KafkaDirect: Zero-copy Data Access for Apache Kafka over RDMA Networks

Konstantin Taranov*
ETH Zurich
Switzerland
ktaranov@inf.ethz.ch

Steve Byan
Oracle Labs
USA
steve.byan@oracle.com

Virendra Marathe
Oracle Labs
USA
virendra.marathe@oracle.com

Torsten Hoefler
ETH Zurich
Switzerland
htor@inf.ethz.ch

ABSTRACT

Apache Kafka 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. 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. In this work, we explore the design choices including which RDMA operations to use to take full advantage of offloaded communication. Our RDMA design relies on one-sided RDMA requests 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. KafkaDirect can offer up to 9x increase in throughput for both Kafka producers and Kafka consumers, and can provide 4x and 50x reduction in latency for Kafka producers and Kafka consumers, respectively.

CCS CONCEPTS

• **Information systems** → **Parallel and distributed DBMSs**; • **Networks** → *In-network processing*; • **Computer systems organization** → *Distributed architectures.*

KEYWORDS

RDMA, Apache Kafka, Pub/Sub, Memory management

ACM Reference Format:
Konstantin Taranov, Steve Byan, Virendra Marathe, and Torsten Hoefler. 2022. KafkaDirect: Zero-copy Data Access for Apache Kafka over RDMA Networks . In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3514221.3526056

*Work done while at Oracle Labs.

SIGMOD '22, June 12–17, 2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-9249-5/22/06…$15.00
https://doi.org/10.1145/3514221.3526056

1 INTRODUCTION

Decreasing prices on RDMA-capable network controllers (RNICs) have made them widely available in data centers and cloud infrastructures [12, 24, 34]. RDMA networking offers higher throughput and lower latency compared to the traditional TCP/IP stack by offloading most of the networking functionality to RNICs, effectively bypassing the OS kernel. The availability of RNICs and their performance advantages have already affected the design of modern database management systems to improve performance of query processing [5, 43], data replication [8, 19, 25, 50, 63], distributed index structures [64], distributed transactions [9, 15, 59, 62], and processing of analytical database workloads [4, 29].

The naive use of RDMA, nonetheless, is unable to achieve maximum performance. The high-level frameworks such as RPCs [6] 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 [60]. In addition, 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 [3, 8, 18, 28]. Therefore, many applications are forced to use intermediate buffers or multiple round trips in their protocols. Although RDMA design challenges have been extensively studied for general key-value stores [20, 28, 32], not many RDMA solutions have been proposed for log-structured storage systems [21, 36].

In this paper, we explore the most efficient way of using existing RDMA features to accelerate Apache Kafka [26], a publish-subscribe log-structured storage 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 storage 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 them 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 (§6).

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 us, which is a 4x improvement compared to the 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. To the best of our knowledge, this is the first Kafka design for high-performance RDMA capable interconnects that exploits native RDMA programming.

In summary, we make the following contributions: We extend Apache Kafka to use RDMA for the Produce, Consume, and Replication datapaths without compromising backward compatibility. We investigate the space of possible design decisions for RDMA datapaths, which could be employed by other log-structured memory systems. The proposed RDMA design completely offloads processing of the consume datapath to RNICs. For other datapaths, the processing overhead is minimal and does not involve data copies. We extensively evaluate each RDMA-enabled component of KafkaDirect in comparison with the unmodified Kafka and the RDMA-accelerated Kafka [33] proposed by the Ohio State University.

2 BACKGROUND ON RDMA NETWORKING

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 RNICs without any CPU intervention or context switches. RDMA is offered by several network architectures [1, 7, 41]. In this work, we focus on the InfiniBand standard and its reliable RDMA connection type called *reliably connected queue pair (RC QP)*. Applications make use of offloaded RDMA communication by directly posting asynchronous work requests to an RNIC, bypassing the operating system. Upon completion of a request, the RNIC places a corresponding completion event into a completion queue created by the application.

In this work, 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 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* (CAS), and *Fetch-and-Add* (FAA).

Accelerating systems with RDMA. Networked systems employ RDMA to reduce CPU usage and to enable specific services by exploiting one-sided communication primitives that are more efficient than traditional socket interface. Typically, these systems are built from scratch to exploit RDMA writes for delivering requests directly to request queues and RDMA reads for fetching remote data without involvement of remote CPUs, thereby reducing overall latency and CPU usage. In our work, we also exploit RDMA Reads and Writes to accelerate communication and bypass CPU and operating system. However, unlike other projects, we aim to integrate RDMA networking to a huge codebase without compromising backward compatibility and still achieving zero-copy communication, that is even more challenging in managed language such as Java [51].

3 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 [16, 57, 58], as they shift the burden of reliable messaging from communicating applications. Publish-Subscribe applications are available as open-source systems (e.g., Apache Kafka [26], Corfu [2], Scalog [14], Fuzzylog [30]) and as a service by various cloud providers [23, 31, 35, 45]. 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 [26], but the RDMA design could be borrowed by other systems (§6).

Apache Kafka. Kafka [26] 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.

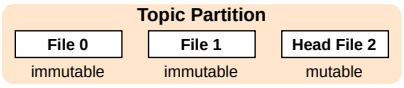

Figure 1: A topic partition may consist of multiple files. New records are appended to the head file. All preceding files of the topic partition are sealed and cannot be modified.

Kafka Topics. All records in Kafka are categorized into topics that are partitioned into 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 1). A new record is always appended to the head segment of the log. The record size in Kafka is limited to 1 MiB and the segment size is 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 local disks. 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 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: 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 [10, 15, 22, 48, 49] exist, their performance can still suffer from the RPC abstraction [60] 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

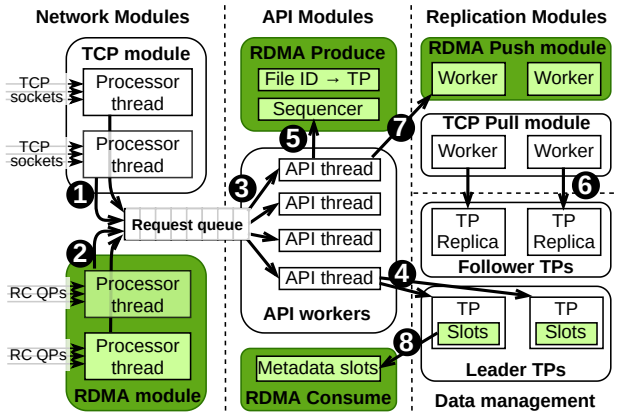

Figure 2: Kafka's broker architecture and our RDMA extensions (in color).

data from the network receive buffers to storage structures can further aggravate the already well-known CPU-bottleneck problems encountered in many distributed applications [37, 55]. 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 [33], 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 5 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 2. KafkaDirect has a dedicated RDMA network module (§4.1) to serve RC QP connections from clients and brokers. We extend Kafka with our RDMA Produce module (§4.2) 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 (§4.2.2) 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 (§4.3.2) that uses RDMA writes to replicate data directly from replication leaders to replication followers. Finally, consumers in KafkaDirect exploit RDMA Reads to directly read records from subscribed topics (§4.4), 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.

4.1 Network Layer

The original Kafka uses TCP connections to exchange requests between 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 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 (§4.2) and consume datapaths (§4.4) for zero-copy data accesses. Second, our replication and produce datapaths rely on the delivery guarantees of reliable transport to pipeline multiple produce requests (§4.2.2).

4.2 Produce datapaths

4.2.1 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. Once the records are committed, the broker starts replicating them (§4.3). The broker will make as many copies as configured for the requested topic, but the producer receives the acknowledgment as soon as the requested number of copies are made.

The shortcoming of the existing 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.

4.2.2 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.

KafkaDirect does not change the persistency model of the original Kafka, allowing KafkaDirect to reuse Kafka's original failover mechanisms. The produce datapath of KafkaDirect only uses RDMA to write records directly to the destination TPs, but not for committing them. The written records are persisted and processed according to the existing rules of Kafka, which include verifying checksums of new records, assigning offsets to new records, and committing the processed data.

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.

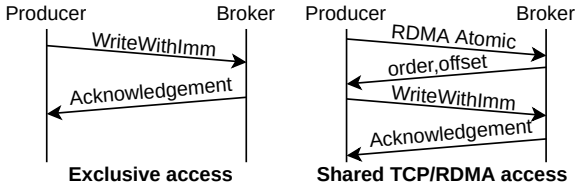

Figure 3: Methods for producing records via RDMA. In the exclusive mode, the single producer tracks the offset. In the shared mode, producers acquire the offset before they write.

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 WriteWithImm work requests to notify the broker of incoming Write requests. Note that WriteWithImm 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). 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 WriteWithImm 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

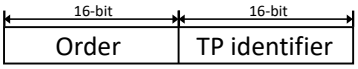

Figure 4: The 32-bit immediate value used to inform broker where the records has been written with WriteWithImm.

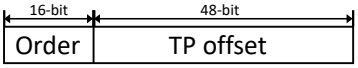

Figure 5: The 64-bit atomic value used to enforce ordering across producers. They must atomically fetch and increment the current order and offset before writing records.

avoid the situation where 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 CAS and 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 5.

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.

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). 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 ❺, 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

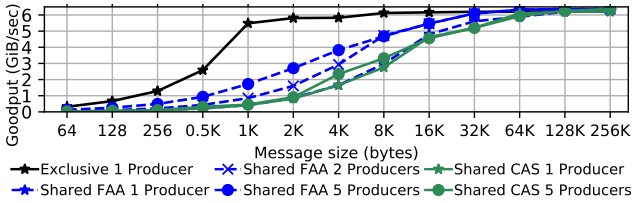

Figure 6: Aggregated Write bandwidth of different RDMA produce approaches with increasing message size.

abortion of all pending produce requests to the file. Clients can re-enable the RDMA datapath by requesting RDMA access again.

Performance comparison. Figure 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/s 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 2.68 Mreq/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. In other words, it is guaranteed that when the application receives metadata corresponding data records are already written to the main memory of the broker by the preceding Write request.

We evaluate the latency and bandwidth of the Write+Send approach against the WriteWithImm 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 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 us, whereas Write+Send approaches are by 1 us slower on average.

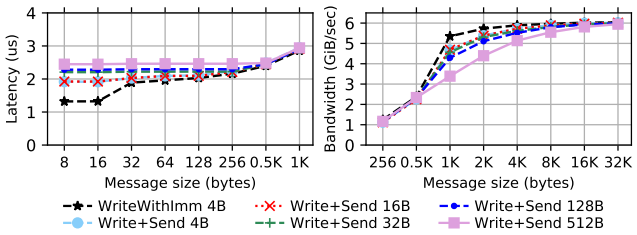

Figure 7: Latency and write bandwidth of different approaches for notifying the broker.

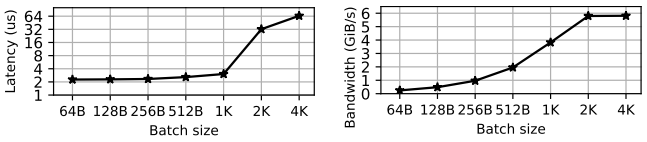

Figure 8: Latency and bandwidth of batching 64-byte RDMA Writes. Note log scale of Y axis for the latency plot.

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 metadata, 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 Replication datapaths

4.3.1 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 new records and appends ❻ them to the corresponding replica TP. Such an approach is commonly called a *pull* approach.

4.3.2 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.

The push replication module has RC QP connections to all other brokers that follow the TP. In KafkaDirect, the leader uses a *"get RDMA produce address"* request to get RDMA access to the replica files on the followers, and then uses RDMA WriteWithImm to write the data from its mapped file to the mapped files of the followers.

A naive implementation of push RDMA replication may have trouble with fast leaders that could overflow the RDMA completion queue of a slow follower leading to disconnection of all corresponding QPs. To prevent overflow of the RDMA queues, we use a credit-based approach where each follower of KafkaDirect assigns credit to the leader which limits its number of outstanding RDMA replicate requests. Setting limits on incoming replication requests allows followers to process replication requests at their own pace.

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 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 us, 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.

4.4 Consume datapaths

4.4.1 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, as brokers can serve thousands of consumers and each consumer periodically sends fetch requests regardless of whether the broker has new records.

4.4.2 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 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 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 gets incremented when new records are appended. Therefore, the RDMA consumer needs to periodically update that information. We propose to use RDMA Reads for reading the last readable byte values of TPs: 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 they need to request access to the new head file.

An RDMA consumer *periodically* reads, using RDMA, metadata slots of subscribed TPs to get informed whether new records have been appended to them. Since a consumer can be subscribed to several TPs, a naive reading of a single metadata slot at a time 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 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 9 needs to read all four slots (including two free slots) to update its metadata, whereas consumers 1 and 2 can read only

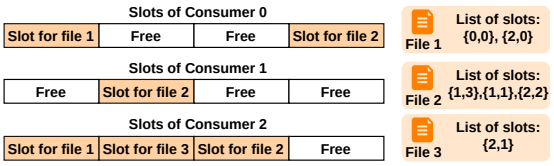

Figure 9: RDMA readable metadata slots for mutable files. KafkaDirect creates a contiguous region of slots for each consumer. Each registered file has a list of slots associated with it, so the slots may be updated as the file grows.

their active slots. The broker 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 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 us) 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.

5 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 [13], OpenMessaging [40], and the event processing [56] 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 [13, 40]. First, we configure KafkaDirect to enable RDMA only in the produce datapath and study the performance of exclusive and shared RDMA produce protocols (§5.1). Second, we deploy KafkaDirect in distributed mode and measure the latency and bandwidth of the RDMA replication module (§5.2). Then, we study the performance of the RDMA consume module for fetching new data records and checking the availability of new records (§5.3). Lastly, we show results for event processing benchmark [56] to show how KafkaDirect improves the performance of data processing frameworks such as Spark [61].

Implementation. Our implementation of KafkaDirect is based on Kafka 2.2.1. We use DiSNI [47], a low-latency RDMA library allowing applications to access RNICs directly from within the Java Virtual Machine through Java Native Interface calls. For this work, we also extended the DiSNI library to support RDMA atomics.

We compare the performance of KafkaDirect with the original Apache Kafka 2.2.1, and an RDMA-enabled Kafka [33] 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.

Settings. The experiments are conducted on a 12-node InfiniBand cluster, where each machine is equipped with a 56 Gbit/s 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, Kafka was deployed over the same network to have a fair comparison with RDMA-enabled systems. All implementations are deployed with 1 GiB log files and enabled file preallocation, i.e., Kafka immediately allocates storage for each created file. Unless otherwise specified, Kafka-based systems were deployed with default parameters that include eight API threads and three network threads.

To be completely oblivious of the performance of storage device used for storing log and TP files, Kafka's files are created in *tmpfs* [46], 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 tolerance only relies on replication and is independent of the availability of persistent storage. The bottleneck of the persistent storage 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 write bandwidth [54]) or the use of multiple storage media at each broker. We leave this exploration for future work.

5.1 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 10 shows that OSU Kafka reduces the latency of the original Kafka by about 90 us 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 us for small messages and approximately 345 us for large 128 KiB messages. Overall, KafkaDirect provides 3.3x and 2x improvement over Kafka and OSU Kafka, respectively.

The latency of an exclusive RDMA producer is 2.5 us lower than the shared TCP/RDMA producer. The difference comes from the

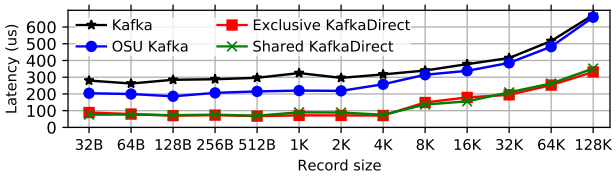

Figure 10: Latency of produce request when replication is disabled. Producers do not batch requests.

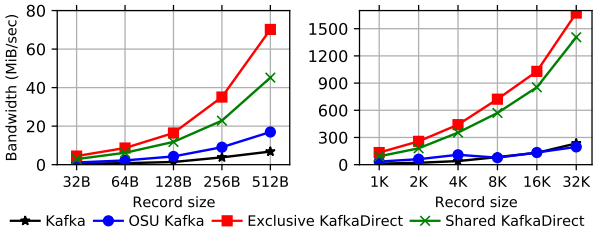

Figure 11: Bandwidth of produce request to one partition. Replication is disabled. Producers do not batch requests.

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 us. The overhead of 88 us 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 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 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 and shows a 5x improvement compared to Kafka.

Figure 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

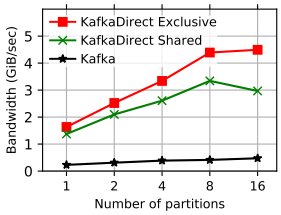

Figure 12: Bandwidth of produce requests for 32 KiB records.

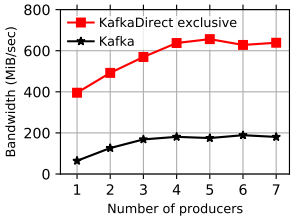

Figure 13: Total bandwidth of producers for 4 KiB records. Broker is deployed with one worker.

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 processing new records in a Kafka broker. KafkaDirect achieves 4.5 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.

We did not achieve the performance of our C/C++ microbenchmarks (see Figure 6). The reduction in bandwidth comes from the difference in processing new records and thread scheduling between Kafka and our C++ prototypes. In our C/C++ microbenchmarks, the producer could send data without copies, whereas Kafka always makes a copy of records at the producer to prevent their mutation. In addition, the C++ prototype did not perform integrity checks of records required by Kafka. Finally, Kafka uses different threads for API and network workers, incurring costly inter-thread communication, whereas our C++ prototypes were single-threaded.

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. The main goal of this configuration is to remove contention between threads at polling the request queue (see Figure 2).

Figure 13 reveals that the performance of the KafkaDirect broker plateaus at 630 MiB/sec when the system must process the records from more than four clients. For 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.*

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

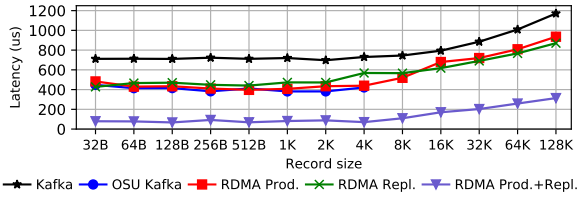

Figure 14: Latency of producer for 3-way replication.

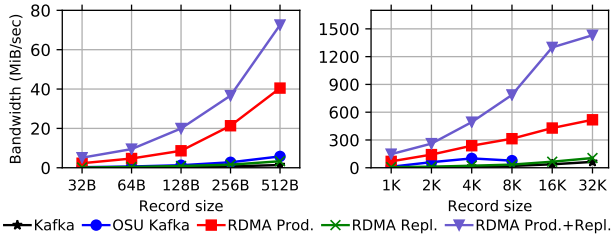

Figure 15: Bandwidth of producer for three-way replication.

According to Figure 14, the latency of Kafka with three-way replication is approximately 700 us 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 us. Interestingly, when both modules are enabled the latency decreases to about 100 us. OSU Kafka only reduces the latency by 300 us, 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 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. The speedup of KafkaDirect is from 9x to 14x depending on the size of the records.

To understand the role of the replication on performance, we measure the bandwidth of a producer with increasing replication factor. Figure 16 shows the bandwidth for 32 KiB records when data is replicated from one to four times (a replication factor of one means 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, providing a 14x speedup compared to the original Kafka.

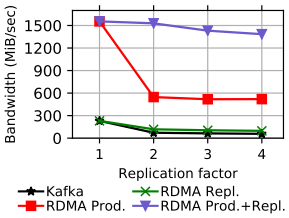

Figure 16: Bandwidth of produce request for 32 KiB records.

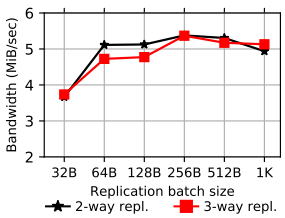

Figure 17: Bandwidth of 32-byte produce requests with increasing batch size.

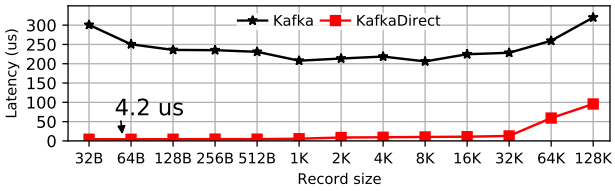

Figure 18: Latency of consumer with increasing record size.

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 [38] 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 (§4.3.2). 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 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, 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 (§4.3.2). 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.

5.3 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 18 shows that the latency of the original Kafka is at least 200 us 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 us, which is a 50x improvement over the original Kafka. The latency is so low because the data was preloaded to the broker and the RDMA consumer could read all the remote records with RDMA without requesting access to new TP files. In general, if an RDMA consumer reads entries from a TP consisting of many files, it needs to request file access after reading each 1 GiB of data.

The latency of an RDMA fetch request is 4.2 us, which is 2 us 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 (§4.4.2). The experiment reveals that the latency of an empty TCP fetch request is at least 200 us, whereas the latency of reading a remote metadata slot is only 2.5 us. 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 plays the role of producer and consumer. The client sends one record to Kafka and then fetches it with the consumer API to measure the round-trip latency consisting of 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.

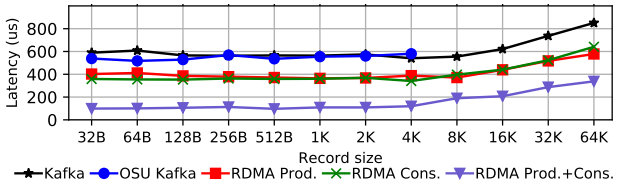

Figure 19: End-to-end latency with increasing record sizes.

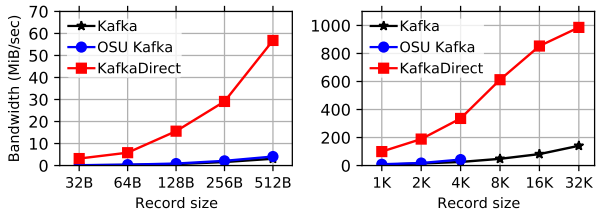

Figure 20: Consume bandwidth with increasing record size.

Figure 19 shows the median end-to-end latency of Kafka is about 600 us for small records. OSU Kafka gives approximately the same latency as the original Kafka, but at some data points, we observe a 50 us reduction in latency. The use of RDMA for either the produce or consume datapath reduces the latency by at least 200 us. When both RDMA modules are enabled the latency is as low as 100 us. Interestingly, as the latency of RDMA produce is about 93 us, the actual latency of RDMA fetch was about 7 us which consists of data fetching (4.2 us) and metadata update (2.8 us). *We conclude that KafkaDirect offers a 5.8x reduction in end-to-end latency, and that our RDMA consumer efficiently works 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 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 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.

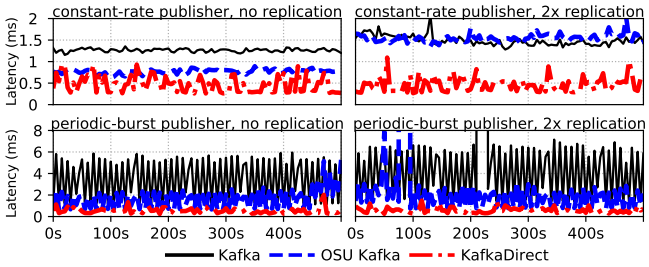

Figure 21: Event delays under constant-rate and periodic-burst workloads for no and 2x replication settings.

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 RDMA Reads to find out about newly available records by reading remote metadata slots (§4.4.2). Note that the processing of empty fetch requests in KafkaDirect does not involve the CPU of brokers and is bottlenecked by the RNIC speed.

5.4 Improving Data Processing Applications.

We integrated KafkaDirect into Apache Spark 2.4.4 [61] and measured its performance for the streaming benchmark [56]. 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 21 reports the measured delays for two workloads: constant-rate and periodic-burst. The first workload has a constant publishing rate (400 messages per second), whereas, in the periodic burst 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.*

6 RELATED WORK

Publish-subscribe systems. Corfu [2] and Scalog [14] 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's clients are also responsible for data replication. Unlike Corfu, a client of Scalog appends records to any server, which then replicates 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 [30] is a partially ordered log that tracks order between records stored in geo-replicated shards using Skeen's algorithm [17]. 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 storage systems. HERD [20], FlatStore [11], and RamCloud [36] 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 [25] is a framework that offloads chain replication to RNICs with cross-channel communications support [53]. HyperLoop improves the replication performance of write-ahead transactional logs and can be employed by log-structured systems. DaRE [39] 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.

7 DISCUSSION

Memory usage. The main disadvantage of KafkaDirect is that it has higher memory usage compared to the original Kafka. The increase in memory comes from a requirement of RDMA networking to have files mapped and be present in the main memory. Since the default file size is 1 GiB, each memory-mapped file increases the DRAM usage by 1 GiB. To alleviate this overhead, one can employ *on-demand paging* capability offered by modern RDMA controllers, which allows the OS to swap out RDMA accessible memory [27]. Since on-demand paging is not widely available, our current implementation does not use it, so that it can be deployed on any commodity RDMA-enabled network.

Batching requests targeting different TPs. In the original Kafka, a producer can batch produce requests to different TPs if they target the same broker to ease TCP/IP overheads. Our RDMA datapaths cannot batch RDMA requests that target different TP files. Thus, our RDMA producer needs to issue an RDMA write per target TP. Nonetheless, RDMA networking allows having multiple outstanding write requests, that are processed independently by the broker. Thus, we believe that this disadvantage is negligible.

Similarly, a TCP consumer can batch consume requests targeting the same broker. Our RDMA datapath requires multiple read requests to fetch new records from different TPs. However, our implementation allows RDMA consumers to fetch metadata of several TPs using a single RDMA request to check for new records (§4.4.2). In addition, an RDMA consumer can have multiple outstanding read requests to concurrently fetch records from different TPs.

Reliability of RDMA. We use reliable RDMA connections (similar to TCP), which guarantee that RDMA messages are delivered from a requester to a responder at most once, in order, and without corruption. In addition, RDMA reliable transport notifies applications about connection failures, allowing us to reuse failover mechanisms of the original Kafka.

Security of RDMA. The security of current RDMA data center networks highly depends on isolation [44], as the InfiniBand architecture does not offer a secure transport and application-level encryption (e.g., based on TLS [42]) is not possible with RDMA operations that bypass the CPU. Thus, users of Kafka that relied on TLS [42] for secure networking will not be able to securely migrate to RDMA networking. Nonetheless, a proposal for a secure RDMA transport, sRDMA [52], has been recently released to provide authentication and encryption for RDMA networking without changes to the programming interface of RDMA. If sRDMA is adopted, our KafkaDirect will offer secure networking similar to TLS by requiring only a small change to the RDMA connection establishment process.

8 CONCLUSION

This paper 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, 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 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.

ACKNOWLEDGEMENTS

This work was partially supported by the European Research Council (Project PSAP, No. 101002047), Project RED-SEA, No. 955776, and by Microsoft Research through its Swiss Joint Research Center.

REFERENCES

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

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

[3] Claude Barthels, Gustavo Alonso, and Torsten Hoefler. 2017. Designing Databases for Future High-Performance Networks. *IEEE Data Engineering Bulletin* 40, 1 (2017), 15–26.

[4] Claude Barthels, Simon Loesing, Gustavo Alonso, and Donald Kossmann. 2015. Rack-Scale In-Memory Join Processing Using RDMA. In *Proceedings of the 2015 ACM International Conference on Management of Data (SIGMOD'15)*. Association for Computing Machinery, 1463–1475. https://doi.org/10.1145/2723372.2750547

[5] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. 2016. The End of Slow Networks: It's Time for a Redesign. *Proceedings of the VLDB Endowment* 9, 7 (2016), 528–539. https://doi.org/10.14778/2904483.2904485

[6] Andrew D. Birrell and Bruce Jay Nelson. 1984. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems* 2, 1 (1984), 39–59. https://doi.org/10.1145/2080.357392

[7] Mark S Birrittella, Mark Debbage, Ram Huggahalli, James Kunz, Tom Lovett, Todd Rimmer, Keith D Underwood, and Robert C Zak. 2015. Intel® Omni-path Architecture: Enabling Scalable, High Performance Fabrics. In *Proceedings of the 23rd IEEE Symposium on High-Performance Interconnects (HOTI'15)*. IEEE Computer Society, 1–9.

[8] Matthew Burke, Sowmya Dharanipragada, Shannon Joyner, Adriana Szekeres, Jacob Nelson, Irene Zhang, and Dan R. K. Ports. 2021. PRISM: Rethinking the RDMA Interface for Distributed Systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP'21)*. Association for Computing Machinery, 228–242. https://doi.org/10.1145/3477132.3483587

[9] Haibo Chen, Rong Chen, Xingda Wei, Jiaxin Shi, Yanzhe Chen, Zhaoguo Wang, Binyu Zang, and Haibing Guan. 2015. Fast In-Memory Transaction Processing Using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*. Association for Computing Machinery, 87–104.

[10] Youmin Chen, Youyou Lu, and Jiwu Shu. 2019. Scalable RDMA RPC on Reliable Connection with Efficient Resource Sharing. In *Proceedings of the 14th EuroSys Conference (EuroSys'19)*. Association for Computing Machinery, Article 19, 14 pages. https://doi.org/10.1145/3302424.3303968

[11] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. 2020. 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)*. Association for Computing Machinery, 1077–1091. https://doi.org/10.1145/3373376.3378515

[12] Alibaba Cloud. 2018. *Super computing cluster.* https://www.alibabacloud.com/product/scc.

[13] Inc. Cloudera. 2019. *kafka-*-perf-test.* https://docs.cloudera.com/runtime/7.2.0/kafka-managing/topics/kafka-manage-cli-perf-test.html.

[14] Cong Ding, David Chu, Evan Zhao, Xiang Li, Lorenzo Alvisi, and Robbert Van Renesse. 2020. 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)*. USENIX Association, 325–338.

[15] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI'14)*. USENIX Association, 401–414.

[16] Ken Goodhope, Joel Koshy, Jay Kreps, Neha Narkhede, Richard Park, Jun Rao, and Victor Yang Ye. 2012. Building LinkedIn's Real-time Activity Data Pipeline. *IEEE Data Engineering Bulletin* 35, 2 (2012), 33–45.

[17] R. Guerraoui and A. Schiper. 1997. Total order multicast to multiple groups. In *Proceedings of the 17th International Conference on Distributed Computing Systems (ICDCS'97)*. IEEE Computer Society, 578–585.

[18] Torsten Hoefler, Salvatore Di Girolamo, Konstantin Taranov, Ryan E. Grant, and Ron Brightwell. 2017. sPIN: High-performance Streaming Processing In the Network. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'17)*. Association for Computing Machinery, Article 59, 16 pages.

[19] Sagar Jha, Jonathan Behrens, Theo Gkountouvas, Matthew Milano, Weijia Song, Edward Tremel, Robbert Van Renesse, Sydney Zink, and Kenneth P. Birman. 2019. Derecho: Fast State Machine Replication for Cloud Services. *ACM Transactions on Computer Systems* 36, 2, Article 4 (2019), 49 pages. https://doi.org/10.1145/3302258

[20] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2014. Using RDMA Efficiently for Key-Value Services. In *Proceedings of the 2014 ACM Conference on SIGCOMM (SIGCOMM'14)*. Association for Computing Machinery, 295–306. https://doi.org/10.1145/2619239.2626299

[21] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. Design Guidelines for High Performance RDMA Systems. In *Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC'16)*. USENIX Association, 437–450.

[22] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. 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)*. USENIX Association, 185–201.

[23] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a Warehouse-Scale Computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA'15)*. Association for Computing Machinery, 158–169. https://doi.org/10.1145/2749469.2750392

[24] Tejas Karmarkar. 2015. *Availability of linux RDMA on Microsoft Azure.* https://azure.microsoft.com/en-us/blog/azure-linux-rdma-hpc-available.

[25] Daehyeok Kim, Amirsaman Memaripour, Anirudh Badam, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Shachar Raindel, Steven Swanson, Vyas Sekar, and Srinivasan Seshan. 2018. 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)*. Association for Computing Machinery, 297–312. https://doi.org/10.1145/3230543.3230572

[26] Jay Kreps, Neha Narkhede, Jun Rao, et al. 2011. Kafka: A distributed messaging system for log processing. In *Proceedings of the 2011 IEEE International Workshop on Networking Meets Databases (NetDB'11)*. 1–7.

[27] Ilya Lesokhin, Haggai Eran, Shachar Raindel, Guy Shapiro, Sagi Grimberg, Liran Liss, Muli Ben-Yehuda, Nadav Amit, and Dan Tsafrir. 2017. Page fault support for network controllers. *ACM SIGARCH Computer Architecture News* 45, 1 (2017), 449–466.

[28] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. 2017. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*. Association for Computing Machinery, 137–152. https://doi.org/10.1145/3132747.3132756

[29] Feng Li, Sudipto Das, Manoj Syamala, and Vivek R. Narasayya. 2016. Accelerating Relational Databases by Leveraging Remote Memory and RDMA. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD'16)*. Association for Computing Machinery, 355–370. https://doi.org/10.1145/2882903.2882949

[30] Joshua Lockerman, Jose M. Faleiro, Juno Kim, Soham Sankaran, Daniel J. Abadi, James Aspnes, Siddhartha Sen, and Mahesh Balakrishnan. 2018. The FuzzyLog: A Partially Ordered Shared Log. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)*. USENIX Association, 357–372.

[31] Microsoft. 2020. *Azure Service Bus Messaging.* https://azure.microsoft.com/en-us/services/service-bus/.

[32] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. 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)*. USENIX Association, 103–114.

[33] The Ohio State University Network-Based Computing Laboratory. 2018. *RDMA-based Apache Kafka (RDMA-Kafka).* http://hibd.cse.ohio-state.edu/#kafka.

[34] Oracle. 2020. *Oracle Cloud.* https://www.oracle.com/cloud/hpc/.

[35] Oracle. 2020. *Oracle Messaging Cloud Service.* https://www.oracle.com/technical-resources/articles/cloud/wilkins-ocms.html.

[36] 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. 2015. The RAMCloud Storage System. *ACM Transactions on Computer Systems* 33, 3, Article 7 (Aug. 2015), 55 pages. https://doi.org/10.1145/2806887

[37] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. 2015. Making Sense of Performance in Data Analytics Frameworks. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI'15)*. USENIX Association, 293–307. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/ousterhout

[38] Sathish K Palaniappan and Pramod B Nagaraja. 2008. Efficient data transfer through zero copy. *IBM developerworks* (2008).

[39] Marius Poke and Torsten Hoefler. 2015. 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)*. Association for Computing Machinery, 107–118. https://doi.org/10.1145/2749246.2749267

[40] OpenMessaging Project. 2017. *OpenMessaging Benchmark Framework.* https://github.com/openmessaging/openmessaging-benchmark.

[41] Renato Recio, Bernard Metzler, Paul Culley, Jeff Hilland, and Dave Garcia. 2007. *A Remote Direct Memory Access Protocol Specification.* Technical Report RFC 5040. Network Working Group.

[42] Eric Rescorla. 2018. *The Transport Layer Security (TLS) Protocol Version 1.3.* Technical Report RFC 8446. Network Working Group.

[43] Wolf Rödiger, Tobias Mühlbauer, Alfons Kemper, and Thomas Neumann. 2015. High-Speed Query Processing over High-Speed Networks. *Proceedings of the*

VLDB Endowment 9, 4 (Dec. 2015), 228–239. https://doi.org/10.14778/2856318.2856319

[44] Benjamin Rothenberger, Konstantin Taranov, Adrian Perrig, and Torsten Hoefler. 2021. ReDMArk: Bypassing RDMA Security Mechanisms. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security'21)*. USENIX Association.

[45] Amazon Web Services. 2020. *Amazon Simple Queue Service*. https://aws.amazon.com/sqs/.

[46] Peter Snyder. 1990. tmpfs: A virtual memory file system. In *Proceedings of the Autumn 1990 European UNIX Users' Group Conference (EUUG'90)*. 241–248.

[47] Patrick Stuedi, Bernard Metzler, and Animesh Trivedi. 2013. JVerbs: Ultra-Low Latency for Data Center Applications. In *Proceedings of the 4th ACM Symposium on Cloud Computing (SoCC'13)*. Association for Computing Machinery, Article 10, 14 pages. https://doi.org/10.1145/2523616.2523631

[48] Patrick Stuedi, Animesh Trivedi, Bernard Metzler, and Jonas Pfefferle. 2014. DaRPC: Data Center RPC. In *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC'14)*. Association for Computing Machinery, 1–13. https://doi.org/10.1145/2670979.2670994

[49] Maomeng Su, Mingxing Zhang, Kang Chen, Zhenyu Guo, and Yongwei Wu. 2017. RFP: When RPC is Faster than Server-Bypass with RDMA. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys'17)*. Association for Computing Machinery, 1–15. https://doi.org/10.1145/3064176.3064189

[50] Yacine Taleb, Ryan Stutsman, Gabriel Antoniu, and Toni Cortes. 2018. Tailwind: Fast and Atomic RDMA-based Replication. In *Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC'18)*. USENIX Association, 851–863.

[51] Konstantin Taranov, Rodrigo Bruno, Gustavo Alonso, and Torsten Hoefler. 2021. Naos: Serialization-free RDMA networking in Java. In *Proceedings of the 2021 USENIX Annual Technical Conference (USENIX ATC'21)*. USENIX Association, 1–14.

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

[53] Mellanox Technologies. 2015. *RDMA Aware Networks Programming User Manual, Rev 1.7*. https://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf.

[54] Gigabyte Technology. 2021. *AORUS Gen4 AIC SSD 8TB*. https://www.gigabyte.com/Solid-State-Drive/AORUS-Gen4-AIC-SSD-8TB.

[55] Hung-Wei Tseng, Qianchen Zhao, Yuxiao Zhou, Mark Gahagan, and Steven Swanson. 2016. Morpheus: Creating Application Objects Efficiently for Heterogeneous Computing. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA'16)*. IEEE Press, 53–65. https://doi.org/10.1109/ISCA.2016.15

[56] Giselle van Dongen and Dirk Van den Poel. 2020. Evaluation of Stream Processing Frameworks. *IEEE Transactions on Parallel and Distributed Systems* 31, 8 (2020), 1845–1858.

[57] Guozhang Wang, Joel Koshy, Sriram Subramanian, Kartik Paramasivam, Mammad Zadeh, Neha Narkhede, Jun Rao, Jay Kreps, and Joe Stein. 2015. Building a Replicated Logging System with Apache Kafka. *Proceedings of the VLDB Endowment* 8, 12 (Aug. 2015), 1654–1655. https://doi.org/10.14778/2824032.2824063

[58] 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. 2017. 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)*. USENIX Association, 35–49.

[59] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. 2018. Deconstructing RDMA-enabled Distributed Transactions: Hybrid is Better!. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*. USENIX Association, Carlsbad, CA, 233–251. https://www.usenix.org/conference/osdi18/presentation/wei

[60] Jilong Xue, Youshan Miao, Cheng Chen, Ming Wu, Lintao Zhang, and Lidong Zhou. 2019. Fast Distributed Deep Learning over RDMA. In *Proceedings of the 14th European Conference on Computer Systems (EuroSys'19)*. Association for Computing Machinery, Article 44, 14 pages. https://doi.org/10.1145/3302424.3303975

[61] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud'10)*. USENIX Association, 10.

[62] Erfan Zamanian, Carsten Binnig, Tim Harris, and Tim Kraska. 2017. The End of a Myth: Distributed Transactions Can Scale. *Proceedings of the VLDB Endowment* 10, 6 (2017), 685–696. https://doi.org/10.14778/3055330.3055335

[63] Erfan Zamanian, Xiangyao Yu, Michael Stonebraker, and Tim Kraska. 2019. Rethinking Database High Availability with RDMA Networks. *Proceedings of the VLDB Endowment* 12, 11 (July 2019), 1637–1650. https://doi.org/10.14778/3342263.3342639

[64] Tobias Ziegler, Sumukha Tumkur Vani, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. Designing Distributed Tree-Based Index Structures for Fast RDMA-Capable Networks. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD'19)*. Association for Computing Machinery, 741–758. https://doi.org/10.1145/3299869.3300081