requests is not trivial as they achieve different performance guarantees. With a series of experiments, we have found the fastest approaches for directly writing records to and reading records from Kafka's storage files, thereby outperforming all existing implementations of Kafka in both latency and throughput.

Chapter 5 focused on the communication of Java objects across Java virtual machines. We presented a collection of algorithms and data structures to efficiently transmit large complex object structures and take advantages of offloaded zero-copy networking with RDMA. Our experiments indicated the need for contiguous or sparsity-optimized data structures to empower networked systems to take full advantage of RDMA accelerations.

Chapter 6 described in detail how RDMA providers can implement secure channels for the InfiniBand Architecture. We also demonstrated that a naive adoption of IPSec for RDMA cannot incorporate InfiniBand-related features compared to our design, thereby weakening security of one-sided memory accesses and missing potential memory savings in managing security keys.

7.1 Future work

Our work motivates further research in accelerated data management and in the in-network data processing. A prominent research direction is the system design with the presence of programmable network controllers, that empower the developers to extend RDMA requests to execute small data processing functions in network controllers.

Erasure coded storage

An advantage of implementing erasure coding over RDMA, as in Chapter 2, is that CPUs on nodes are not involved in receiving and sending messages. Instead, they can perform

other operations, such as data recovery and data coding. Future programmable NICs are expected to extend the success of RDMA to simple data processing tasks that are dominated by data movement. In fact, Reed-Solomon coding over GF(2^8), the field used in industrial data storages [40, 59, 136], is a memory-bound problem that consumes a lot of CPU cycles on moving data to and from CPU caches. It would be interesting to design in-network erasure coded data storage that does not spend CPU time on any GF math, thereby making the CPU cost of the erasure coded storage the same as of a common replicated storage and still taking all advantages of reduced storage overhead. We believe that such offload could significantly reduce the management and storage cost of existing data storage systems.

Compute Express Link

Compute Express Link (CXL) is an upcoming cache-coherent interconnect for processors and accelerators. CXL aims to bring a unified, coherent memory space between the CPU and acceleration devices, allowing them to coherently cache and share memory resources. Such cache-coherent interconnect should improve the performance of various accelerators such as GPUs and programmable NICs and reduce software stack complexity for systems that want to utilize the accelerators. The advent of CXL is expected to facilitate a new generation of accelerated data management architectures, bringing new research problems.

RDMA Append capability

In Chapter 4, multiple RDMA-enabled producers could publish the records to the same topic partition only by synchronizing using RDMA atomics, that have limited throughput and entails multiple round trips. However, the shared RDMA accesses could be realized with no coordination if all incoming produce requests were serialized by the network device and written to the appropriate file offsets. Such an approach could be implemented on specialized RNIC devices that support *RDMA Append with tag matching* [15]. RDMA Append functionality requires the RNIC to reuse a posted receive request for multiple incoming messages (this feature is available for some InfiniBand devices and is called "multi-packet work request"), whereas a standard receive request can be used only once. Tag matching is available on some InfiniBand devices, and it allows the RNIC to match tagged incoming messages with receive requests which have the same tag. Tag matching capability enables RNIC to distinguish sends to different files and redirect each of them to

the requested file. However, such capabilities are not yet offered by existing RNICs, but could be easily implemented on the upcoming programmable NICs.

Offloaded Naos

In Chapter 5, the Naos library cannot modify its on-heap memory before sending. Therefore, a receiver has to employ a complex pointer recovery algorithm, whereas Skyway can pre-process buffers before sending them to help the receiver to recover objects faster. We believe that such a feature is better implemented at the network level: a network controller could fix the pointers on the fly before writing the data to DRAM. In particular, since a Naos' RDMA sender already knows the destination addresses of the objects, either the network controller at the sender or at the receiver could fix the object pointers. The class pointers could be fixed by storing class translation tables in the network controller. Such design would require the availability of programmable network controllers, allowing network devices to apply small data processing functions to incoming and outgoing messages.

Confidential computing

Confidential computing is a new trend in datacenter design, that assumes that all devices should not be trusted by default. In Chapter 6, the sRDMA protocol have to trust RNICs to allow them to encrypt and decrypt packets, as well as to trust DMA engines and PCIe interconnects in order to copy packets unencrypted to and from a local RNIC. We believe that it is interesting to research how to enable high-performance secure communication channels for confidential computing platforms or at least strengthen the trust model of sRDMA. The zero-trust solution may require to be compatible with trusted execution environments (e.g., Intel SGX [39]) that offer zero-trust execution of software in cloud servers.

Bibliography

[1] RDMA core userspace libraries and daemons. `https://github.com/linux-rdma/rdma-core`, 2021. [Accessed 13-Sep-2021].

[2] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote Regions: A Simple Abstraction for Remote Memory. In *Proceedings of the 2018 USENIX Annual Technical Conference*, USENIX ATC'18, pages 775–787. USENIX Association, 2018.

[3] Marcos K. Aguilera, Kimberly Keeton, Stanko Novakovic, and Sharad Singhal. Designing Far Memory Data Structures: Think Outside the Box. In *Proceedings of the 17th Workshop on Hot Topics in Operating Systems*, HotOS'19, pages 120–126. Association for Computing Machinery, 2019.

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

[5] Robert Alverson, Duncan Roweth, and Larry Kaplan. The Gemini System Interconnect. In *Proceedings of the 18th IEEE Symposium on High-Performance Interconnects*, HOTI'10, pages 83–87. IEEE Computer Society, 2010.

[6] Oracle Corporation and/or its affiliates. Java SE Development Kit 11. `https://openjdk.java.net/projects/jdk/11/`, 2019. [Accessed 13-Sep-2021].

[7] Dubbo Apache. Apache Dubbo Project. `https://dubbo.apache.org`, 2018. [Accessed 13-Sep-2021].

[8] Kafka Apache. Apache Kafka Project. https://kafka.apache.org, 2011. [Accessed 13-Sep-2021].

[9] MINA Apache. Apache MINA Project, 2009. https://mina.apache.org.

[10] InfiniBand Trade Association et al. *The InfiniBand Architecture Specification 1.4*, 2020. https://www.infinibandta.org/ibta-specification/.

[11] InfiniBand Trade Association et al. *The InfiniBand Architecture Specification 1.4, Annex A16: RDMA over Converged Ethernet (RoCE)*, 2020. https://www.infinibandta.org/ibta-specification/.

[12] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-Scale Key-Value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS'12, pages 53–64. Association for Computing Machinery, 2012.

[13] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. Group Formation in Large Social Networks: Membership, Growth, and Evolution. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD'06, pages 44–54. Association for Computing Machinery, 2006.

[14] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobbler, Michael Wei, and John D. Davis. CORFU: A Shared Log Design for Flash Clusters. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'12, pages 1–14. USENIX Association, 2012.

[15] 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. *Sandia National Laboratories, November 2012, Technical Report SAND2012*, 10087, 2012.

[16] Claude Barthels, Gustavo Alonso, and Torsten Hoefler. Designing databases for future high-performance networks. *IEEE Data Engineering Bulletin*, 40(1):15–26, 2017.

[17] Claude Barthels, Simon Loesing, Gustavo Alonso, and Donald Kossmann. Rack-Scale In-Memory Join Processing Using RDMA. In *Proceedings of the 2015 ACM International Conference on Management of Data*, SIGMOD'15, pages 1463–1475. Association for Computing Machinery, 2015.

[18] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'IX, pages 117–128. Association for Computing Machinery, 2000.

[19] Daniel J Bernstein. The Poly1305-AES Message-Authentication Code. In *Proceedings of the 12th International Conference on Fast Software Encryption*, FSE'05, pages 32–49. Springer-Verlag, 2005.

[20] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. The End of Slow Networks: It's Time for a Redesign. *Proceedings of the VLDB Endowment*, 9(7):528–539, 2016.

[21] Andrew D. Birrell and Bruce Jay Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, 1984.

[22] 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 23rd IEEE Symposium on High-Performance Interconnects*, HOTI'15, pages 1–9. IEEE Computer Society, 2015.

[23] William J Bolosky, Dexter Bradshaw, Randolph B Haagens, Norbert P Kusters, and Peng Li. Paxos Replicated State Machines as the Basis of a High-Performance Data Store. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'11, pages 141–154. USENIX Association, 2011.

[24] Broadcom. Stingray PS225 2x25Gb High-Performance Data Center Smart NIC. `https://docs.broadcom.com/doc/PS225-PB`, 2018. [Accessed 13-Sep-2021].

[25] Rodrigo Bruno, Vojin Jovanovic, Christian Wimmer, and Gustavo Alonso. Compiler-Assisted Object Inlining with Value Fields. In *Proceedings of the 42nd*

ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI'21, pages 128–141. Association for Computing Machinery, 2021.

[26] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. The Primary-Backup Approach. *Distributed Systems*, 2:199–216, 1993.

[27] Chiranjeeb Buragohain, Knut Magne Risvik, Paul Brett, Miguel Castro, Wonhee Cho, Joshua Cowhig, Nikolas Gloy, Karthik Kalyanaraman, Richendra Khanna, John Pao, Matthew Renzelmann, Alex Shamis, Timothy Tan, and Shuheng Zheng. A1: A Distributed In-Memory Graph Database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD'20, pages 329–344. Association for Computing Machinery, 2020.

[28] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. Efficient Distributed Memory Management with RDMA and Caching. *Proceedings of the VLDB Endowment*, 11(11):1604–1617, 2018.

[29] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink: Stream and batch processing in a single engine. *IEEE Data Engineering Bulletin*, 36(4), 2015.

[30] Barbara Chapman, Tony Curtis, Swaroop Pophale, Stephen Poole, Jeff Kuehn, Chuck Koelbel, and Lauren Smith. Introducing OpenSHMEM: SHMEM for the PGAS community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, PGAS'10, pages 1–3. Association for Computing Machinery, 2010.

[31] Haibo Chen, Rong Chen, Xingda Wei, Jiaxin Shi, Yanzhe Chen, Zhaoguo Wang, Binyu Zang, and Haibing Guan. Fast In-Memory Transaction Processing Using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP'15, pages 87–104. Association for Computing Machinery, 2015.

[32] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185, 1994.

[33] Youmin Chen, Youyou Lu, and Jiwu Shu. Scalable RDMA RPC on Reliable Connection with Efficient Resource Sharing. In *Proceedings of the 14th EuroSys Conference*, EuroSys'19, pages 1–14. Association for Computing Machinery, 2019.

[34] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'20, pages 1077–1091. Association for Computing Machinery, 2020.

[35] Alibaba Cloud. Super computing cluster. `https://www.alibabacloud.com/product/scc`, 2018. [Accessed 13-Sep-2021].

[36] Inc. Cloudera. kafka-*-perf-test. `https://docs.cloudera.com/runtime/7.2.0/kafka-managing/topics/kafka-manage-cli-perf-test.html`, 2019. [Accessed 13-Sep-2021].

[37] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC'10, pages 143–154. Association for Computing Machinery, 2010.

[38] Intel Corporation. Intel Data Direct I/O Technology Overview. `https://www.intel.co.jp/content/dam/www/public/us/en/documents/white-papers/data-direct-i-o-technology-overview-paper.pdf`, 2012. [Accessed 13-Sep-2021].

[39] Intel Corporation. Intel Software Guard Extensions. `https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions.html`, 2015. [Accessed 13-Sep-2021].

[40] Intel Corporation. Intelligent Storage Acceleration Library. `https://github.com/intel/isa-l`, 2021. [Accessed 13-Sep-2021].

[41] Joan Daemen and Vincent Rijmen. AES Proposal: Rijndael. 1999.

[42] Daniele De Sensi, Salvatore Di Girolamo, Kim H. McMahon, Duncan Roweth, and Torsten Hoefler. An In-Depth Analysis of the Slingshot Interconnect. In *Proceed-*

ings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC'20, pages 1–14. IEEE Press, 2020.

[43] Leonard Eugene Dickson. *Linear groups: With an exposition of the Galois field theory.* Courier Corporation, 2003.

[44] Cong Ding, David Chu, Evan Zhao, Xiang Li, Lorenzo Alvisi, and Robbert Van Renesse. Scalog: Seamless Reconfiguration and Total Order in a Scalable Shared Log . In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'20, pages 325–338. USENIX Association, 2020.

[45] Naganand Doraswamy and Dan Harkins. *IPSec: the new security standard for the Internet, intranets, and virtual private networks.* Prentice Hall Professional, 2003.

[46] Aleksandar Dragojevic, Dushyanth Narayanan, and Miguel Castro. RDMA reads: To use or not to use? *IEEE Data Engineering Bulletin*, 40(1):3–14, 2017.

[47] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'14, pages 401–414. USENIX Association, 2014.

[48] Aleksandar Dragojevic, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP'15, pages 54–70. Association for Computing Machinery, 2015.

[49] Brad Fitzpatrick. Distributed caching with memcached. *Linux Journal*, 2004(124), 2004.

[50] Christine H. Flood, Roman Kennke, Andrew Dinn, Andrew Haley, and Roland Westrelin. Shenandoah: An Open-Source Concurrent Compacting Garbage Collector for OpenJDK. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ'16. Association for Computing Machinery, 2016.

[51] Kris Gaj and Pawel Chodowiec. FPGA and ASIC implementations of AES. pages 235–294, 2009.

[52] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP'03, pages 29–43. Association for Computing Machinery, 2003.

[53] Ken Goodhope, Joel Koshy, Jay Kreps, Neha Narkhede, Richard Park, Jun Rao, and Victor Yang Ye. Building LinkedIn's Real-time Activity Data Pipeline. *IEEE Data Engineering Bulletin*, 35(2):33–45, 2012.

[54] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient Memory Disaggregation with INFINISWAP. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'17, pages 649–667, USA, 2017. USENIX Association.

[55] R. Guerraoui and A. Schiper. Total order multicast to multiple groups. In *Proceedings of the 17th International Conference on Distributed Computing Systems*, ICDCS'97, pages 578–585. IEEE Computer Society, 1997.

[56] Robin Hecht and Stefan Jablonski. NoSQL evaluation: A use case oriented survey. In *Proceedings of the 2011 International Conference on Cloud and Service Computing*, CSC'11, pages 336–341. IEEE Computer Society, 2011.

[57] Torsten Hoefler, Salvatore Di Girolamo, Konstantin Taranov, Ryan E. Grant, and Ron Brightwell. sPIN: High-performance Streaming Processing In the Network. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC'17, pages 1–16. Association for Computing Machinery, 2017.

[58] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP'91, pages 21–38. Springer, 1991.

[59] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. Erasure Coding in Windows Azure Storage. In *Proceedings of the 2012 USENIX Annual Technical Conference*, USENIX ATC'12, pages 15–26. USENIX Association, 2012.

[60] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of*

the 2010 USENIX Annual Technical Conference, USENIX ATC'10. USENIX Association, 2010.

[61] Russell Impagliazzo, Leonid A Levin, and Michael Luby. Pseudo-random generation from one-way functions. In *Proceedings of the 21st annual ACM symposium on Theory of computing*, STOC'89, pages 12–24. Association for Computing Machinery, 1989.

[62] Google Inc. Protocol buffers: Google's data interchange format. `https://github.com/protocolbuffers/protobuf`, 2008. [Accessed 13-Sep-2021].

[63] Google Inc. Pub/Sub. `https://cloud.google.com/pubsub`, 2020. [Accessed 13-Sep-2021].

[64] Jana Iyengar and Martin Thomson. QUIC: A UDP-Based Multiplexed and Secure Transport. Internet-Draft RFC 9000, Internet Engineering Task Force, May 2021.

[65] Jaeyoung Jang, Sung Jun Jung, Sunmin Jeong, Jun Heo, Hoon Shin, Tae Jun Ham, and Jae W. Lee. A Specialized Architecture for Object Serialization with Applications to Big Data Analytics. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ISCA'20, pages 322–334. IEEE Press, 2020.

[66] Sagar Jha, Jonathan Behrens, Theo Gkountouvas, Matthew Milano, Weijia Song, Edward Tremel, Robbert Van Renesse, Sydney Zink, and Kenneth P. Birman. Derecho: Fast State Machine Replication for Cloud Services. *ACM Transactions on Computer Systems*, 36(2), 2019.

[67] Jim Seeger, Naresh Chainani, Aruna De Silva, Karen Mcculloch, Kiran Chinta, Vincent Kulandai Samy, Tom Hart. *DB2 V10.1 Multi-temperature Data Management Recommendations*, 2012.

[68] Ricardo Jiménez-Peris, M. Patiño Martínez, Gustavo Alonso, and Bettina Kemme. Are Quorums an Alternative for Data Replication? *ACM Transactions on Database Systems*, 28(3):257–294, 2003.

[69] Hyun-Wook Jin, Pavan Balaji, Chuck Yoo, Jin-Young Choi, and Dhabaleswar K Panda. Exploiting NIC architectural support for enhancing IP-based protocols

on high-performance networks. *Journal of Parallel and Distributed Computing*, 65(11):1348–1365, 2005.

[70] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using RDMA Efficiently for Key-Value Services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM'14, pages 295–306. Association for Computing Machinery, 2014.

[71] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design Guidelines for High Performance RDMA Systems. In *Proceedings of the 2016 USENIX Annual Technical Conference*, USENIX ATC'16, pages 437–450. USENIX Association, 2016.

[72] Anuj Kalia, Michael Kaminsky, and David G. Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 185–201. USENIX Association, 2016.

[73] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a Warehouse-Scale Computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA'15, pages 158–169. Association for Computing Machinery, 2015.

[74] Tejas Karmarkar. Availability of linux RDMA on Microsoft Azure. `https://azure.microsoft.com/en-us/blog/azure-linux-rdma-hpc-available`, 2015. [Accessed 13-Sep-2021].

[75] Antonios Katsarakis, Vasilis Gavrielatos, M.R. Siavash Katebzadeh, Arpit Joshi, Aleksandar Dragojevic, Boris Grot, and Vijay Nagarajan. Hermes: A Fast, Fault-Tolerant and Linearizable Replication Protocol. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'20, pages 201–217. Association for Computing Machinery, 2020.

[76] Jonathan Katz, Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography.* CRC press, 1996.

[77] Antoine Kaufmann, Simon Peter, Naveen Kr Sharma, Thomas Anderson, and Arvind Krishnamurthy. High Performance Packet Processing with FlexNIC. In *Proceedings of the Twenty-First International Conference on Architectural Support*

for Programming Languages and Operating Systems, ASPLOS'16, pages 67–81. Association for Computing Machinery, 2016.

[78] Stephen Kent. IP Authentication Header. Internet-Draft RFC 4302, Network Working Group, 2005.

[79] Anurag Khandelwal, Rachit Agarwal, and Ion Stoica. BlowFish: Dynamic Storage-Performance Tradeoff in Data Stores. In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'16, pages 485–500. USENIX Association, 2016.

[80] 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 *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM'18, pages 297–312. Association for Computing Machinery, 2018.

[81] Tim Kraska, Martin Hentschel, Gustavo Alonso, and Donald Kossmann. Consistency Rationing in the Cloud: Pay Only when It Matters. *Proceedings of the VLDB Endowment*, 2(1):253–264, August 2009.

[82] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the 2011 IEEE International Workshop on Networking Meets Databases*, NetDB'11, pages 1–7, 2011.

[83] Michael Kurth, Ben Gras, Dennis Andriesse, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. NetCAT: Practical Cache Attacks from the Network. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy*, S&P'20, pages 20–38, 2020.

[84] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. Coordinated and Efficient Huge Page Management with Ingens. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'16, pages 705–721. USENIX Association, 2016.

[85] Sean Leary. JSON-java. `https://github.com/stleary/JSON-java`, 2015. [Accessed 13-Sep-2021].

[86] Manhee Lee and Eun Jung Kim. A comprehensive framework for enhancing security in InfiniBand architecture. *IEEE Transactions on Parallel and Distributed Systems*, 18(10), 2007.

[87] Manhee Lee, Eun Jung Kim, and Mazin Yousif. Security enhancement in InfiniBand architecture. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing*, IPDPS'05. IEEE Computer Society, 2005.

[88] Ilya Lesokhin, Haggai Eran, Shachar Raindel, Guy Shapiro, Sagi Grimberg, Liran Liss, Muli Ben-Yehuda, Nadav Amit, and Dan Tsafrir. Page Fault Support for Network Controllers. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'17, pages 449–466. Association for Computing Machinery, 2017.

[89] 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 *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP'17, pages 137–152. Association for Computing Machinery, 2017.

[90] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making Geo-replicated Systems Fast As Possible, Consistent when Necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 265–278. USENIX Association, 2012.

[91] Feng Li, Sudipto Das, Manoj Syamala, and Vivek R. Narasayya. Accelerating Relational Databases by Leveraging Remote Memory and RDMA. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD'16, pages 355–370. Association for Computing Machinery, 2016.

[92] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'14, pages 429–444, Seattle, WA, 2014. USENIX Association.

[93] Bin Liu and Bevan M Baas. Parallel AES encryption engines for many-core processor arrays. *IEEE Transactions on Computers*, 62(3):536–547, 2013.

[94] Joshua Lockerman, Jose M. Faleiro, Juno Kim, Soham Sankaran, Daniel J. Abadi, James Aspnes, Siddhartha Sen, and Mahesh Balakrishnan. The FuzzyLog: A Partially Ordered Shared Log. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, pages 357–372. USENIX Association, 2018.

[95] Mellanox Technologies Ltd. ConnectX-6 EN Single/Dual-Port Adapter. https://www.mellanox.com/products/infiniband-adapters/connectx-6, 2019. [Accessed 13-Sep-2021].

[96] Mellanox Technologies Ltd. Optimized Memory Access. https://docs.mellanox.com/display/MLNXOFEDv492240/Optimized+Memory+Access, 2020. [Accessed 13-Sep-2021].

[97] X. Lu, D. Shankar, S. Gugnani, and D. K. Panda. High-performance design of Apache Spark with RDMA and its benefits on various workloads. In *Proceedings of the 2016 IEEE International Conference on Big Data*, BigData'16, pages 253–262. IEEE Computer Society, 2016.

[98] Martin Maas, David G. Andersen, Michael Isard, Mohammad Mahdi Javanmard, Kathryn S. McKinley, and Colin Raffel. Learning-Based Memory Allocation for C++ Server Workloads. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'20, pages 541–556. Association for Computing Machinery, 2020.

[99] Linux Programmer's Manual. memfd_create - create an anonymous file. https://man7.org/linux/man-pages/man2/memfd_create.2.html, 2019. [Accessed 13-Sep-2021].

[100] Marvell. Marvell LiquidIO III: An inline DPU based SmartNIC for cloud network and security acceleration. https://www.marvell.com/content/dam/marvell/en/public-collateral/embedded-processors/marvell-liquidio-III-solutions-brief.pdf, 2020. [Accessed 13-Sep-2021].

[101] Microsoft. Azure service bus messaging. https://azure.microsoft.com/en-us/services/service-bus/, 2020. [Accessed 13-Sep-2021].

[102] Microsoft Azure. Azure Storage Pricing. https://azure.microsoft.com/en-us/pricing/details/storage/blobs/, 2018. [Accessed 13-Sep-2021].

[103] Microsoft Azure. Understanding Block Blobs, Append Blobs, and Page Blobs. https://docs.microsoft.com/en-us/rest/api/storageservices/Understanding-Block-Blobs--Append-Blobs--and-Page-Blob, 2021. [Accessed 13-Sep-2021].

[104] Frank Mietke, R. Baumgartl, R. Rex, Torsten Mehlan, Torsten Hoefler, and Wolfgang Rehm. Analysis of the Memory Registration Process in the Mellanox InfiniBand Software Stack. In *Proceedings of the 12th European Conference on parallel computing*, Euro-Par'06, pages 124–133. Springer-Verlag, 2006.

[105] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *Proceedings of the 2016 USENIX Annual Technical Conference*, USENIX ATC'16, pages 103–114. USENIX Association, 2013.

[106] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-Tolerant Software Distributed Shared Memory. In *Proceedings of the 2015 USENIX Annual Technical Conference*, USENIX ATC'15, pages 291–305. USENIX Association, 2015.

[107] The Ohio State University Network-Based Computing Laboratory. RDMA-based Apache Kafka (RDMA-Kafka). https://hibd.cse.ohio-state.edu/#kafka, 2018. [Accessed 13-Sep-2021].

[108] Khanh Nguyen, Lu Fang, Christian Navasca, Guoqing Xu, Brian Demsky, and Shan Lu. Skyway: Connecting Managed Heaps in Distributed Big Data Systems. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'18, pages 56–69. Association for Computing Machinery, 2018.

[109] Oracle. Oracle Cloud. https://www.oracle.com/cloud/hpc/, 2020. [Accessed 13-Sep-2021].

[110] Oracle. Oracle messaging cloud service. https://www.oracle.com/technical-resources/articles/cloud/wilkins-ocms.html, 2020. [Accessed 13-Sep-2021].

[111] 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 Transactions on Computer Systems*, 33(3):7:1–7:55, August 2015.

[112] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. Making Sense of Performance in Data Analytics Frameworks. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'15, pages 293–307. USENIX Association, 2015.

[113] Sathish K Palaniappan and Pramod B Nagaraja. Efficient data transfer through zero copy. *IBM developerworks*, 2008.

[114] Ashish Panwar, Sorav Bansal, and K. Gopinath. HawkEye: Efficient Fine-Grained OS Support for Huge Pages. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'19, pages 347–360. Association for Computing Machinery, 2019.

[115] Ashish Panwar, Aravinda Prasad, and K. Gopinath. Making Huge Pages Actually Useful. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'18, pages 679–692. Association for Computing Machinery, 2018.

[116] Jungho Park, Wookeun Jung, Gangwon Jo, Ilkoo Lee, and Jaejin Lee. PIPSEA: A Practical IPsec Gateway on Embedded APUs. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS'16, pages 1255–1267. Association for Computing Machinery, 2016.

[117] Amar Phanishayee, David G Andersen, Himabindu Pucha, Anna Povzner, and Wendy Belluomini. Flex-KV: Enabling high-performance and flexible KV systems. In *Proceedings of the 2012 workshop on Management of big data systems*, pages 19–24. ACM, 2012.

[118] J. Pinkerton and E. Deleganes. Direct Data Placement Protocol (DDP) / Remote Direct Memory Access Protocol (RDMAP) Security. Technical Report RFC 5042, Network Working Group, October 2007.

[119] James S Plank et al. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software-Practice and Experience*, 27(9):995–1012, 1997.

[120] James S Plank, KM Greenan, EL Miller, and WB Houston. GF-Complete: A comprehensive open source library for Galois Field arithmetic. Technical Report UT-CS-13-703, University of Tennessee, 2013.

[121] James S Plank, Scott Simmerman, and Catherine D Schuman. Jerasure: A library in C/C++ facilitating erasure coding for storage applications-Version 1.2. (CS-08-627), 2008.

[122] Marius Poke and Torsten Hoefler. DARE: High-Performance State Machine Replication on RDMA Networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC'15, pages 107–118. Association for Computing Machinery, 2015.

[123] Arash Pourhabibi, Siddharth Gupta, Hussein Kassir, Mark Sutherland, Zilu Tian, Mario Paulo Drumond, Babak Falsafi, and Christoph Koch. Optimus Prime: Accelerating Data Transformation in Servers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'20, pages 1203–1216. Association for Computing Machinery, 2020.

[124] Bobby Powers, David Tench, Emery D. Berger, and Andrew McGregor. Mesh: Compacting Memory Management for C/C++ Applications. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'19, pages 333–346. Association for Computing Machinery, 2019.

[125] OpenMessaging Project. Openmessaging benchmark framework. `https://github.com/openmessaging/openmessaging-benchmark`, 2017. [Accessed 13-Sep-2021].

[126] Renato Recio, Bernard Metzler, Paul Culley, Jeff Hilland, and Dave Garcia. A Remote Direct Memory Access Protocol Specification. Technical Report RFC 5040, Network Working Group, October 2007.

[127] E. Rescorla and N. Modadugu. Datagram Transport Layer Security Version 1.2. Technical Report RFC 6347, Network Working Group, January 2012.

[128] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. Technical Report RFC 8446, Network Working Group, August 2018.

[129] Eric Rescorla and Tim Dierks. The Transport Layer Security (TLS) Protocol Version 1.2. Technical Report RFC 5246, Network Working Group, August 2008.

[130] John M Robson. Worst case fragmentation of first fit and best fit storage allocation strategies. *The Computer Journal*, 20(3):242–244, 1977.

[131] Wolf Rödiger, Tobias Mühlbauer, Alfons Kemper, and Thomas Neumann. High-Speed Query Processing over High-Speed Networks. *Proceedings of the VLDB Endowment*, 9(4):228–239, December 2015.

[132] Phillip Rogaway. Nonce-based symmetric encryption. In *Proceedings of the 2004 International Workshop on Fast Software Encryption*, FSE'04, pages 348–358. Springer Berlin Heidelberg, 2004.

[133] Benjamin Rothenberger, Konstantin Taranov, Adrian Perrig, and Torsten Hoefler. ReDMArk: Bypassing RDMA Security Mechanisms. In *Proceedings of the 30th USENIX Security Symposium*, USENIX Security'21. USENIX Association, 2021.

[134] Stephen M. Rumble, Ankita Kejriwal, and John Ousterhout. Log-structured Memory for DRAM-based Storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, FAST'14, pages 1–16. USENIX Association, February 2014.

[135] Salvatore Sanfilippo. Redis. `https://redis.io`, 2009. [Accessed 13-Sep-2021].

[136] Maheswaran Sathiamoorthy, Megasthenis Asteris, Dimitris Papailiopoulos, Alexandros G Dimakis, Ramkumar Vadali, Scott Chen, and Dhruba Borthakur. XORing Elephants: Novel Erasure Codes for Big Data. In *Proceedings of the VLDB Endowment*, volume 6, pages 325–336. VLDB Endowment, 2013.

[137] Fred B. Schneider. Byzantine Generals in Action: Implementing Fail-stop Processors. *ACM Transactions on Computer Systems*, 2(2):145–154, May 1984.

[138] Amazon Web Services. Amazon Simple Queue Service. `https://aws.amazon.com/sqs/`, 2020. [Accessed 13-Sep-2021].

[139] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *Proceedings of the 26th Symposium on Mass Storage Systems and Technologies Mass storage systems and technologies*, MSST'10, pages 1–10. IEEE Computer Society, 2010.

[140] Peter Snyder. tmpfs: A virtual memory file system. In *Proceedings of the Autumn 1990 European UNIX Users' Group Conference*, EUUG'90, pages 241–248, 1990.

[141] Esoteric Software. Kryo - object graph serialization framework for Java. `https://github.com/EsotericSoftware/kryo`, 2008. [Accessed 13-Sep-2021].

[142] Storage Performance Council. SPC Trace Files for OLTP Application I/O and Search Engine I/O. `http://traces.cs.umass.edu/index.php/Storage/Storage`, 2002. [Accessed 13-Sep-2021].

[143] Patrick Stuedi, Bernard Metzler, and Animesh Trivedi. JVerbs: Ultra-Low Latency for Data Center Applications. In *Proceedings of the 4th ACM Symposium on Cloud Computing*, SoCC'13. Association for Computing Machinery, 2013.

[144] Patrick Stuedi, Animesh Trivedi, Bernard Metzler, and Jonas Pfefferle. DaRPC: Data Center RPC. In *Proceedings of the 5th ACM Symposium on Cloud Computing*, SoCC'14, pages 1–13. Association for Computing Machinery, 2014.

[145] Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, Ana Klimovic, Adrian Schuepbach, and Bernard Metzler. Unification of Temporary Storage in the NodeKernel Architecture. In *Proceedings of the 2019 USENIX Annual Technical Conference*, USENIX ATC'19, pages 767–782. USENIX Association, 2019.

[146] Maomeng Su, Mingxing Zhang, Kang Chen, Zhenyu Guo, and Yongwei Wu. RFP: When RPC is Faster than Server-Bypass with RDMA. In *Proceedings of the 12th European Conference on Computer Systems*, EuroSys'17, pages 1–15. Association for Computing Machinery, 2017.

[147] Yacine Taleb, Ryan Stutsman, Gabriel Antoniu, and Toni Cortes. Tailwind: Fast and atomic rdma-based replication. In *Proceedings of the 2018 USENIX Annual Technical Conference*, USENIX ATC'18, pages 851–863. USENIX Association, 2018.

[148] Konstantin Taranov, Gustavo Alonso, and Torsten Hoefler. Fast and Strongly-consistent Per-item Resilience in Key-value Stores. In *Proceedings of the 13th EuroSys Conference*, EuroSys'18, pages 39:1–39:14. Association for Computing Machinery, 2018.

[149] Konstantin Taranov, Rodrigo Bruno, Gustavo Alonso, and Torsten Hoefler. Naos: Serialization-free RDMA networking in Java. In *Proceedings of the 2021 USENIX*

Annual Technical Conference, USENIX ATC'21, pages 1–14. USENIX Association, 2021.

[150] Konstantin Taranov, Salvatore Di Girolamo, and Torsten Hoefler. CoRM: Compactable Remote Memory over RDMA. In *Proceedings of the 2021 ACM International Conference on Management of Data*, SIGMOD'21. Association for Computing Machinery, 2021.

[151] Konstantin Taranov, Benjamin Rothenberger, Adrian Perrig, and Torsten Hoefler. sRDMA – Efficient NIC-based Authentication and Encryption for Remote Direct Memory Access. In *Proceedings of the 2020 USENIX Annual Technical Conference*, USENIX ATC'20, pages 691–704. USENIX Association, 2020.

[152] Mellanox Technologies. RDMA Aware Networks Programming User Manual, Rev 1.7. `https://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf`, 2015. [Accessed 13-Sep-2021].

[153] Mellanox Technologies. Mellanox BlueField SmartNIC. `http://www.mellanox.com/related-docs/prod_adapter_cards/PB_BlueField_Smart_NIC.pdf`, 2020. [Accessed 13-Sep-2021].

[154] Mellanox Technologies. Mellanox Innova-2 Flex Open Programmable SmartNIC. `https://www.mellanox.com/related-docs/prod_adapter_cards/PB_Innova-2_Flex.pdf`, 2020. [Accessed 13-Sep-2021].

[155] Gigabyte Technology. AORUS Gen4 AIC SSD 8TB. `https://www.gigabyte.com/Solid-State-Drive/AORUS-Gen4-AIC-SSD-8TB`, 2021. [Accessed 13-Sep-2021].

[156] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. Consistency-based Service Level Agreements for Cloud Storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP'13, pages 309–324. Association for Computing Machinery, 2013.

[157] Shin-Yeh Tsai, Mathias Payer, and Yiying Zhang. Pythia: remote oracles for the masses. In *Proceedings of the 28th USENIX Security Symposium*, USENIX Security'19, pages 693–710. USENIX Association, 2019.

[158] Hung-Wei Tseng, Qianchen Zhao, Yuxiao Zhou, Mark Gahagan, and Steven Swanson. Morpheus: Creating Application Objects Efficiently for Heterogeneous Computing. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA'16, pages 53–65. IEEE Press, 2016.

[159] Giselle van Dongen and Dirk Van den Poel. Evaluation of stream processing frameworks. *IEEE Transactions on Parallel and Distributed Systems*, 31(8):1845–1858, 2020.

[160] Robbert Van Renesse and Fred B Schneider. Chain Replication for Supporting High Throughput and Availability. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation*, OSDI'04, pages 91–104. USENIX Association, 2004.

[161] VMWare. ESXi VM and Hypervisor Escape Advisory. `https://blogs.vmware.com/security/2018/11/vmware-and-the-geekpwn2018-event.html`, 2019. [Accessed 13-Sep-2021].

[162] Guozhang Wang, Joel Koshy, Sriram Subramanian, Kartik Paramasivam, Mammad Zadeh, Neha Narkhede, Jun Rao, Jay Kreps, and Joe Stein. Building a Replicated Logging System with Apache Kafka. *Proceedings of the VLDB Endowment*, 8(12):1654–1655, August 2015.

[163] Michael Wei, Amy Tai, Christopher J. Rossbach, Ittai Abraham, Maithem Munshed, Medhavi Dhawan, Jim Stabile, Udi Wieder, Scott Fritchie, Steven Swanson, Michael J. Freedman, and Dahlia Malkhi. vCorfu: A Cloud-Scale Object Store on a Shared Log. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'17, pages 35–49. USENIX Association, 2017.

[164] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. Deconstructing RDMA-enabled Distributed Transactions: Hybrid is Better! In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'18, pages 233–251, Carlsbad, CA, 2018. USENIX Association.

[165] Matthias Wiesmann, Fernando Pedone, André Schiper, Bettina Kemme, and Gustavo Alonso. Database replication techniques: A three parameter classification. In *Proceedings 19th IEEE Symposium on Reliable Distributed Systems*, SRDS-2000, pages 206–215. IEEE Press, 2000.

[166] Christian Wimmer and Hanspeter Mössenböck. Automatic Feedback-Directed Object Inlining in the Java Hotspot™ Virtual Machine. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, VEE'07, pages 12–21. Association for Computing Machinery, 2007.

[167] Christian Wimmer and Hanspeter Mössenböck. Automatic Array Inlining in Java Virtual Machines. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO'08, pages 14–23. Association for Computing Machinery, 2008.

[168] Mingyuan Xia, Mohit Saxena, Mario Blaum, and David A. Pease. A Tale of Two Erasure Codes in HDFS. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, pages 213–226. USENIX Association, 2015.

[169] Jilong Xue, Youshan Miao, Cheng Chen, Ming Wu, Lintao Zhang, and Lidong Zhou. Fast Distributed Deep Learning over RDMA. In *Proceedings of the 14th European Conference on Computer Systems*, EuroSys'19. Association for Computing Machinery, 2019.

[170] Jaewon Yang and Jure Leskovec. Defining and Evaluating Network Communities Based on Ground-Truth. *Knowledge and Information Systems*, 42(1):181–213, 2015.

[171] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, page 10. USENIX Association, 2010.

[172] Erfan Zamanian, Carsten Binnig, Tim Harris, and Tim Kraska. The End of a Myth: Distributed Transactions Can Scale. *Proceedings of the VLDB Endowment*, 10(6):685–696, 2017.

[173] Erfan Zamanian, Xiangyao Yu, Michael Stonebraker, and Tim Kraska. Rethinking database high availability with rdma networks. *Proc. VLDB Endow.*, 12(11):1637–1650, July 2019.

[174] Heng Zhang, Mingkai Dong, and Haibo Chen. Efficient and Available In-memory KV-Store with Hybrid Erasure Coding and Replication. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies*, FAST'16, pages 167–180, Santa Clara, CA, 2016. USENIX Association.

[175] Yili Zheng, Amir Kamil, Michael B Driscoll, Hongzhang Shan, and Katherine Yelick. UPC++: a PGAS extension for C++. In *Proceedings of the 28th IEEE International Parallel and Distributed Processing*, IPDPS'14, pages 1105–1114. IEEE Computer Society, 2014.

[176] Weixi Zhu, Alan L. Cox, and Scott Rixner. A Comprehensive Analysis of Superpage Management Mechanisms and Policies. In *Proceedings of the 2020 USENIX Annual Technical Conference*, USENIX ATC'20, pages 829–842. USENIX Association, 2020.

[177] Tobias Ziegler, Sumukha Tumkur Vani, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. Designing Distributed Tree-Based Index Structures for Fast RDMA-Capable Networks. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD'19, pages 741–758. Association for Computing Machinery, 2019.